Gestion hétérogène des données dans les hiérarchies mémoires pour l'optimisation énergétique des architectures multi-coeurs

Gregory Vaumourin

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Grégory Vaumourin**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Gestion hétérogène des données dans les hiérarchies mémoires pour l'optimisation énergétique des architectures multi-cœurs

---

Date de soutenance : 4 Octobre 2016

Devant la commission d'examen composée de :

Olivier Sentieys	Directeur de Recherche, INRIA CAIRN	Rapporteur
Albert Cohen	Directeur de Recherche, INRIA PARKAS	Rapporteur
Alexandra Jimborean	Chargée de Recherche, Université d'Uppsala	Examinateur
William Jalby	Professeur des Université, UVSQ	Examinateur
Alexandre Guerre	Ingénieur, Trinnov Audio	Encadrant
Thomas Dombek	Ingénieur Chercheur, CEA LIST	Encadrant
Denis Barthou	Directeur de Recherche, INRIA STORM	Directeur de Thèse

– 2016 –

Résumé de la Thèse

Résumé

Titre : Gestion hétérogène des données dans les hiérarchies mémoires pour l'optimisation énergétiques des architectures multi-coeurs.

Résumé : Les problématiques de consommation dans la hiérarchie mémoire sont très présentes dans les architectures actuelles que ce soit pour les systèmes embarqués limités par leurs batteries ou pour les supercalculateurs limités par leurs enveloppes thermiques. Introduire une information de classification dans le système mémoire permet une gestion hétérogène, adaptée à chaque type particulier de données. Nous nous sommes intéressé dans cette thèse plus précisément aux données en lecture seule et étudions les possibilités d'une gestion spécifique dans la hiérarchie mémoire à travers un codesign compilation/architecture. Cela permet d'ouvrir de nouveaux potentiels en terme de localité des données, passage à l'échelle des architectures ou design des mémoires. Evaluée par simulation sur une architecture multi-coeurs, la solution mise en oeuvre permet des gains significatifs en terme de réduction de la consommation d'énergie à performance constante.

Mots clés : Hiérarchie mémoire, cache, localité des données, protocoles de cohérence

Abstract

Title : Read Only Data Specific Management for an Energy Efficient Memory System

Abstract : The energy consumption of the memory system in modern architectures is a major issue for embedded system limited by their battery or supercalculators limited by their Thermal Design Power. Using a classification information in the memory system allows a heterogeneous management of data, more specific to each kind of data. During this thesis, we focused on the specific management of read-only data into the memory system through a compilation/architecture codesign. It allows to explore new potentials in terms of data locality, scalability of the system or cache designs. Evaluated by simulation with multi-core architecture, the proposed solution offers significant energy consumption reduction while keeping the performance stable.

Keywords : Memory System, Cache Memory, data locality, coherence protocol

Remerciements

"Le vrai dévouement s'offre non pas par étincelle, mais chaque jour, contre l'oubli."
Zachary Hoel

Je tiens tout d'abord à témoigner ma gratitude à M. Olivier Sentieys et M. Albert Cohen, pour avoir bien voulu juger mes travaux en tant que rapporteurs et merci également à Mme Alexandra Jimborean et M. William Jalby pour avoir accepté de se joindre au jury de ma thèse en tant qu'examinateurs.

J'ai réalisé mes travaux de thèse d'abord au sein du Laboratoire du Calcul Embarqué (LCE) puis au sein du Laboratoire Adéquation Algorithme Architecture (L3A), les deux laboratoires faisant parti du Département Architecture Conception et Logiciels (DACLE) du Commissariat à l'Energie Atomique et aux Energies Alternatives (CEA) de Saclay. C'est pourquoi je voudrais d'abord remercier Raphaël David et Thomas Dombek de m'avoir accueilli dans leurs laboratoires respectifs et permis d'effectuer ma thèse dans les meilleures conditions.

Ma reconnaissance va à Denis Barthou pour m'avoir fait l'honneur de diriger ma thèse. Il fut un directeur de thèse précieux qui a marqué de façon importante ces travaux de part son expérience et son expertise. Je remercie également chaleureusement Alexandre Guerre pour son investissement et son soutien sans faille pendant 3 ans. Avoir accepté de continuer à encadrer ma thèse malgré des changements professionnels en est une preuve évidente. Enfin, je témoigne ma gratitude à Thomas Dombek qui a accepté de prendre en charge cette thèse, pour son écoute et son expertise. Quand bien même l'auteur signe ce manuscrit de son nom seul, l'existence de ce document dépend largement des 3 noms précédemment évoqués et l'importance de leurs contributions ne saurait être oubliée en lisant la suite de ce document.

Amical clin d'oeil à l'ensemble du personnel des laboratoires L3A et LCE pour leurs accueil durant ces 3 années ainsi qu'aux doctorants actuels ou passés. Je n'aurai pas assez de cette page pour remercier toutes les personnes qui m'ont aidé durant cette thèse.

Je voudrais remercier mes amis et ma famille, et en particulier mon frère, pour leurs soutiens dans les moments difficiles de cette thèse, qui restent si proches et pourtant si souvent inaccessibles. Je profite également de ces remerciements pour rendre hommage à mon père décédé durant l'écriture de ce manuscrit. Son soutien inconditionnel m'aura permis d'avancer jusqu'ici et je ne peux qu'imaginer sa fierté de me voir réussir ici.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Sommaire

"Idéalement, nous utiliserions une mémoire de taille infinie et dont chaque donnée est accessible instantanément. Nous sommes forcés de réaliser la possibilité de contruire une hiérarchie de mémoire, chacune étant plus grande que la précédente mais accessible moins rapidement"

A. W. Burks, H. H. Goldstine et J. Von Neumann
Discussion préliminaires pour le design des circuits logiques
d'un instrument de calcul électronique (1946)

Ce chapitre décrit d'abord brièvement l'évolution des architectures et de la hiérarchie mémoire en particulier. Des éléments de contexte plus spécifique sont ensuite introduits afin de permettre de poser la problématique de la thèse ainsi que les contributions et le plan du manuscrit.

1.1 Evolution des architectures

1.1.1 Apparition de la hiérarchie mémoire

Les premiers processeurs pouvaient accéder directement à la mémoire car ce temps d'accès était à peine plus long qu'un accès aux registres. Avec l'augmentation importante des fréquences des processeurs dans les années 1980, une différence de performance entre la mémoire et le processeur apparait. La vitesse de calcul des architectures n'est plus alors limitée par les capacité du processeur mais par le temps d'accès en mémoire aux données à calculer. Cette asymétrie, illustrée figure 1.1, est appelé le *Memory Wall*. . Afin de palier à ce problème, plusieurs solutions matérielles et logicielles ont été proposées comme l'exécution dans le désordre (ooo), le calcul dans la mémoire (PIM), ou le pré-chargement mais la plus importante reste la hiérarchie mémoire. Son principe est de disposer des mémoires plus petites et plus rapides à côté du processeur qui stockent les données les plus réutilisées par le processeur. Le processeur va chercher les données dans cette mémoire si elles y sont présentes plutôt que dans la mémoire centrale. Si la donnée n'est pas présente en cache, on parle alors de défauts de cache qu'il s'agira donc de minimiser. Ainsi, ces petites mémoire appelées mémoire cache ou plus simplement cache permettent de "cacher" les temps d'accès importants à la mémoire principale. Cette gestion est automatisée au niveau matériel et donc transparente vis-à-vis de l'utilisateur. Cela a permis d'augmenter la capacité des mémoires tout en gardant des temps d'accès moyen à la mémoire acceptables. L'utilisation de ces mémoires se développe avec l'apparition de plusieurs niveaux de cache, chaque niveau étant de plus en plus volumineux et donc de plus en plus lent à mesure que l'on s'éloigne du processeur. Ces mémoires adoptent donc une organisation pyramidale de plusieurs niveaux, désignée sous le terme de hiérarchie mémoire. Cette hiérarchie de caches a évolué en même temps que la complexité des architectures. On décrit d'abord brievement l'évolution des architectures avant de s'intéresser plus spécifiquement à la hiérarchie mémoire.

Avant les années 2000, l'évolution des performances des processeurs étaient rythmés par la loi de Moore [7]. Cette loi, plutôt une conjecture, énoncée par Gordon E. Morre en 1965, observait la tendance du nombre de transistors utilisés pour les processeurs à doubler tous les 18 mois. Cette augmentation du nombre de transistors permettait l'existence de processeurs toujours plus complexes. Les fondeurs tels qu'Intel et AMD ont ainsi profité du doublement régulier de la densité des processeurs afin d'accroitre la complexité des architectures et augmenter les performances de calcul. La théorie du passage à l'échelle de Dennard [8] anticipait que l'on pouvait baisser la tension d'alimentation lorsque ceux-ci étaient gravé plus finement, la densité de puissance (mesurée en W/cm^2) reste alors constante. Couplée à la loi de Moore, elle prévoyait un doublement du nombre de transistors tous les deux ans et une élévation de la fréquence de 40% tout en conservant une puissance électrique équivalente stable. Cependant, depuis le début des années

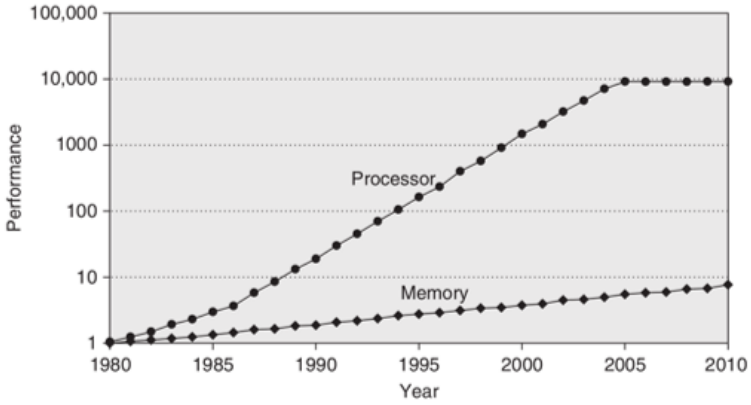


FIGURE 1.1 – Evolution asymétrique de la performance des processeurs et des mémoires depuis les années 1980[1]

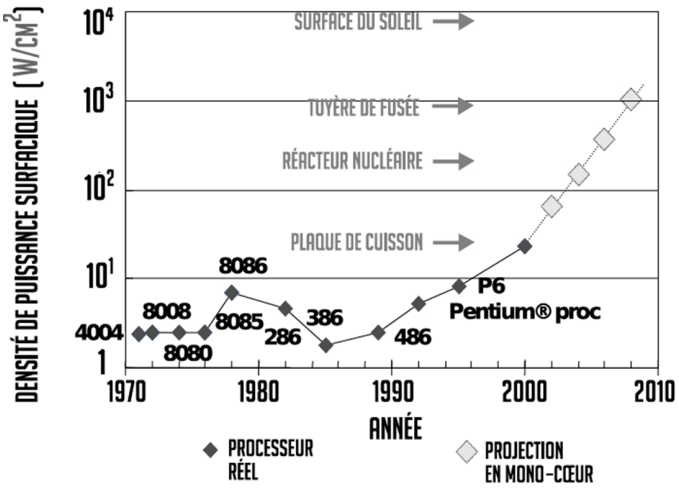


FIGURE 1.2 – Projection de la puissance des micro-processeurs avant l'introduction des multi-coeurs (*source : S. Borkar, Intel*)

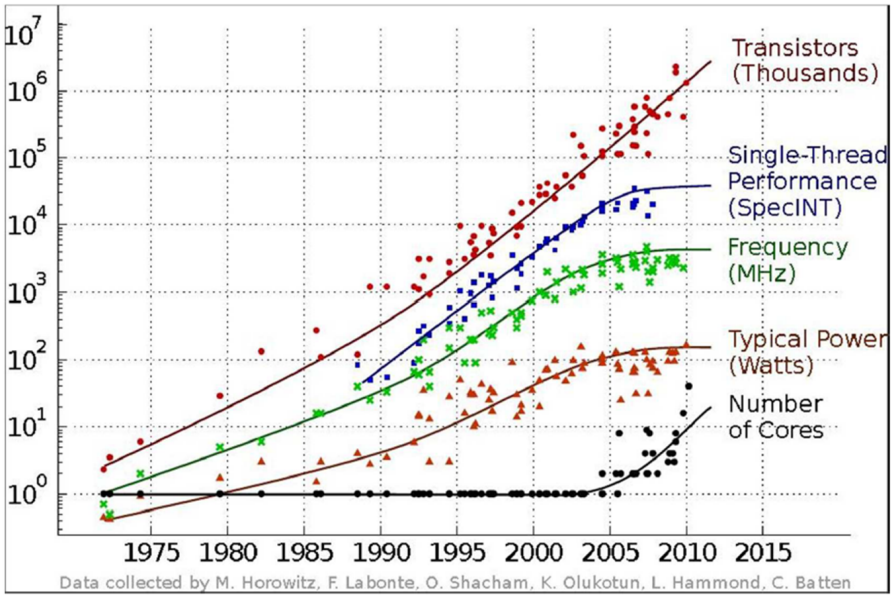


FIGURE 1.3 – Evolution du nombre de transistors dans les processeurs et conséquences

2000, ce facteur de performance a rencontré des limitations majeures. Afin de comprendre ce phénomène, il faut descendre au niveau transistor, composant de base d'un circuit numérique. Ces transistors portent une information binaire (0 ou 1), et une puissance est dégagée lorsque que celui-ci commute (ou change d'état). Cette puissance est calculée par la formule :

$$P = 1/2 * (\text{Capacité de charge}) * (\text{Fréquence du processeur}) * (\text{Tension d'alimentation})^2$$

Ainsi la figure 1.2 effectue une projection de l'ampleur de la puissance thermique à dissiper par extrapolation de la vitesse d'évolution de la fréquence. La densité de puissance diverge et le passage à l'échelle de Dennard est alors abandonné. Des modèles radicalement différents ont dû alors émerger en matière d'architecture aussi bien pour les processeurs que pour la hiérarchie mémoire associée. .

1.1.2 Des systèmes séquentiels aux multi-processeurs sur puce

Ainsi, depuis le début des années 2000, les architectures ont adopté le modèle du multi-processeur sur puce comme modèle dominant pour leurs architectures comme on peut le voir sur la figure 1.3. En effet, en réduisant les fréquences de fonctionnement, il devient donc possible de combiner les capacités de calculs de plusieurs processeurs sur une même puce, sans augmenter la puissance thermique à dissiper. Cet ajout de processeurs au niveau de l'architecture peut se faire de deux façons.

L'approche symétrique repose sur une juxtaposition de coeurs de calculs similaires. Le recours à la combinaison de tous les coeurs sert alors à accélérer les portions parallèles de l'application et le fait que les ressources de calcul soient uniformes permet théoriquement de faciliter leur programmation car le calcul peut ainsi être effectué indifféremment sur chacun des processeurs. Cette approche définit la catégorie des multi-processeurs symétriques ou architecture SMP. On trouve dans cette catégorie, parmi les processeurs embarqués, la famille de processeurs des Cortex-A de ARM avec laquelle il est possible d'associer jusqu'à 4 coeurs identiques sur une même puce depuis la deuxième génération de cette famille de processeurs, on parle alors de système *MPCore*. Cette idée a été améliorée en 2013 avec la famille suivante des Cortex-A50 qui permet d'associer jusqu'à 16 coeurs Cortex-53 ou Cortex-A57 en parallèle sur une même puce. Ce type d'architecture se destine au marché des smartphones ou des tablettes. On retrouve également ce même principe dans le domaine du calcul haute performance. Ainsi, Intel sort, en 2012, une puce appelée *Xeon Phi* basée sur l'architecture Larrabee [9]. Elle est composée de 61 processeurs identiques reliés entre eux par un réseau d'interconnexion en anneau, chacun pouvant exécuter de larges instructions vectorielles. Cette architecture offre donc un parallélisme important, similaire à ce que l'on retrouve dans les processeurs graphiques (GPU), mais en plus, offre une compatibilité avec les modèles de programmation parallèles standards et ne requiert donc pas de réécriture complète du code. Ainsi, on retrouve cette puce dans le supercalculateur *Tianhe-2*, premier en Novembre 2015 au classement TOP500, il est composé de seize mille noeuds, comportant chacun trois *Xeon Phi*. Parmi les autres représentants de ce type d'architecture, on peut également citer le Tile64 de Tilera [10], le Magny-Cours [11] d'AMD, ou le POWER5 [12] d'IBM.

La seconde approche à être apparu est l'approche asymétrique. Elle utilise des processeurs spécifiques pour accélérer certains types de calculs. Parmi les accélérateurs matériels couramment utilisés, on trouve les processeurs spécialisés dans le traitement de signal numérique (DSP), ou les processeurs graphiques. Ainsi, il ne s'agit pas dans ce cas d'exploiter le parallélisme dans une application mais de répondre à un besoin applicatif précis. Cela permet une meilleure efficacité de la solution au niveau énergétique et performance, car les portions de code s'exécutent sur un processeur adapté au calcul à effectuer. L'allocation des tâches sur ces ressources est souvent statique car l'absence d'homogénéité complique la répartition efficace de la charge de calcul entre plusieurs ressources. Par ailleurs, les communications entre des éléments de nature différente sont difficiles à mettre en oeuvre, même si l'utilisation d'un contrôleur centralisé réduit ce besoin. On trouve, comme exemple, dans cette classe d'architecture pour les systèmes embarqués, les processeurs Tegra de NVIDIA présent dans les tablettes et des smartphones. La version de dernière génération de ce processeur, la version Tegra X1, embarque sur une même puce, quatre processeurs ARM A53 et quatre processeurs A57. Cette asymétrie de capacité de calcul entre les deux groupes de processeurs permet d'exécuter le code sur le groupe approprié selon la charge de calcul à effectuer. Cette optimi-

sation appelée *big.LITTLE* a pour objectif de réduire la consommation du système. Ces processeurs sont également associés à un GPU de 256 coeurs, ce GPU instancié avec l'architecture Maxwell permet principalement de déporter les fonctions de traitement d'image et de vidéo. D'autres exemples de ce type d'architectures sont la famille S6000 de Tensilica [13] ou la série des OMAP de Texas Instrument [14]

Ainsi, les deux approches sont présentes dans les systèmes embarqués et apportent des défis différents sur la hiérarchie mémoire. Dans le cadre des travaux de cette thèse, nous nous intéresserons aux multi-processeurs symétriques sur puce.

1.2 Organisation hiérarchique de la mémoire

Dans une organisation hiérarchie de la mémoire tel qu'évoqué précédemment, la mémoire centrale sert en quelques sorte de dépôt pour les données de calcul. Elle ne suffit pas à elle seule à maintenir un état d'occupation suffisant des unités de calcul à cause de ses temps d'accès relativement élevés. De plus, l'augmentation du nombre de coeurs, implique un besoin en données de plus en plus conséquent. La hiérarchie mémoire est donc cruciale pour le bon fonctionnement du système permet donc de masquer une partie de ces latences élevées. Dans les architectures SMP, tous les processeurs ont une vue globale de la mémoire centrale. L'enjeu est de conserver cette caractéristique, tout en minimisant les points de contention sur la mémoire. En effet, cette dernière doit répondre à un besoin croissant en bande passante pour alimenter les processeurs de plus en plus nombreux. Afin de résoudre ce problème, deux modèles de mémoire s'opposent alors : le modèle à mémoire distribuée ou à mémoire partagée.

1.2.1 Modèle de mémoire distribuée

Cette première solution consiste à distribuer logiquement la mémoire entre tous les processeurs. Cette solution est utilisée depuis bien longtemps pour les supercalculateurs et a été adoptée à l'échelle des multi-processeurs sur puce. Chaque processeur possède ainsi son propre espace d'adressage dans lequel il peut effectuer ses calculs propres et il n'y a donc pas d'accès concurrents entre processeurs pour accéder à la mémoire. Cette solution a été proposée pour alléger le goulot d'étranglement vers la mémoire qui représente une architecture à mémoire partagée. Reste que bien souvent, les tâches ont besoin de partager des données. Dans ce cas, il faut mettre en place une communication explicite par envoi de messages. Les connexions sont alors prises en charge par les systèmes d'exploitation ou des couches logicielles et nécessitent la mise en place de tampons mémoire de communication pour accueillir et préparer l'envoi des données véhiculées. Ce processus peut se révéler pénalisant en matière de performance, et de ressource mémoire. De plus, comme dans tout système distribué, les ressources mémoires allouées à chacun peuvent également ne pas être exploitées de façon optimale. La latence

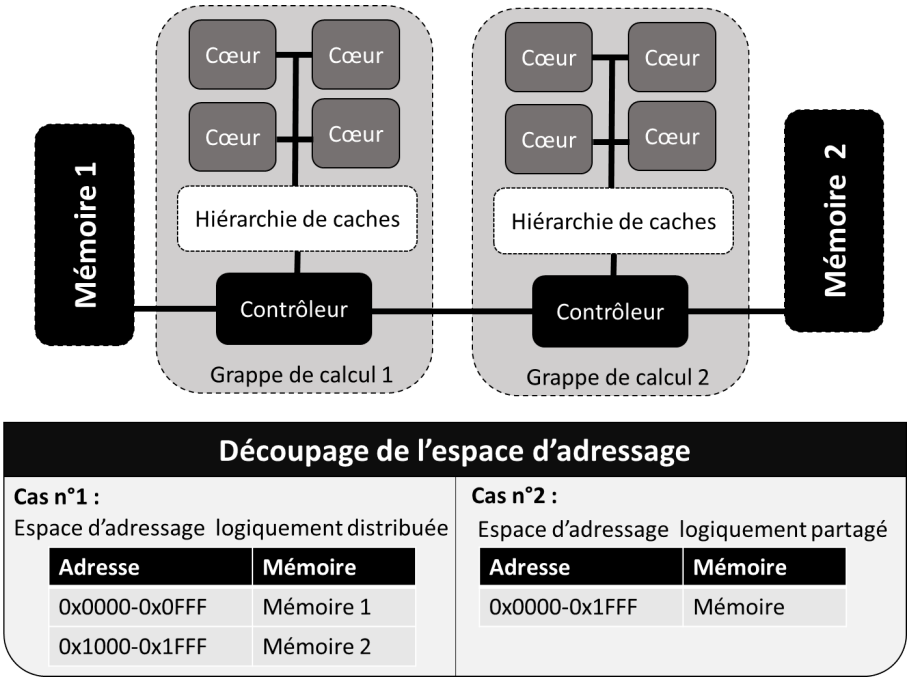


FIGURE 1.4 – Exemple d'architecture à mémoire physiquement distribuée avec les deux cas de partitionnement de l'espace mémoire

étant différente selon le bloc que l'on veut accéder, cela crée des latences différentes selon la mémoire accédée. On parle alors d'architecture NUMA [1].

Une solution pour améliorer la programmation des systèmes à mémoire distribuée consiste à partager logiquement la mémoire entre tous les processeurs. Ainsi, les processeurs travailleront sur un espace d'adressage commun ne requérant pas d'envoi/réception de messages, les processeurs pouvant alors directement accéder à l'ensemble de l'espace d'adressage commun. Ce modèle de mémoire est surtout utilisé pour distribuer la mémoire à des processeurs rassemblés en grappe de calcul (cluster). Alors chaque grappe de calcul possède ses propres ressources enter de de mémoire et de bande passante. Ainsi, dans l'exemple développé figure 1.4, selon qu'un processeur de la grappe de calcul 1 accède à la mémoire 1 ou la mémoire 2, le temps sera différent.

1.2.2 Modèle de mémoire partagée

Afin de s'affranchir de la pénalité des communications entre tâches, un autre modèle peut être utilisé. Celui-ci repose sur une mémoire partagée entre tous les processeurs de la puce et tous les processeurs possèdent donc le même espace d'adressage. Le partage des données s'effectue par lecture/écriture des données

1. *Non-uniform Memory Access Latency* : Temps d'accès à la mémoire non uniforme

partagées et du fait que plusieurs processeurs peuvent accèder à la même donnée en en même temps, il faut gérer les accès concurrents à la mémoire partagée. Si la mémoire offre des accès multiples, il faut par ailleurs protéger les données en écriture car deux écritures simultanées à un même emplacement mémoire peut provoquer une incohérence des données. Les communications par mémoire partagée peuvent être alors très efficaces dans les architectures SMP dès lors que les coeurs de processeur et les caches sont intégrés sur la même puce.

Dans de telles architectures, la hiérarchie mémoire développée s'appuie sur une imbrication entre caches privés et partagés. Cette organisation varie en fonction du nombre de coeurs et de la ressource en cache disponible. La figure 1.5 illustre deux exemples d'architectures, les architecture Dunnington et Happertown d'Intel. On observe que plus l'on s'approche du processeur, plus les caches ont tendance à être privés. Les caches proche processeurs sont plus critique en terme de ressources car accédés plus souvent et c'est pourquoi on retrouve plus souvent des caches privés. En effet, un cache privé permet de mieux exploiter le principe de localité des données utilisées par le processeur (que l'on détaillera plus loin) et possède ses propres ressources en termes de port et de bande passante ce qui lui permet d'être accédé par son processeur sans conflit avec les autres processeurs. Un cache partagé permet, quant a lui, une meilleure utilisation de la ressource en cache du fait de sa mutualisation et un partage des blocs de cache entre plusieurs processeurs facilités mais peut également conduire à des conflits d'accès ou de ressources entre processeurs.

Ce type de hiérarchie mémoire suppose que le processeur copie les données dans son cache L1 local avant de les utiliser. Ce mécanisme peut créer des incohérences dans le cas ou le processeur écrit une nouvelle version de la donnée dans son cache L1 sans que le reste du système soit informés. Si la mise à jour n'est pas propagée, chaque processeur travaille sur différentes versions de la donnée et cela produit des résultats incohérents. Afin d'éviter de tels effets, un protocole de cohérence peut être implémenté afin de garantir la vue cohérente de la mémoire par chacun des processeurs. Ce protocole permet un partage automatique des blocs de cache entre processeurs sur des architectures multi-processeurs. Cette gestion a un cout qui augmente avec l'augmentation du nombre de coeurs à gérer, son implémentation doit donc être la plus transparente possible. Ces protocoles sont donc une des composantes du système limitant le passage à l'échelle des architectures.

1.2.3 Les protocoles de cohérence

Un protocole de cohérence s'appuie sur des communications complexes entre les contrôleurs des différents caches pour garantir la cohérence du système. La règle de base que doit respecter un protocole de cohérence est formulée par Sorin, Hill et Wood appelé "Une seul peut écrire, plusieurs peuvent lire" ("the Single-Writer/Multiple-Reader") [15] est énoncé comme suit : "Pour chaque emplacement

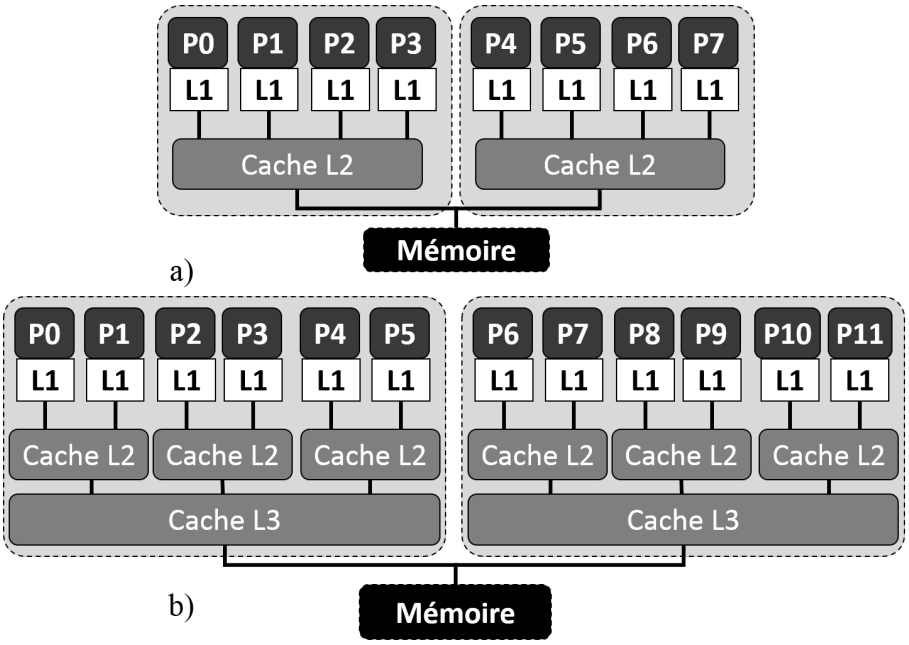


FIGURE 1.5 – Description de la hiérarchie mémoire des architectures INTEL a) Happertown et b) Dunnington

mémoire, à tout instant, un seul processeur peut l'écrire (et éventuellement la lire) ou plusieurs processeurs peuvent lire simultanément". Autrement dit, à un instant donné, soit un processeur possède les droits en lecture/écriture sur une donnée, soit plusieurs processeurs possèdent les droits de lecture à cette donnée. Ainsi, il ne peut y avoir simultanément une écriture par un processeur et des lectures/écritures par d'autres, la cohérence du système est donc préservée. La cohérence fonctionne habituellement à une granularité bloc de cache, le mot "donnée" dans la définition ci-dessus, est donc à comprendre comme un bloc de cache.

On peut distinguer deux parties dans le protocole de cohérence :

— Le protocole de cohérence proprement dit qui définit les différents états possibles et les transitions pour chaque bloc de mémoire. Un graphe d'état modélise les différentes transitions. Beaucoup de protocoles utilisent un sous-ensemble du modèle proposé par Sweazey et Smith [16], ces protocoles utilisent à minima les état Modifié (état M), Partagé (état S), ou Invalide (état I). Des optimisations peuvent ensuite rajouter les états Possédé (état O), Exclusif (état E), ... Ce sont les états dits stables du protocole, il faut également rajouter les états transitoires, les changements entre états stables n'étant pas immédiats.

— Le mécanisme de cohérence qui correspond à l'implémentation matérielle du protocole, il définit notamment les communications entre les différents contrôleurs, ou est stocké l'état de la variable, etc ...

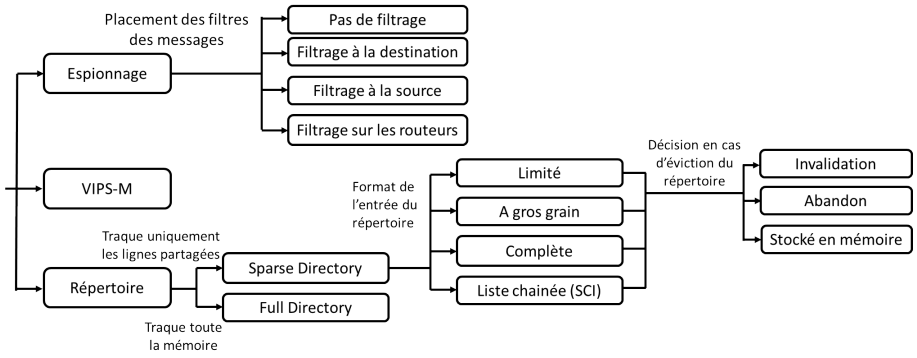


FIGURE 1.6 – Classification des mécanismes de cohérence

Une classification des mécanismes de cohérence est proposée figure 1.6. Elle distingue 3 principaux mécanismes de cohérence que nous détaillons ci-dessous.

1.2.3.1 Les mécanismes à base d'espionnage

Avec ce type de mécanisme, les caches privés doivent communiquer directement les uns avec les autres pour se partager les données. Le cache partagé n'intervient en rien dans la gestion de la cohérence. Dans le cas où les caches privés sont reliés via un bus, cette méthode est particulièrement efficace, car chaque cache peut espionner via le bus, les transactions qui s'effectuent sur les autres caches privés. Dans le cas d'un réseau d'interconnexion de type réseau sur puce, des messages doivent alors être diffusés à tous les caches privés à chaque transaction qui modifie la vue cohérente de l'espace d'adressage. Cette classe de mécanisme présente l'avantage d'effectuer des transferts entre caches plutôt que par le cache partagé ce qui est beaucoup plus rapide et elle ne requiert pas de stockage important dédié à la cohérence. Mais du fait de l'obligation de diffusion à tous les processeurs de toutes les transactions modifiant la cohérence du système, énormément de messages de contrôle doivent être envoyés sur le réseau, ce qui entraine un surcout important en terme de communications. Beaucoup d'optimisations cherchent à réduire ces messages en les filtrant ce qui aboutit à la sous-classification de la Figure 1.6 selon l'emplacement des filtres sur le réseau. Du à ces problèmes de passage à l'échelle, ce mécanisme est en général limité à des architectures avec un petit nombre de cœurs. On retrouve des implémentations de type de mécanisme d'espionnage notamment dans le Starfire E10000 [17] de Sun Microsystems ou le POWER5 [12] d'IBM.

1.2.3.2 Les mécanismes par répertoire

Avec une philosophie opposée, les mécanismes par répertoire proposent une gestion centralisée au niveau du cache partagé. Le répertoire associe à chaque bloc de cache, une entrée permettant de stocker l'état de la variable selon le protocole de

cohérence et la liste des processeurs partageant cette donnée. Cette information est souvent stockée dans une partie spécifique du cache partagé ou directement dans un cache séparé à côté du cache partagé, le répertoire désigne alors ce stockage spécifique. Lorsqu'une requête atteint le cache partagé, le répertoire est interrogé pour connaître l'état de partage de ce bloc. Le contrôleur de cache arbitre alors les différentes requêtes et gère l'émission des messages d'invalidations ou de mise des jours des blocs dans les caches privés, il peut également rediriger les requêtes si le bloc de cache demandé est présent dans un autre cache L1, on parle alors d'indirections. Un répertoire complet (*full-parse*) attribue à chaque bloc de donnée de l'espace d'adressage une entrée de répertoire, cette information peut être stockée directement dans la mémoire principale. Même en supposant une entrée de répertoire compacte, cette méthode requiert une quantité importante d'espace de stockage supplémentaire, elle est peu utilisable en pratique lorsque le nombre de données utilisé est important. Un répertoire clairsemé (*sparse directory*) stocke uniquement les blocs de cache activement partagés. Parmi les choix importants de design d'un répertoire clairsemé, on trouve la façon dont l'information sur les processeurs partageant le bloc de cache est encodée dans l'entrée de répertoire. La solution peut être d'utiliser un bit par processeur (entrée *Full-map*), ou de stocker uniquement les k premiers (entrée *limited directory*). Si ce dernier est partagé par un nombre de processeurs supérieur à k, alors le mécanisme de cohérence suppose que tous les processeurs partagent la donnée. On trouve beaucoup d'autres optimisations permettant de stocker efficaces les processeurs partageant la donnée dans l'entrée du répertoire. Les mécanismes par répertoire sont considérés comme étant la solution permettant un meilleur passage à l'échelle mais les implémentations restent encore couteuse en matière d'énergie, de performance, et de surface. Parmi les architectures commercialisé avec un tel mécanisme, l'une des premiers exemples est le Origin 2000[18] de SGI, commercialisé dans les années 1990 prévu pour aller jusqu'à 1024 coeurs. Plus récemment, on peut citer le protocole *Coherent HyperTransport* d'AMD implémenté pour son architecture Magny-Cours[11] à 12 coeurs.

1.2.3.3 Les mécanisme VIPS-M

Face à la gestion lourde des mécanismes à répertoire, un nouveau mécanisme de cohérence a été proposé au niveau de la recherche depuis quelques années, ne nécessitant plus de répertoires [19, 20, 21]. Cette nouvelle catégorie de mécanisme de cohérence appelée VIPS-M [2] utilise une classification des données qui permet d'identifier les blocs de cache qui ont un vrai besoin de cohérence, c'est-à-dire les données partagées en lecture-écriture. Ainsi, cette détection est effectuée par de façon réactive au niveau page avec quelques bits ajoutés dans les pages afin d'encoder cette information de classement. Ainsi, les pages du système sont initialisée avec un statut privé et dès qu'un accès partagé est détecté, la page change de statut

2. *Valid-Invalid, Private-Shared* : mécanisme de cohérence sur l'invalidation (plutôt que la promotion) et le classement entre privé et partagée

et passe en statut partagé. L'idée est donc d'utiliser la cohérence uniquement pour les pages classées en mode partagées. Pour cela, les blocs mémoire de ces pages propagent systématiquement leurs écritures au cache partagé ce qui lui permet d'avoir toujours une version à jour du bloc et de pouvoir gérer ainsi plus facilement la cohérence du système. Pour les autres blocs, ce système de propagation immédiate d'écriture n'est pas utilisé car trop couteux, l'écriture est alors propagée lors de l'éviction du bloc du cache privé. Associés à d'autres mécanismes, ils permettent de garantir le maintien de la cohérence du système.

De manière générale, ces mécanismes constituent un défi actuel important car ils possèdent encore un coût d'intégration élevé et induisent des baisses de performance significatives dues, notamment, à la forte charge du réseau. Sans compter que celle-ci augmente avec le nombre de ressources de calcul. C'est pourquoi des processeurs décident de ne pas implémenter de protocole de cohérence en matériel comme par exemple pour le processeur MPPA-256 de Kalray. Dans cette architecture, 256 processeurs sont rassemblés en 16 grappes de calcul de 16 coeurs chacun. Au sein d'une grappe de calcul, on retrouve une hiérarchie mémoire classique avec un cache L2 partagé et des caches L1 privés à chaque processeur, on ne trouve pas de mécanisme pour garantir la cohérence entre les caches L1. La bonne exécution de l'application s'appuie donc sur des instructions spécifiques du jeu d'instruction afin de contourner le cache L1 lors d'un accès à une donnée partagée est effectué ce qui demande un effort supplémentaire à la programmation pour tenir compte de cet effet. Cette solution permet d'avoir un nombre important de coeurs tout en gardant une consommation énergétique faible.

1.2.4 Vers un design plus simple : les mémoires scratchpads

L'utilisation de cache pour la hiérarchie mémoire a un impact important sur la consommation. Ainsi, une voie d'optimisation importante a été l'utilisation de mémoires scratchpads plus simples au niveau du design mais plus complexes à utiliser. En effet, comme montré sur la figure 1.7, une mémoire scratchpad correspond à un cache dont la partie de stockage de l'adresse et gestion du contenu sont retirées du cache, il ne reste donc plus que la partie essentielle, le tableau de stockage des données. Cette mémoire utilise un espace d'adressage différent et doit être gérée manuellement par l'utilisateur. Elle a une surface plus petite de 34% et un cout par accès inférieur de 40% par rapport à un cache de même taille [22]. Ces avantages correspondent aux attentes des architectures embarquées qui sont par définition plus contraintes en surface et en consommation d'énergie. Cette complexité de programmation pour la gestion des déplacements de données au sein de la hiérarchie peut créer des architectures très compliquées à utiliser efficacement, surtout dans le cadre d'architectures parallèles. De plus, du fait de la gestion explicite des déplacements de données, il n'y a donc pas d'indétermination sur le temps d'accès à une donnée dans une mémoire scratchpad comme on peut l'avoir avec un cache. En effet, un accès à un cache prend plus ou moins de temps selon si le

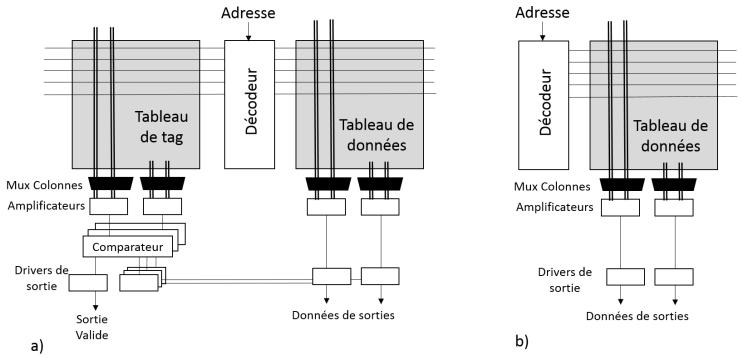


FIGURE 1.7 – Différence de l'organisation entre a) un cache et b) une mémoire scratchapd

cache possède la donnée ou est obligé d'aller la chercher en mémoire principale. Cette propriété de prédictibilité des mémoires scratchpads est importante pour le domaine du temps réel qui peut alors effectuer des calculs plus précis de la latence des pires chemins d'exécutions.

Afin d'aider les développeurs à utiliser de telles mémoires sur des architectures complexes, des modèles de programmation apparaissent afin notamment d'utiliser de telles hiérarchies, ce sont des modèles dits PGAS pour espace d'adressage global partitionné (Partionned Global Address Space) dont les langages X10 [23] ou UPC [24] sont des exemples. Ces langages introduisent des modèles complexes de gestion de la mémoire et de la cohérence loin du modèle de mémoire partagée habituellement utilisé pour les architectures SMP à mémoire partagée. Egalement, on associe généralement chaque mémoire scratchpad avec un DMA. Ce composant matériel permet de faire ces accès mémoire direct à la mémoire sans bloquer le processeur. Le processeur doit alors piloter son DMA afin de copier les données dans la scratchapd pour pouvoir ensuite y accéder.

Dans le cadre des architectures SMP, comme les mémoires scratchpad utilisent un espace d'adressage différent, l'architecture peut autoriser les processeurs à aller chercher les données dans toute mémoire scratchpad présente de l'architecture avec des latences différentes selon qu'ils accèdent à leur mémoire scratchpad local ou à celle d'un autre processeur. L'accès à une mémoire scratchpad distante est appelé accès distant et a une latence comprise entre un accès à la mémoire locale et un accès à la mémoire de niveau inférieure. On parle alors de mémoires scratchpads virtuellement partagée ou d'architecture VS-SPM. Un exemple générique d'architecture VS-SPM est illustré figure 1.8.

Les mémoires scratchpads et les caches possèdent donc une philosophie opposée en matière d'utilisation. Pour le cache, toute la gestion du contenu et de

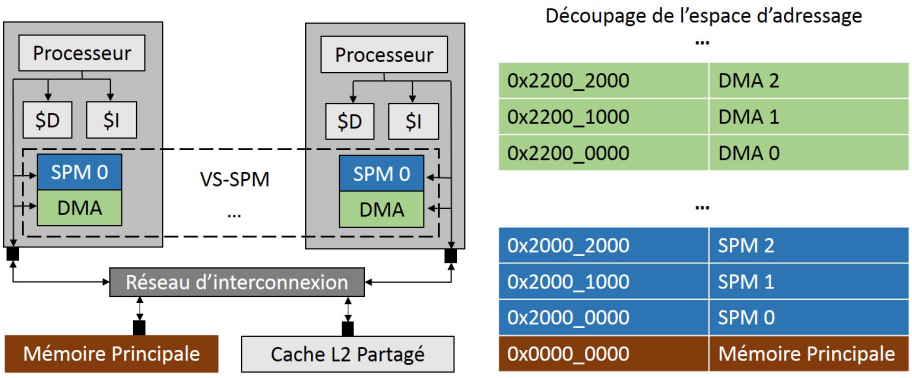


FIGURE 1.8 – Architecture hybride cache scratchpad VS-SPM et exemple du découpage de l'espace mémoire associé

la cohérence est laissée au matériel, ce qui a pour effet de rendre transparent au développeur l'utilisation de telles mémoires mais qui crée dans le même temps potentiellement des déplacements de données inutiles et donc peut conduire à des performances sous-optimales. Les mémoires scratchpads s'appuient sur une gestion purement logicielle et crée donc une complexité importante de programmation de ces mémoires. Pour des architectures SMP, on retrouve les deux systèmes dans les architectures de la dernière décennie. Par exemple, deux consoles de jeu sorties la même année en 2006 utilisèrent chacune une hiérarchie mémoire différente avec le Xenon [25] de la XBox 360 qui utilise une hiérarchie de caches et l'architecture Cell [26] de la Playstation 3 basée sur des mémoires scratchpads. Une étude effectuée par Leverich et al. en 2007 [27] compare les deux modèles de hiérarchie mémoire dans des conditions comparables, elle montre peu de différence par rapport à la performance ou à l'énergie consommée et ne permet pas de conclure de façon définitive quant à la supériorité d'un modèle sur l'autre.

1.2.5 Les architectures NUCA

Une autre problématique, différente des protocoles de cohérence, associée aux architectures à mémoire partagée est le goulot d'étranglement formé par l'accès à la mémoire principale externe à la puce, la latence associée est également très importante. Il s'agit donc de garder un maximum de données sur la puce avant de la réécrire en mémoire principale. C'est pourquoi, la tendance est à l'augmentation de la taille des caches de dernier niveau dans les architectures actuelles, ce qui produit plusieurs problèmes dont des temps d'accès plus important pour ces caches et une consommation statique élevée. Pour la réduction de la latence, un nouveau paradigme a émergé, proposé pour la première fois par Kim et al. [2] en 2002, pour répondre à ce problème avec des designs de cache de dernier niveau, basés sur des temps d'accès non-uniformes (NUCA). La ressource en cache est alors répartie en bancs de caches accessible séparément ce qui autorise des temps d'accès variable

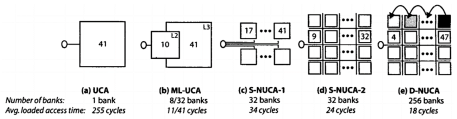


FIGURE 1.9 – Evolution de la latence du cache L2 en fonction de la stratégie de répartition des bancs de cache [2]

selon le banc de cache accédé. Le choix du banc de cache sur lequel on place les blocs de données peut être effectuée statiquement (S-NUCA) ou dynamiquement (D-NUCA). Avec la seconde approche, ce placement peut évoluer au cours de l'exécution pour rapprocher les données les plus accédées par un processeur sur des bancs de cache plus proche du dit-processeur.

1.2.6 Hétérogénéité des technologies mémoire

Comme nous l'avons vu dans la section précédente, la tendance actuelle dans les architectures est à l'augmentation de la taille des caches de derniers niveaux. Cette tendance crée une augmentation de la surface allouée sur la puce pour le cache de dernier niveau, ce qui crée par conséquent une augmentation de sa consommation statique. C'est pourquoi, l'alternative des technologies non-volatile est proposée quant au choix de technologie utilisé pour l'implémentation des caches de derniers niveaux plutôt que la technologie SRAM standard. Ces technologies possèdent une consommation statique quasi-nulle et une densité 4 à 16 fois supérieure à une technologie SRAM classique. Elles introduisent cependant une asymétrie nouvelle entre cout en lecture et écriture, une écriture est en effet très couteuse avec de telles technologies. Des travaux récents [28, 29] suggèrent une efficacité importante des solutions qui utilisent des technologies SRAM et NVM pour un même cache, ce qui permet de tirer le meilleur parti des deux technologies. Cependant, une telle hybridation des technologies suppose d'utiliser des techniques de fabrication des puces siliciums dites d'intégration 3D afin de pouvoir utiliser plusieurs technologies sur une même puce. D'après la communauté Hipeac [30], une communauté européenne importante d'universitaires et d'industriels, cette technologie introduit de nouveaux concepts qui vont amener à repenser la hiérarchie mémoire à tous les niveaux de la hiérarchie et permettre d'adopter dans les prochaines années des modèles d'architecture différents du modèle classiquement utilisé jusqu'ici dit de Von Neumann. Au niveau de l'industrie, ces technologies commencent à arriver à maturation pour être utilisables sur des architectures réelles. Ainsi, Intel et Micron ont annoncé en juillet 2015 une nouvelle technologie baptisée "3D XPoint" qui permet de rajouter une mémoire non-volatile de type ReRAM entre la mémoire principal (DRAM) et la mémoire de masse (NAND).

Après avoir développé les différentes évolutions technologiques des hiérar-

chies mémoires en guise de contextualisation, nous allons développer par la suite quelques éléments permettant de poser la problèmatique adressée par nos travaux.

1.3 Consommation de la hiérarchie mémoire dans les systèmes embarqués

La consommation a toujours été un critère important du design des systèmes embarqués. En effet, l'autonomie est une priorité et il s'agit d'élaborer des solutions matérielles qui conservent des niveaux de consommation suffisamment bas imposés par les capacités d'alimentation des batteries, tout en améliorant les performances de traitement. A ce titre, une convergence semble s'engager entre les architectures du domaine de l'embarqué et les machines à hautes performances. En effet, dans le domaine des supercalculateurs, l'objectif est d'obtenir les meilleures performances possibles et la consommation électrique est également une forte contrainte, puisque les machines sont conçues pour une puissance donnée. Cette dernière est limitée par les systèmes de dissipation de la chaleur autour de la machine. Afin de pouvoir augmenter les performances, il faut donc diminuer la puissance dégagée. Cette inflexion vers la réduction d'énergie pour le calcul haute performance est importante avec l'apparition d'un classement des supercalculateurs depuis 2005, *TheGreen500*, qui vise l'efficacité énergétique des calculs plutôt que la performance pure.

Ainsi que l'on peut l'observer sur les architecture des systèmes embarqués modernes, la hiérarchie mémoire représente une parte importante de la surface et de la consommation de la puce. Quelques chiffres présents dans la littérature montre l'importance de la hiérarchie mémoire dans la consommation de la puce. Pour le ARM 920T, la consommation de la hiérarchie mémoire représente 44% du total de la puissance dissipée par la puce [31]. Pour le Strong ARM SA-110, les caches représentent 27% de la puissance totale, et 60% de la surface [32]. Selon la présentation d'Intel à la conférence MICRO'11 [33], la consommation des caches représente entre 12 et 45% de l'énergie selon les applications étudiées. Ces travaux de thèse se fixent donc pour objectif une réduction de la consommation de la hiérarchie mémoire. Afin d'énoncer la problématique associée, il s'agit d'identifier quelques facteurs de la consommation de la hiérarchie mémoire.

1.3.1 Définitions

Lorsque l'on parle de consommation d'énergie, on distingue habituellement deux grandeurs : puissance et énergie. L'énergie se mesure en Joules alors que la puissance se mesure en Watt, les deux grandeurs sont liées par la formule $E = \int_T P(t)\,\mathrm{d}t$ ou E est l'énergie, P la puissance, et T un intervalle de temps. On utilise l'une ou l'autre des métriques selon le contexte. Cette distinction peut être importante car une diminution de la puissance ne signifie pas nécessairement

une diminution de l'énergie consommée. Cette remarque est valable, par exemple, pour les techniques de modifications de fréquence de fonctionnement (DVFS) qui diminuent la puissance mais augmente le temps d'exécution. Pour les systèmes embarqués, il est souvent plus pertinent de considérer l'énergie consommée que la puissance car le système est alors limité par sa source d'alimentation comme les batteries des téléphones portables.

Avec des technologies CMOS standards, on distingue habituellement la consommation d'énergie statique de la consommation d'énergie dynamique. La consommation statique est une énergie dissipée par les courants de fuites des transistors et dépend des caractéristiques technologiques des transistors. Donc, plus une mémoire est de taille importante (plus elle utilise de transistors), plus elle a une consommation statique élevée. Les façons de réduire cette consommation peuvent être de réduire la taille de la mémoire ou un changement de technologie pour implémenter les cellules de mémoire.

La consommation dynamique est due à l'apparition de courants lorsque qu'un niveau logique change, ce courant sert à charger les capacités de charge des portes logiques. Cette consommation dépend donc du nombre de commutations et donc majoritairement du nombre d'accès à la mémoire. La principale façon de réduire la consommation dynamique d'un cache est donc de réduire le nombre de défauts de caches d'un cache proche processeur pour ainsi accéder moins souvent aux caches plus éloignées. Réduire le nombre de défauts de cache permet de réduire la consommation dynamique et également le nombre total de cycles nécessaires à exécuter l'application ce qui conduit également à une réduction de la consommation statique.

Historiquement, la consommation dynamique représentait la grande majorité de la consommation d'une mémoire. Mais avec des techniques de gravures plus fines qui permettent d'avoir un nombre important de transistors sur la puce, et des mémoires de plus en plus volumineuses, la consommation statique de ces mémoires devient importante. Les études actuelles [3] montrent que la tendance est en train de s'inverser entre les 2 types de consommation. L'évolution de cette proportion de consommation est affichée figure 1.10 entre les technologies 90nm et 32nm. Cela ouvre la voie à des optimisations propres à la consommation statique.

1.3.2 Le principe de localité des données

En pratique, un cache ne peut pas stocker toutes les données de l'application et doit donc faire un choix sur les données à conserver. Idéalement, un cache importe la donnée dont le processeur a besoin et la conserve dans toute sa durée d'utilisation. Pour guider sa décision, on peut observer que les accès aux programmes ne sont pas complètement aléatoires mais peut suivre des comportements prédictibles regroupés sous l'appellation de principe de localité. Ce principe conjecture qu'un

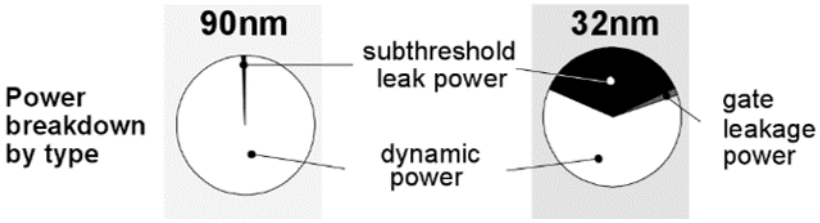


FIGURE 1.10 – Répartition de la consommation en fonction des technologies de gravure [3] sur un cache 64kB 4 voies

programme effectue 90% des accès sur 10% des données. Vérifié par nos expériences sur la suite d'application Mibench[34], on observe que 10% des données les plus réutilisées représentent 93.1% des accès mémoires en moyenne. Ce principe est très important pour un cache qui devrait stocker justement ces données fortement réutilisées aux dépends d'autres données. On distingue plusieurs types de localité :

- La localité temporelle : cette propriété traduit la proximité temporelle entre deux accès à une même donnée ou la tendance d'une application à réutiliser la même donnée au cours de son exécution
- La localité spatiale : cette propriété désigne la tendance à rassembler dans l'espace d'adressable, les données utilisées de la même façon. Ainsi, l'élément *i+1* du tableau est adjacent dans l'espace d'adressage à l'élément *i* et a de fortes chances d'être accédé juste après.
- La localité algorithmique : cette propriété survient lorsque le programme accède régulièrement aux mêmes données mais qui sont distribuées dans l'espace d'adressage. Les applications qui effectuent des calculs répétitifs sur des structures allouées dynamiquement présentent souvent ce type de localité. Ce principe est surtout utilisé pour les algorithmes de préchargement des caches.

Les caches optimisent en se basant sur la localité spatiale et temporelle des données, la localité spatiale parce qu'ils manipulent des données à la granularité du bloc de cache, ce qui signifie que lorsqu'une donnée est importée dans un cache, les autres données situées sur dans le même bloc de cache sont alors importées également. Pour exploiter la localité temporelle, on utilise une politique de remplacement qui s'appuie justement sur ce principe de localité temporelle, la politique LRU (*Least Recently Used*). Une politique de remplacement a la responsabilité de déterminer le bloc à évincer en cas de chargement d'un nouveau bloc de cache et la politique LRU privilégie dans ce cas, de conserver les données qui viennent juste d'être utilisées au dépend d'autres données. Cette politique montre d'excellents résultats justement parce que la plupart des applications présentent une localité temporelle importante. Ce principe de localité est donc très important pour l'utilisation des caches car il permet d'améliorer l'efficacité des caches et donc de réduire la consommation globale de la hiérarchie mémoire. C'est pourquoi on peut trouver une littérature importante sur les solutions qui renforcent ce principe

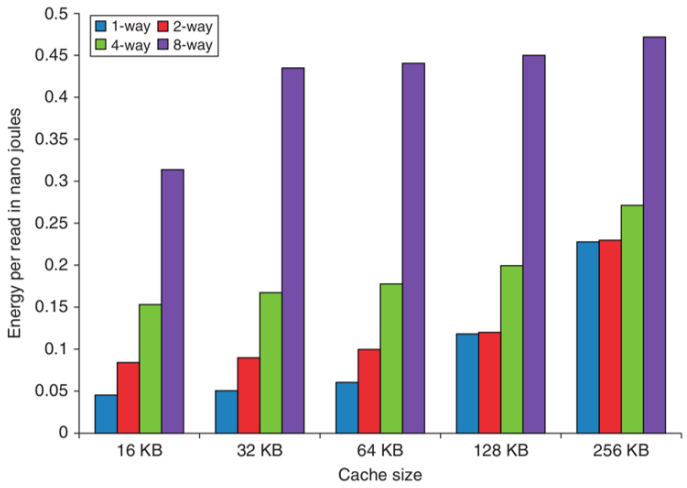


FIGURE 1.11 – Evolution du cout par accès en lecture selon la taille et l'associativité du cache pour une technologie de 40nm[1] selon CACTI

de localité et permettent ainsi d'améliorer l'utilisation des caches. Ce principe de localité est moins respecté sur des niveaux de cache plus éloignés car les accès aux données très réutilisées sont alors filtrés par les caches proches processeurs.

1.3.3 Complexité du design de la hiérarchie mémoire

Le design des caches a une grande importance dans la consommation dynamique. Les paramètres de design d'un cache dont les plus importants sont la taille, l'associativité et la taille de bloc fixent la consommation dynamique par accès. La Figure 1.11 affiche la variation du cout par accès selon l'associativité et la taille de cache pour une technologie de 40nm avec une taille de bloc fixée à 64 octets. On observe que le cout par accès augmente avec la taille et l'associativité du cache, du fait d'un design plus complexe. La réduction de la consommation dynamique peut donc également passer par un cache plus petit ou par une associativité plus faible afin de réduire le cout par accès. Réduire ces paramètres diminue le cout par accès mais entraine automatiquement une augmentation du nombre de défaut de cache. Dans ce cas, le nombre d'accès à la mémoire inférieure augmente et donc la consommation globale de la hiérarchie mémoire augmente dans le même temps. Pour chaque cache, il existe donc un point d'équilibre à trouver entre réduction du nombre de défauts et augmentation du cout par accès pour optimiser la consommation.

Pour montrer ce point d'équilibre, l'exemple d'un algorithme de détection de contour d'une image basée sur le détecteur de Canny sera utilisé. On cherche à designer une hiérarchie mémoire composée d'un seule cache. On se concentre sur la taille du cache, l'associativité étant fixée à 2 et la taille de bloc à 64 octets. On utilise un simulateur de cache pour obtenir le nombre de défaut de ce cache selon

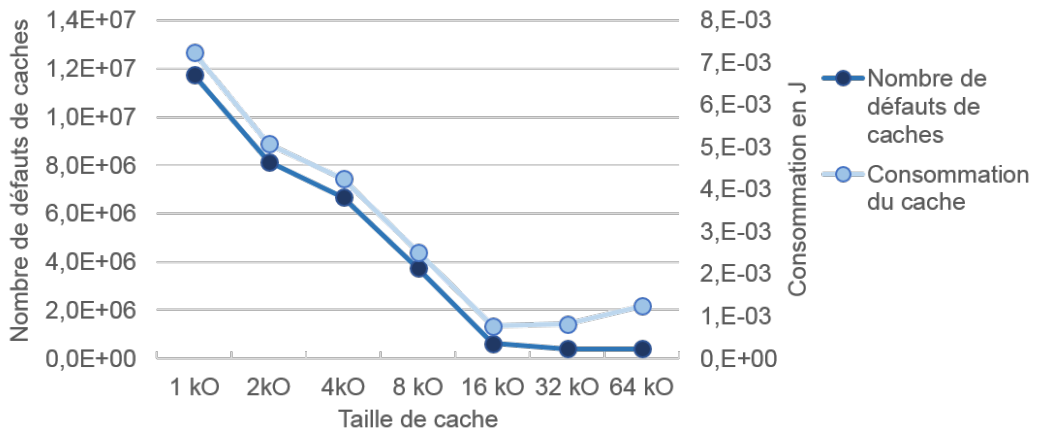


FIGURE 1.12 – Variation du nombre de défauts de cache et de la consommation de *deriche* en fonction de la taille du cache

la taille et CACTI [35] pour obtenir la consommation du cache pour chaque design. On teste donc par simulation toutes les tailles de cache possibles sur un intervalle suffisamment grand jusqu'à tomber sur un optimum. La Figure 1.12 montre les résultats en terme de consommation et de nombre de défauts. Elle montre un design optimal pour la consommation, pour un cache de 16Ko. Il est important de noter que cette solution optimise la consommation et non pas le nombre de défauts de cache.

Cet exemple permet de rendre compte de plusieurs phénomènes. Tout d'abord, cela montre la complexité des choix du design des caches. En effet, dans l'exemple, uniquement la taille du cache varie parmi les 3 principaux paramètres de design évoqués. A titre d'illustration, la hiérarchie mémoire de cache employée dans l'architecture Nehalem d'Intel compte plus d'une dizaine de caches. Il est important de noter que ces paramètres sont généralement interdépendants et faire varier tous ces paramètres simultanément rend la tâche du design de la hiérarchie mémoire complexe. De plus, chaque application utilise le cache de façon différente et il s'agit de trouver un design suffisamment générique qui puisse convenir à un maximum d'applications.

1.4 Problématique

Notre objectif est de réduire la consommation de la hiérarchie mémoire des architectures SMP à mémoire partagée. Nous avons relevé plusieurs paramètres importants dans la consommation dynamique des hiérarchies mémoires actuelles : le design des caches, la question de la localité des données et la gestion de la cohérence. C'est pourquoi les travaux de la thèse proposent de réduire la consommation de la hiérarchie mémoire à travers la gestion spécifique d'un type de données peu

étudié dans l'état de l'art : des données en lecture seule. Nos motivations sont, d'une part, que cette gestion spécifique peut permettre une amélioration de la localité des données du fait d'une séparation du flux mémoire. D'autre part, cela permet d'obtenir des degrés de libertés supplémentaires quant au design des caches de la hiérarchie. Enfin, les données en lecture seule présente l'avantage important de ne pas présenter de contraintes de cohérence et une gestion spécifique peut permettre de réduire le surcout lié à la cohérence dans des protocoles de cohérence de l'état de l'art. Dans le cadre des travaux de la thèse, nous cherchons des solutions transparentes n'entrainant pas de complexités supplémentaires pour l'utilisateur.

1.5 Contributions de la thèse

Les principales contributions de cette thèse sont :

— Une analyse quantitative de l'utilisation des données en lecture seule sur des applications utilisées dans le domaine de l'embarqué et leurs impact en terme de localité. Cette analyse montre que ces données possèdent un comportement bien distinct du reste des données et engendrent une pollution sur les caches proches processeurs.

— Une chaine de compilation qui permet une détection des données en lecture seule sur un chemin de données particulier de façon automatique. L'évaluation de cette chaine montre un taux de détection des données en lecture suffisant pour rendre possible une optimisation architecturale transparente pour l'utilisateur.

— Une exploration architecturale sur la gestion des données en lecture seule au sein d'une hiérarchie mémoire sur architecture multi-cœurs. Cette exploration montre qu'une solution appropriée pour les données en lecture seule permet d'obtenir des gains en consommation significatif tout en conservant des performances stables. Plusieurs optimisations sont également proposées spécifiquement pour les données en lecture seule

1.6 Plan du manuscrit

Le chapitre 2 décrit l'état de l'art et place la thèse par rapport à l'existant. Les thèmes abordés sont l'optimisation de la localité dans les mémoires caches, l'utilisation des mémoires scratchpads, et les protocoles de cohérence. Le chapitre 3 étudie l'utilisation des données en lecture seule sur des applications et l'impact de ces données en termes de localité mémoire. Le chapitre 4 étudie et évalue une chaine de compilation permettant la détection des données en lecture seule et le placement de ces données sur un chemin de données. Le chapitre 5 étudie le placement des données en lecture seule sur des architectures multi-coeurs. Le chapitre 6 étudie diverses optimisations mémoire propre à la gestion de données en lecture seule. Enfin le chapitre 7 conclut et ouvre les perspectives.

Chapitre 2

Etat de l'art

Sommaire

"L'homme connaît le monde dans la mesure où il se connaît : sa profondeur se dévoile à lui dans la mesure où il s'étonne lui-même de sa propre complexité."

Friedrich Nietzsche

Notre objectif est de réduire la consommation dynamique de la hiérarchie mémoire des architectures SMP à mémoire partagée. Nous proposons dans nos travaux une gestion spécifique des données en lecture seule pour des raisons d'optimisation de la localité des données, de simplification que cela peut apporter pour la gestion de la cohérence et des degrés de liberté que cela ouvre sur le design des caches. Afin de placer notre solution dans l'espace des solutions déjà proposées, l'état de l'art sera donc découpé en trois sections. Dans la section 2.1, nous développons différentes techniques d'amélioration de la localité des données proposées au niveau compilateur et au niveau matériel. La section 2.2 fera un état de l'art des protocoles de cohérence d'un point de vue de la classification des données en lecture seule.

2.1 Optimisation de la localité des données

Comme nous l'avons expliqué dans l'introduction, le cache repose sur le principe de localité pour prendre des décisions sur les données à conserver. De nombreuses optimisations ont été proposées pour renforcer ce principe et permettre ainsi une amélioration de l'efficacité de la hiérarchie mémoire. Nous étudions ces optimisations principalement au niveau compilateur et au niveau matériel.

2.1.1 Au niveau compilateur

Cette sous-section montre différentes techniques au niveau compilateur pour améliorer la localité des données. Historiquement, la question de l'optimisation de la localité des données au niveau compilation a largement été développée dans les années 60 pour réduire le nombre de défauts de pages, la communication entre la mémoire de masse et la mémoire de travail constituaient alors un goulot d'étranglement pour les applications. On trouve beaucoup de littérature sur le problème du placement des données sur ces pages, pour placer sur la même page des données utilisées en même temps. Certains de ces principes utilisés ont été transposés à l'optimisation de la localité des données pour les caches pour minimiser les communications entre cache et mémoire principale.

2.1.1.1 Optimisation de boucles

L'optimisation de la localité pour les nids de boucles se pose sous la forme d'un problème d'ordonnancement des différentes itérations. L'idée est donc de rapprocher dans le temps, les itérations accédant aux mêmes données ou aux données proches. Ce problème est généralement décomposé en trois principales étapes : l'analyse des dépendances, l'algorithme de transformation de la boucle proprement dit et la génération de code. Un framework complet permet donc de couvrir ces 3 étapes. Parmi les transformations utilisées dans ces frameworks. On trouve dans la littérature plusieurs frameworks [36, 37, 38, 39] de transformation pour des optimisations de boucles que ce soit pour la localité ou la parallélisation automatique, les deux domaines partageant les mêmes bases théoriques. Ces modèles considèrent le plus souvent une classe particulière de boucles dans laquelle les bornes et les fonctions d'accès aux tableaux sont des fonctions affines des itérateurs de boucles englobantes et des paramètres du nid de boucles. Cette classe de boucles est appelé Scop (*Static Control Program* : programme à contrôle statique) et recouvre un nombre important de cas dans le domaine du traitement d'image, ou du calcul algébrique dense. Il peut arriver que le code ne rentre pas directement dans cette classe de boucles, et plusieurs méthodes permettent alors de s'y rapporter pour pouvoir appliquer ces analyses.

Plusieurs frameworks comme ClooG ou Pluto utilisent le modèle polyèdrique pour décrire les boucles et les transformations. Ce modèle, introduit par Cousot et al.[40] puis utilisé pour la compilation par Feautrier et al. [36], capture les régularités des boucles à l'aide de fonctions affines ce qui permet de décrire pour chaque instruction un espace d'itération ayant la forme d'un polyèdre. Etant donné une instruction S_i à une profondeur p, le vecteur $\vec{x}$ composé de tous les itérateurs de boucles englobant S_i est noté vecteur d'itération de S_i. L'ensemble des valeurs possibles de $\vec{x}$ représente le domaine d'itération de S_i noté $D_{S_i}(\vec{n}) \subseteq \mathbb{Z}^p$, ou $\vec{n} \in \mathbb{Z}^q$ représente les q paramètres du programme. Etant donné que les bornes de boucles

```
for (i=0;i<=N;i++) {
  for (j=0;j<N-i;j++)
    S(i,j);
  for (j=N-i+1;j<=N;j++)
    T(i,j);
}
```

$D_S(N) = \{i, j \mid 0 \leq i < N \wedge 0 \leq j < N - i\}$
$D_T(N) = \{i, j \mid 0 < i \leq N \wedge N - i < j \leq N\}$

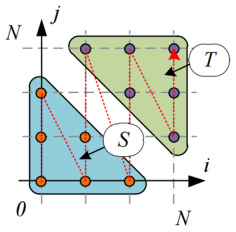


FIGURE 2.1 – Exemple de représentation d'un nid de boucle dans le modèle polyédrique

sont des expressions affines, on peut alors représenter l'espace d'itération de chaque instruction S_i comme une union de d polyèdres, noté alors :

$$D_{S_i}(\vec{n}) = \bigcup_{j=1}^{d} D^j_{S_i}(\vec{n}) \text{ , avec } D^j_{S_i}(\vec{n}) = \{\vec{x} \mid A^j_i.\vec{x} + B^j_i.\vec{n} + \vec{c}^j_i \geq 0\}$$

ou A et B sont des matrices constantes et $\vec{c}$ un vecteur constant. Un exemple d'application de ce modèle est illustré avec la figure 2.1. Dans ce cas, les itérateurs de boucles englobant l'instruction S sont i et j, et le seul paramètre du programme est N. L'ensemble des valeurs prises par ces itérateurs est directement défini par les bornes des boucles. En respectant le formalisme de l'équation précédente, l'ensemble $D_S(N)$ se décrit donc de la façon suivante :

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} (N) + \begin{pmatrix} 0 \\ -1 \\ 0 \\ -1 \end{pmatrix} \geq 0$$

Le domaine d'itération de l'instruction T peut être exprimé de façon similaire. Un des avantages d'un tel modèle est alors que la taille la représentation devient indépendante du nombre d'itérations de la boucle, on peut donc représenter des ensembles d'itérations potentiellement très grands. Une fois la boucle modélisée, des opérateurs peuvent être appliqués sur ce modèle pour modifier l'ordonnancement des itérations et obtenir notamment une meilleure localité des données. Parmi le set de transformations disponibles, on trouve d'abord les transformations de boucle affine ou unimodulaire. Elles s'expriment de façon générique sous la forme d'une matrice de transformation entre deux espaces d'itération telle que les indices de boucles $(i_0, i_1, ..., i_N) = T(i'_0, i'_1, ..., i'_N)$ avec T la matrice de la transformation. La transformation est considérée comme légale, les vecteurs de dépendances de la boucle résultat restent positifs.

Plusieurs transformations classiques permettent alors d'être exprimées de cette façon comme la modification de l'imbrication des boucles (*loop interchange*) ou la modification de la forme de l'espace d'itération (*loop skewing*). La figure 2.2 montre un exemple simple ou la transformation de boucle inverse les boucles intérieures et extérieure ce qui permet d'obtenir un motif d'accès régulier au tableau A (si le tableau est stocké en ligne dans la mémoire). La transformation de ce cas permet

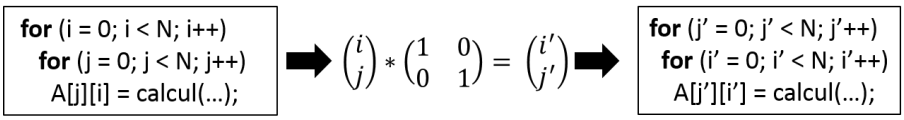


FIGURE 2.2 – Illustration de l'algorithme de permutation de boucle

donc d'améliorer la localité spatiale de l'application.

Ces transformations peuvent alors être composées entre elles facilement pour une optimisation de la localité. Les transformations non affines ne peuvent être exprimée de cette façon, et la plus importante d'entre elle étant le tuilage. Le blocage (*blocking*) ou tuilage (*tiling*) qui partitionne l'espace d'itération afin de décomposer le calcul et de travailler sur des blocs plus petits appelé tuiles qui rentrent dans un cache ce qui augmente la localité des données et donc le taux de réutilisation dans les caches. Les principaux paramètres de tuilage à déterminer sont la taille de la tuile car il faut que les données manipulées par la tuile rentrent complètement dans le cache pour permettre sa réutilisation. On trouve dans la littérature des algorithmes [41, 42, 43] permettant de déterminer ces paramètres de façon optimale pour la localité des données. Ces algorithmes s'appuient sur une modélisation du comportement du cache et selon la précision du modèle adopté, les formes de tuiles et donc les résultats proposés diffèrent. Ainsi, la solution proposée par Lam et al. [41] modèlise sur l'algorithme de multiplication de matrice, les défauts de caches créés par l'associativité limitée du cache (les *conflicts misses*) mais ne considèrent pour la forme de tuile que des tuiles carrées. Esseghir et al. [44] considèrent un plus grand set d'applications, mais n'autorisent que des copies de colonnes entières ce qui conduit à une partie non utilisée du cache. Coleman et al. [43] utilisent l'algorithme d'Euclide pour décider de la taille de tuiles rectangulaire. Du rembourrage de tableau (*padding*) peut également être proposé dans le cadre de cette optimisation pour modifier l'alignement des tableaux en mémoire et ainsi réduire les conflits dans le cache [45, 46].

Les exemples précédents considéraient des boucles dans laquelle il n'y a que la boucle la plus profonde qui exécute des instructions, on parle de nids de boucles parfaitement imbriquées. Des solutions effectuent des transformations sur d'autres classes de boucles afin de pouvoir se ramener à cette situation et ainsi pouvoir appliquer les mêmes optimisations. Ahmed et al. [47] a été un des premiers à proposer un framework pour autoriser ces optimisations à des algorithmes itératifs tel que *jacobi*. Song et al. [48] ont étudié plus précisément le tuilage sur cette même catégorie d'applications. Le tuilage peut également être effectué sur plusieurs niveaux de cache simultanément, chaque cache stockant une tuile correspondant à sa taille. Rivera et al. [49] explorent l'impact d'un tuilage sur plusieurs niveaux et concluent qu'un tuilage sur le cache du premier niveau donne la majorité des gains associés à cette technique.

De manière générale, ces techniques sont maintenant bien maitrisées et les travaux les plus récents sur le sujet s'emploient à maximiser le niveau de parallèlisme de calcul pour des architecture type GPU, à l'aide de formes de tuiles originales comme le tuilage en diamant [50] ou le tuilage hexagonal [51]

2.1.1.2 En multicoeurs

Toutes les optimisations proposées jusqu'à présent considèrent une architecture mono-cœur. Les caches partagées que l'on trouve classiquement dans les architectures parallèles, introduisent une difficulté supplémentaire en termes de localité des données, car du fait que plusieurs processeurs accèdent au même cache, des interférences constructives ou destructives peuvent se former. On parle d'interférences constructives lorsque l'interaction entre deux processeurs sur le cache permet d'augmenter la réutilisation d'un bloc de donnée dans le cache, on parle d'interférences destructives dans le cas contraire. Ainsi, Bao et al. [52] étudient sur les architectures multi-cœurs, l'impact des interférences négatives sur les algorithmes de tuilage et montrent la dégradation importante apportée, lorsque l'algorithme est utilisé sur un cache partagé avec une autre application. Ils utilisent une analyse complexe à la compilation pour déterminer l'impact de ces interférences négatives et ainsi diminuer la taille de la tuile en fonction. Leur solution permet alors d'enlever la majorité toutes les interférences négatives.

Un des premiers travaux qui exploite les interférences constructives est l'étude de Zhang et al. [53]. Ils considèrent un algorithme de tuilage similaire à celui vu précédemment, ou l'ordonnancement des itérations est décrit sous la forme d'une table ou chaque ligne représente un slot de temps de calcul sur une tuile et les colonnes les différents processeurs. Le problème est donc de placer les tuiles dans le temps (slot de temps) et l'espace (les processeurs) là où les algorithmes précédents ne considéraient que la dimension temporelle. Lorsqu'un slot est disponible, l'affinité de chaque tuile candidate à l'acquisition de ce slot est calculé à partir des tuiles déjà placées sur les processeurs ou des tuiles qui ont exécuté les slots de temps précédents. Le meilleur candidat est ainsi sélectionné. Cette technique permet de conserver les gains proposés par les optimisations en mono-cœur sur les caches privés tout en améliorant la réutilisation des données sur les caches partagés. La réduction des taux de défauts de cache des niveaux L2 et L3 est de 22% en moyenne. Cette solution considère une hiérarchie à plusieurs niveaux ou les caches peuvent être plus ou moins partagés.

Après avoir étudié quelques optimisations importantes de la localité au niveau compilateur, nous nous intéresserons aux optimisations proposées au niveau matériel.

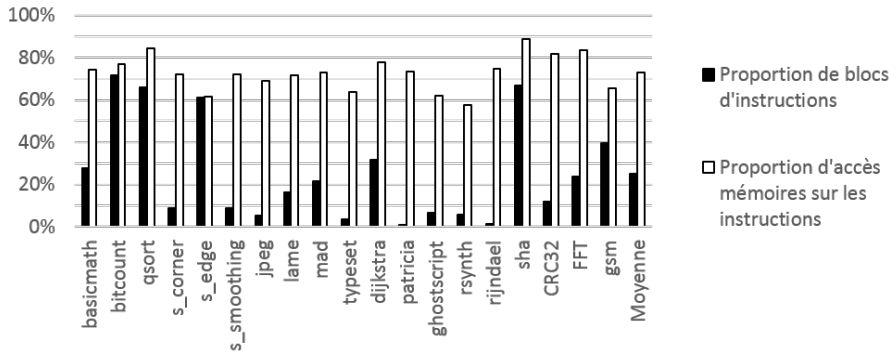


FIGURE 2.3 – Répartition des accès mémoires et des blocs

2.1.2 Au niveau matériel

Au niveau matériel, la technique principale d'optimisation de la localité des données est la partition du cache en plusieurs sous-structures selon une classification des données. En effet, le principe de localité énonce que 10% des données représentent 90% des accès mémoires effectués mais cette répartition n'est pas homogène et les données ont des comportements bien distinct selon le type de données considéré. L'idée est donc d'isoler ces différents comportements pour les rassembler des sous-groupes de données avec des caractéristiques bien distinctes en terme de localité pour pouvoir ainsi mieux les exploiter. La principale difficulté d'une telle gestion est la classification des données au niveau matériel afin de permettre de diriger les requêtes mémoires vers la structure mémoire approriée.

Une technique largement utilisée depuis les années 1960 est la séparation du premier niveau de la hiérarchie entre cache d'instruction et cache de données. Cette séparation a de multiples motivations parmi lesquelles une optimisation de la localité des données. Cette optimisation est communément admise et peu de littérature qualifient cette optimisation en termes de localité. C'est pourquoi nous avons réalisée quelques expériences supplémentaires. Ainsi, une analyse du jeu d'applications Mibench [34] illustrée Figure 2.3 montre que les instructions représentent une fraction faible de l'ensemble des données utilisées, 25.2% en moyenne, mais une fraction importante des accès mémoires 72.8%. On voit donc une asymétrie entre la proportion des instructions sont beaucoup plus réutilisées que les autres données.

Sur quelques applications du jeu Mibench, nous avons comparé un niveau L1 séparé entre instructions et données à un niveau L1 unifié à espace constant. Nous nous plaçons dans un scénario avec La Figure 2.4 affiche les paramètres de design des différents caches et les résultats en termes de taux de défauts de cache et de consommation d'énergie. La séparation du niveau L1 entre cache de données et cache d'instruction apporte un gain important sur la consommation globale du

	Taille du cache	Associativité	Taille de bloc
Cache unifié	32 kO	2 voies	64 octets
Cache de données	16 kO	2 voies	64 octets
Cache d'instructions	16 kO	2 voies	64 octets

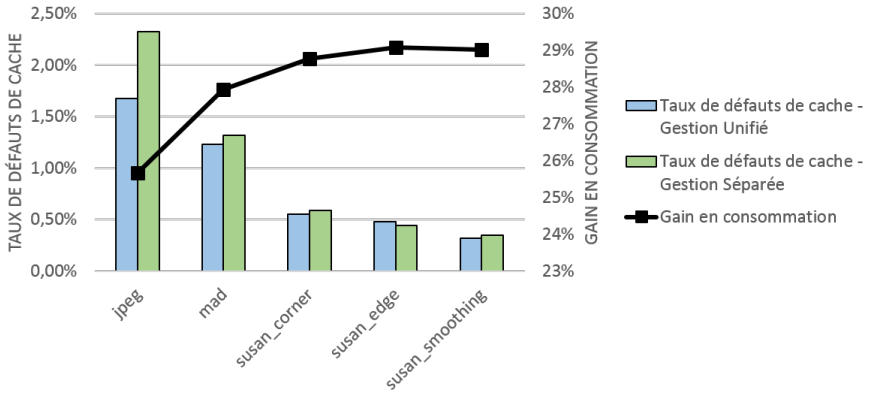


FIGURE 2.4 – Comparaison entre un niveau L1 unifié et séparé entre instructions et données en termes de défauts de cache et de consommation

fait d'un taux de défauts de cache relativement constant malgré le fait que les flux mémoires passent dans des caches deux fois plus petits.

Le fait que les défauts de cache n'augmentent pas ou peu avec la séparation du flux mémoire indique chaque flux possède une localité spatiale et temporelle propre. Les instructions et les données forment deux sous-ensembles exclusifs d'accès mémoires dont la séparation ne peut qu'améliorer la localité des deux sous-ensembles. La question est de déterminer si ces sous-groupes de données possèdent des comportements suffisamment différents pour justifier une gestion spécifique. Pour ce qui de différenciation de gestion entre les instructions et des données, on peut observer que cette séparation est utile. Cependant, avec des ensembles de données non-exclusifs alors cette séparation n'est pas nécessairement bénéfique pour la localité des données.

D'autres exemples de séparation du cache L1 peut être trouvé dans la littérature. Ces techniques divisent le cache de donnée du premier niveau en plusieurs structures indépendantes (*dual data cache* : DDC) et un état de l'art de ces techniques est proposé dans [54]. La première proposition a été faite par Gonzales et al. [55] en 1995. Leurs solutions partent de l'observation que les accès aux données possèdent soit une localité spatiale, soit une localité temporelle. Par exemple, les données scalaires présentent peu de localité spatiale, car deux données scalaires placées à côté dans l'espace mémoire ont peu de raison d'être accédées l'une après l'autre, mais peuvent présenter une localité temporelle importante. A l'inverse, les structures

comme des tableaux accédés avec un pas faible, utilisent plus la localité spatiale que la localité temporelle. Ainsi, le cache doit optimiser selon deux comportements bien distincts. Le cache de donnée est donc divisé en deux : un *spatial cache* optimisant la localité spatiale avec des taille de blocs de cache plus importantes et avec une politique de remplacement FIFO et un *temporal cache* optimisant la localité temporelle avec des tailles de blocs plus petites. Une table de prédiction est ajoutée au système afin de déterminer dans quel cache la donnée doit être importée, l'algorithme utilisé sur cette table fait passer toutes les données d'abord dans le *temporal cache* et si des accès avec un pas similaire sont détectés, les blocs associés sont promus dans le *spatial cache*. Des processeurs tels que le CalmRISC [56] ou le StrongARM SA-1110 [57] implémentent le cache de donnée de cette façon. Cette idée a ensuite été reprise par Adamo et al. [58] qui ajoutent au *spatial cache* un buffer de pré-chargement uniquement sur ce cache. Les techniques de pré-chargement se basent en grande partie sur la localité spatiale et utiliser ces techniques uniquement sur le *spatial cache* permet un pré-chargement plus pertinent.

Dans une autre direction, Lee et al. [4] proposent de découper le cache de données en trois sous-structures pour placer dans chacun d'eux les trois différents types de données : les données globales, du tas et de la pile. Cette technique appelée *Region-Based Caching*, permet de proposer un design approprié aux propriétés de chaque type de données. Ainsi, les données de la pile présentent une localité temporelle très forte, le cache associé peut donc être très petit (2kB), les données du tas ont une faible localité et ont donc un cache plus gros, etc. Ainsi, les trois caches sont associés à un multiplexeur pour que le processeur puisse déterminer quel cache accéder, la figure 2.5 illustre l'architecture du cache de donnée. Cette technique présente l'avantage important d'ouvrir des degrés de liberté quant au design à adopter sur chaque sous-structures et permet donc de réduire le cout par accès de façon très importante par rapport à un cache global et les gains en consommation présentés sont de l'ordre de 60% en moyenne.

Une idée proposée par Geiger et al. [59, 60] propose de spécialiser encore plus le cache gérant les données du tas, car c'est son design est le plus important. Une étude des données du tas montre que l'on retrouve également dans ce cas, le principe de localité avec peu de données qui représentent une majorité des accès. Un petit cache est d'abord utilisé et le cache plus important est activé si nécessaire. Cette proposition permet de diminuer la consommation du cache de données en moyenne de 79% sur les applications évaluées. L'inconvénient majeur de ces méthodes est qu'il suppose que l'espace d'adressage est statique et donc qu'un multiplexage sur l'adresse accédée suffit pour connaitre le type de la donnée. Or, avec les techniques de distribution aléatoires de l'espace d'adressage (ASLR) développées pour des raisons de sécurité des applications, cette hypothèse n'est pas valable. Cependant, au vu des gains proposés, des solutions [61, 62] plus récentes essaient de contourner cette difficulté. Parmi les techniques utilisées, Kang et al. [62] s'appuient un *stack cache* virtuellement adressé qui détermine les accès à la pile à partir du pointeur de pile et du pointeur d'appel. Olson et al. [61] comparent les

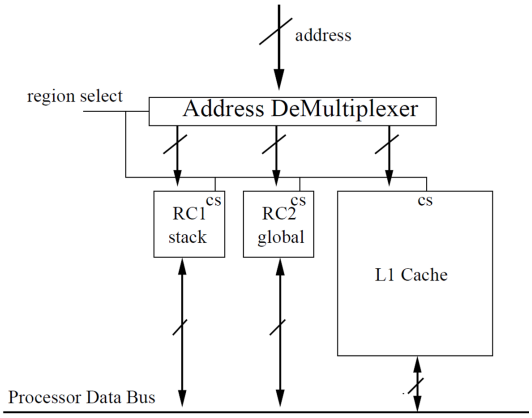


FIGURE 2.5 – Illustration d'un cache de premier niveau découpé selon la technique de *Region-based caching* [4]

bits de poids de l'adresse accédés au pointeur de pile, cette solution permet de détecter les données venant de la pile avec un taux d'erreur inférieure à 1%, les gains en énergie sont alors de 37% sur l'énergie dynamique du cache de données.

2.1.3 Discussion

Nous avons vu dans cette section que des techniques de renforcement de la localité spatiale et temporelle sont importantes dans la diminution du nombre de défauts de cache sur tous les caches de la hiérarchie. Les techniques d'optimisation de localité au niveau compilateur développent des modèles de caches permettent de déterminer précisément les effets sur les caches des optimisations en localité, que ce soit pour des architectures mono-coeur ou multi-coeur. Au niveau matériel, la principale technique d'amélioration de la localité est la séparation du cache de donnée en sous-structure qui permet d'exploiter et de renforcer les propriétés de localité de chaque flux mémoire. On peut observer que cette séparation s'effectue principalement au niveau L1 ou l'on peut observer des comportements de donnée très distincts. Cette technique de réaliser des optimisations propres à chaque type de données considérée. De plus, lors de la séparation du cache de données, les sous-structures considérées restent exclusivement des caches et toute la gestion est faite au niveau matériel. Une partie importante des solutions mises en oeuvre est la façon dont les données sont orientées entre les sous-structures du cache, cette partie se base très souvent sur l'ajout de composants au niveau matériel permettant de faire cette prédiction car cela permet de garder une solution transparente.

2.2 Classification des données dans les protocoles de cohérence

Afin de réduire la consommation liée à l'utilisation des mécanismes de cohérence tels que définie en introduction, une voie d'optimisation importante qui s'est développé ces dernières années proposent d'utiliser ces mécanismes uniquement pour les données partagée et en lecture-écriture. Cette classification fait directement partie des mécanismes VIPS-M mais a été proposée comme optimisation avec les mécanismes d'espionnage [63] ou les mécanisme par répertoire [64]. En effet, une étude menée par Cuesta [5] sur la suite d'applications parallèles standards classifie les blocs mémoires selon ces critères, et les résultats sont présentés Figure 2.6. On observe que 84% des blocs de cache sont soit en privés soit en lecture seule et n'ont donc pas besoin de cohérence. Il y a donc un potentiel important de réduction des effets négatifs liés à la gestion de la cohérence en effectuant un suivi de la cohérence uniquement sur des blocs de cache utiles. Les effets d'une telle classification sont multiples. Dans le cadre d'un mécanisme par répertoire, elle permet un design plus petit du répertoire, elle évite les invalidations des données du à l'invalidation de l'entrée dans le répertoire et permet également d'éviter les indirections causées par le répertoire. Une telle classification des données a également été étudiée pour un meilleur placement des données sur une architecture NUCA [65, 66], les données privées sont alors placées sur les bancs de cache partagés proches des processeurs les utilisant, et les données en lecture seule (dans leurs cas, les instructions) sont autorisées à être dupliquées dans les caches privés sans suivi de cohérence.

Ces solutions peuvent être classifiées selon plusieurs critères. D'abord, selon les solutions, on trouve une granularité au niveau page du système d'exploitation ou au bloc de cache. Une granularité au niveau page implique qu'il suffit qu'un seul bloc de la page soit partagé, pour que toute la page soit considérée comme partagée. Cette granularité peut limiter le nombre de blocs de cache bénéficiant de la solution mais permet de limiter la quantité d'informations à enregistrer. Dans l'étude de Cuesta et al. [5], le passage de granularité de la page au bloc de cache permet de détecter 18% en plus de données non-cohérents en moyenne sur la suite d'applications étudiée. Un autre critère de classement des solutions dans l'état de l'art est le niveau auquel est effectué la détection, elle peut être effectuée soit par le compilateur [65, 67], soit le système d'exploitation [66, 64, 63] ou au niveau matériel [68, 69, 21].

2.2.1 Classification au niveau du système d'exploitation

Au niveau système d'exploitation, cette détection se fait surtout à l'aide du TLB et les informations nécessaires sont encodées dans les pages. Une solution importante a été proposé par Cuesta et al. [64]. Ils proposent une gestion non cohérente des données privées au niveau page de système d'exploitation. Chaque page est

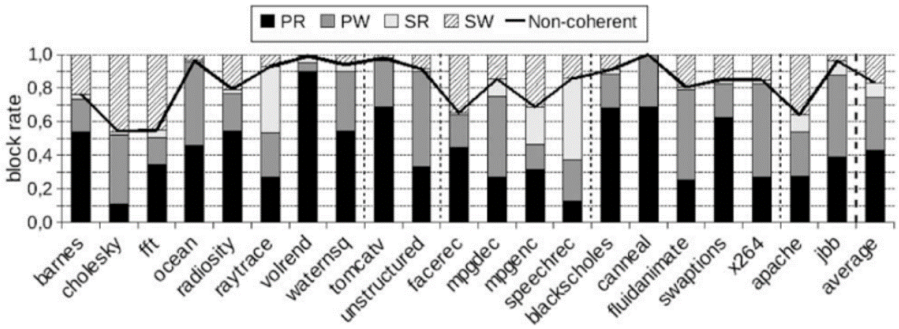


FIGURE 2.6 – Classification des blocs de données PR : bloc privé en lecture seule, PW : bloc privé en lecture-écriture, SR : bloc partagé en lecture seule, SW : bloc partagée en lecture-écriture [5]

considérée comme privée lorsqu'un processeur y fait référence pour la première fois. Les accès mémoires effectués sur cette page sont non-cohérents, c'est-à-dire qu'ils ne créent pas d'entrées dans le répertoire et ne nécessitent pas de contrôles sur les droits d'accès de la part des contrôleurs mémoires (pas plus de contrôles que dans le cas d'un processeur monocoeur). Si un autre processeur tente d'y accéder, la page passe en mode partagé, un mécanisme de rétablissement de la cohérence se met en place, et le protocole de cohérence effectue par la suite les contrôles nécessaires au maintien de la cohérence sur cette page. Les informations nécessaires sont encodées à l'aide de plusieurs bits supplémentaires dans la page. Deux propositions sont effectuées pour le mécanisme de rétablissement de la cohérence. Le premier mécanisme invalide tous les blocs de données appartenant à la page en question dans le cache privé du processeur utilisant la donnée. Une fois les bonnes versions des données récupérées dans le cache partagé, une entrée dans le répertoire est allouée pour les blocs de cache de la page. Le second, au lieu d'invalider les blocs du cache privé ce qui augmente le taux de défauts de caches, préfère ajouter dans l'entrée du TLB, l'information des blocs de la page utilisés par le processeur via un tableau de bits. Ce tableau de bits est envoyé au répertoire qui est alors capable avec l'adresse de la page de retrouver l'adresse de tous les blocs présents dans le cache privé du processeur et crée une entrée de répertoire pour chacun d'eux, il n'y a aucun déplacement de données dans ce cas mais plus d'informations sont alors nécessaires. Testées sur une architecture à 12 cœurs, la solution permet à 59% des blocs de caches d'éviter d'être suivies par la gestion de la cohérence, et la taille du répertoire est réduite par 16. La réduction de la consommation est alors de 40%. Les résultats montrent peu de différence entre les deux mécanismes de rétablissement de la cohérence. Cette solution a ensuite été étendue [5] en 2013 de façon similaire pour les pages partagées en lecture seule ce qui permet de passer à 66% les blocs non concernés par la cohérence. Parmi les autres travaux rentrant de cette catégorie de solutions, Kim et al. [63] utilisent une technique similaire dans le cadre des protocoles de cohérence d'espionnage. L'information du nombre de

processeurs partageant le bloc de cache est stockée dans la page ce qui permet de n'envoyer les messages de contrôles de cohérence qu'aux processeurs utiles. Hardavellas et al.[66] utilisent cette technique avec un cache L2 NUCA pour placer les données privées sur les bancs de caches plus proches du processeur les utilisant permettant de faire gagner 14% de performance en moyenne.

Une classification des données au niveau système d'exploitation permet donc notamment de réduire l'impact du protocole de cohérence de façon importante en termes de consommation et de performance. Cependant, ce type de solution est réactif c'est-à-dire que les pages sont définies comme privées au début de l'exécution puis vont changer de statut selon l'apparition d'accès partagés ou d'écriture sur ces pages. Il n'y pas de possibilités de transition inverse. C'est pourquoi, d'autres solutions ont été proposées au niveau compilation pour permettre des changements dynamiques de cette propriété.

2.2.2 Classification à la compilation

On trouve peu de classification des données à la compilation pour résoudre ce problème en particulier, essentiellement les travaux de Li et al. détaillés dans [65, 67]. Afin de résoudre les différentes indéterminations inhérentes à la compilation, l'analyse statique développée par Li et al. classifie les données en trois catégories : privées, pratiquement privées, ou partagées. Des conjectures sont effectuées sur le comportement des données privées, qui permettent d'identifier les données privés sans pour autant pouvoir le prouver formellement. Ces données rentrent alors dans la catégorie de pratiquement privée. Il peut s'agir par exemple de tuiles d'un tableau répartis qui sont réparties entre les différents threads et qui effectuent des accès partagés uniquement sur les bordures. Les tuiles sont alors considérées comme pratiquement privées. De la même façon que les solutions basées sur le système d'exploitation, l'information est encodée dans les pages et un mécanisme de rétablissement de la cohérence est alors prévu pour les accès partagés aux données pratiquement partagées. Cette solution détecte principalement des données pratiquement privées et permet d'éviter à 65% des données d'être traquées par la gestion de la cohérence.

Les solutions montrées jusqu'à présent utilisent une classification avec une granularité au niveau page ce qui effectue une approximation qui peut limiter les gains apportés car un seul accès partagé suffit à modifier le statut de toute la page. Pour lutter contre ce phénomène de "faux partage", des solutions ont été proposées. Par exemple, pour la solution de Li et al. [67], la fonction *malloc* est surchargée afin de placer les données de même type sur la même page. Une amélioration de la solution de Cuesta et al. [5] a été proposée récemment par Ros et al. [70], elle utilise une classification hydride entre le système d'exploitation et le compilateur.

2.2.3 Classification au niveau matériel

Au niveau matériel, Pugsley et al. [69] envisagent une classification adaptative, c'est-à-dire que les données peuvent changer plusieurs fois de classe pour au cours de l'exécution de l'application. L'idée de leur solution de base est d'utiliser un protocole de cohérence sans répertoire qui maintient les données partagées en lecture-écriture dans le cache partagé, l'accès à ces données est donc plus élevé. Cette solution est proposée avec plusieurs améliorations dont un compteur de saturation à 2 bits à chaque bloc de cache L2 qui permet, si le bloc n'est pas écrit pendant un temps suffisamment long, de retourner à un état privé. Il peut dans ce cas, de nouveau être alloué en L1. Le réglage de ce temps est important pour l'efficacité de la solution et complexe à mettre en œuvre. Les gains obtenus par Pugsley et al. sont peu importants par rapport à un protocole par répertoire clasique, et indiquent que les données partagées doivent pouvoir être accédées via les caches privées pour obtenir une solution satisfaisante. C'est dans la continuité de cette idée, que l'on trouve les protocoles de cohérence VIPS-M[19]. Ces travaux ont été étendues avec une classification hiérarchique pour des systèmes avec des caches partagés avec un sous-ensemble des processeurs [72]. Afin de mesurer l'importance de la granularité et de l'adaptabilité de la solution, une étude est menée par Davari et al.[21]. Sur le jeu d'application SPLASH-2, ils implémentent une optimisation des protocoles VIPS-M qui permet d'effectuer une classification au niveau bloc plutôt qu'au niveau page. De plus, cette classification au niveau bloc est adaptative et permet des changements de statuts. Ainsi, leur taux de détection de données privées et de données en lecture seule est nettement supérieur à un protocole VIPS-M, le taux de détection passe de 33

2.2.4 Discussion

Les protocoles de cohérence limitent le passage à l'échelle des architectures actuelles et introduisent un surcout important en consommation. L'optimisation proposant de classer les données pour n'utiliser cette gestion couteuse qu'aux données qui en ont besoin apporte des gains importants en consommation d'énergie. Cette classification peut donc être effectuée sur les données privées et les données en lecture seule, utiliser une granularité d'une page ou du bloc de cache, et peut également être adaptative. Parmi les solutions de l'état de l'art, peu d'approches s'attachent à une détection à la compilation et ne détecte que les données privées, pas les données en lecture seule. Ils ne considèrent pas non plus une solution adaptative. Une gestion spécifique des données en lecture seule permettra donc d'améliorer les résultats déjà apportés par cette optimisation dans le sens où elle constitue une approche orthogonale aux travaux existants. On peut également observer que tous les travaux détaillés ici utilisent une définition dynamique que ce soit pour les données privée ou pour les données en lecture seule.

2.3 Discussion générale sur l'état de l'art

D'après l'état de l'art développé, on voit que l'information de classification des données dans la hiérarchie mémoire peut permettre une multitude d'optimisations selon la classification effectuée. Ainsi, elle peut permettre dans un cas, d'isoler les localités propres à chaque classe de données et permettre ainsi une séparation du flux mémoire pour des designs de caches plus adaptés, exploitant mieux la localité de chacune des classes de données. Dans d'autres situations, elle peut permettre une réduction importante de la gestion de la cohérence permettant un meilleur partage des données. Cette information de classification peut être dynamique et autoriser des changements de classe au cours de l'exécution. Des erreurs de classification peuvent aussi être autorisés et des mécanismes peuvent peut alors corriger l'erreur, il faut cependant que le taux d'erreur reste suffisamment faibles ou le mécanisme soit peu couteux dans ce cas précis. Pour produire l'information de classification, peu de solutions s'appuient sur le compilateur

D'autre part, peu de solutions font la distinction entre lecture et écriture dans la hiérarchie mémoire, et on ne trouve dans la littérature aucune analyse étudiant directement la propriété de lecture seule des données. Or les solution de gestion spécifique étudiées dans ce chapitre sont pertinentes car la classification des données révèle des comportements bien distincts en terme de localité et de besoin en ressources de cache selon les classes de données. Pour notre cas de la gestion proposée des données en lecture, cette différence est moins intuitive et il s'agira donc d'abord de l'illustrer afin de justifier d'une gestion spécifique de ces données. Le chapitre 3 présente une analyse des données en lecture seule et de leur localité par rapport au reste des données.

Chapitre 3

Etude des données en lecture seule dans des applications usuelles

Sommaire

"Si vous voulez tout optimiser, vous ne serez jamais heureux"

Donald Knuth

Nous avons pu voir dans le chapitre précédent, que séparer le flux mémoire principal en plusieurs sous-ensembles est une stratégie permettant des gains importants en consommation du fait d'un design de cache adapté au type de donnée et qui permet de mieux exploiter les localités propres à chaque type de donnée. Cette approche est intuitive pour des sous-ensembles de données exclusifs comme les données et des instructions qui ont des comportements bien différents, et le fait de les séparer ne peut qu'améliorer la localité de chacun de sous-ensembles. Cependant, cette approche est moins naturelle pour les données en lecture seule. L'étude de l'état de l'art nous permet d'affirmer que la question de la gestion spécifique des données en lecture seule n'a jamais été abordée directement, mais vient plutôt comme extensions à d'autres solutions de gestion spécifique. C'est pourquoi, ce chapitre propose une analyse quantitative et qualitative de l'utilisation des données en lecture seule dans des suites d'applications, également une analyse de la localité des données sera proposée à l'aide de métriques adaptées. Elle nous permettra de justifier l'approche d'une gestion spécifique des données en lecture seule, et de comprendre le comportement singulier de ces données pour proposer une gestion adaptée dans la hiérarchie mémoire.

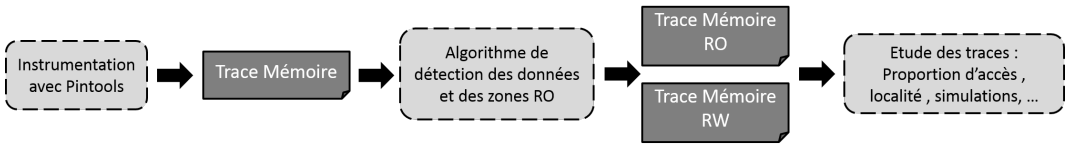


FIGURE 3.1 – Méthodologie employée pour l'étude des données en lecture seule

3.1 Détection des données en lecture seule

Afin de pouvoir étudier les données en lecture seule, il s'agit d'abord d'en donner une définition précise. Une donnée en lecture seule est définie de façon stricte comme étant écrite une seule fois à son initialisation puis uniquement lue pendant le reste de l'exécution du programme. Cette définition nous limite seulement aux constantes et aux données d'entrées de l'application. Afin d'élargir cette définition et d'englober les définitions des données en lecture seule vues dans l'état de l'art, il faut permettre que cette propriété soit dynamique et puisse être appliquée sur des périodes de temps plus courtes que la durée de vie de la donnée. Il s'agit de trouver une période de temps adaptée à l'étude, car on peut à l'extrême, considérer chaque accès isolé en lecture à une donnée comme une période de lecture seule de la donnée. Pour notre étude, nous voulons être capable d'identifier des données étant dans un mode de lecture durant un temps suffisamment long comme un nœud de boucles ou une fonction a minima. Une donnée peut donc changer de statut au cours de l'exécution de l'application et les groupes de données ne sont pas exclusifs comme on peut le trouver pour les solutions de gestion spécifique présentées dans l'état de l'art. Une donnée peut changer de statut au cours de l'exécution et il s'agira de gérer au mieux ces changements.

Certaines données sont déjà identifiées comme étant en lecture seule et ne seront pas considérées dans notre étude :

— Les instructions. Nous avons eu l'occasion de montrer dans le chapitre 2, l'intérêt d'une séparation des instructions et des données au premier niveau de la hiérarchie mémoire. Il ne s'agit pas de remettre en question cette optimisation mais plutôt d'affiner la séparation des données déjà présentes. Dans la suite de ce chapitre, les instructions seront donc systématiquement retirées de l'analyse sauf si mentionné explicitement.

— On peut trouver dans les applications compilées pour x86 ou pour MIPS, une section du code qui rassemble des données en lecture seule (la section *.rodata* sous x86). Cette section sert généralement à stocker les chaines de caractères stockées en dur de l'application, et représente très peu d'accès mémoire ($< 1\%$ sur la suite SPEC CPU2000 d'après Lee et al. [4]) et ne nous intéressera pas dans la suite de ces travaux.

L'analyse des données en lecture seule utilisée dans ce chapitre s'effectuera

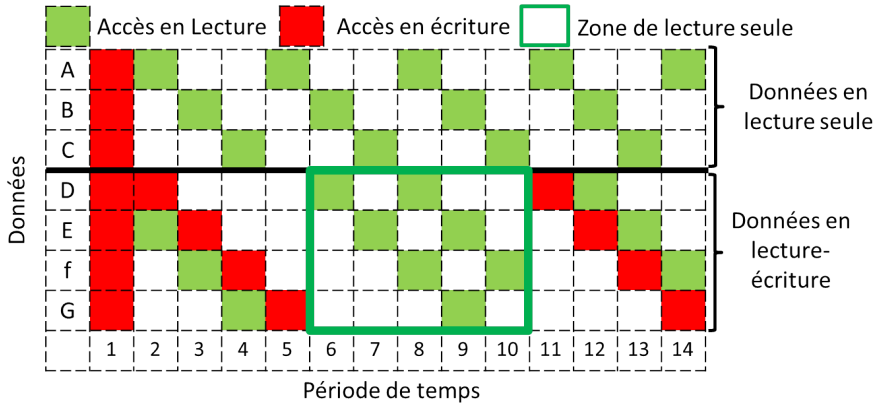


FIGURE 3.2 – Evolution du nombre de transistors dans les processeurs et conséquences

sur des traces mémoires telle que présenté sur la figure 3.1. Tout d'abord, une trace mémoire est générée via une exécution séquentielles de l'application instrumentée avec un outil d'instrumentation dynamique de code développé par Intel, Pintools [73]. Ensuite, cette trace est analysée par un algorithme qui effectue une détection des données en lecture seule correspondant à la définition donnée ci-dessus. Cela permet de générer une séparation de la trace mémoire principale en deux sous-traces mémoire. Un exemple de l'algorithme de détection est présenté figure 3.2. A,B et C correspondent à des données en lecture seule de façon stricte et D,E,F,G des données en lecture-écriture. On peut cependant observer que du slot 6 au slot 10 ces données sont lues et forment ainsi une zone de lecture seule. C'est à l'algorithme de détection de décider si la zone est considérée comme suffisamment importante pour être conservée. Le critère choisi pour garder la zone mémoire est essentiellement basé sur le nombre d'accès que la zone a réussi à agréger qui est d'abord fixé à 1024 accès. Des expériences complémentaires seront menées pour évaluer la variation des résultats sur ce paramètre.

3.1.1 Applications Étudiées

La thèse s'intègre dans un contexte de systèmes embarqués. Afin de réaliser une analyse, il s'agit de se doter d'applications représentatives de ce domaine. Les programmes sélectionnés sont des applications et des noyaux de l'état de l'art actuellement utilisés dans le commerce et l'industrie. Toutes les applications sont compilées avec GCC 4.9 pour x86 et le flag d'optimisation O3. L'analyse sera également plus poussée sur les applications de traitement de signal et d'image. En effet, ce type d'applications est souvent implémenté sous la forme d'un pipeline d'exécution décomposé en tâches, où une tâche effectue un certain calcul avec les données d'entrées et écrit les résultats dans une structure de données. Cette structure est ensuite utilisée (lues) pour la tâche suivante. Pour les applications de traitement d'images, cette structure peut être une image intermédiaire qui sera

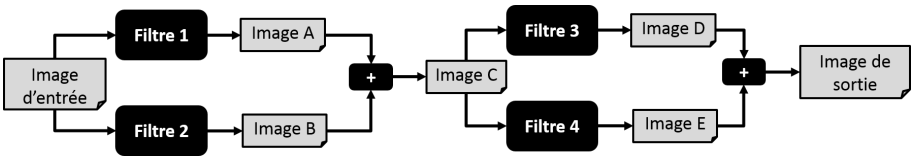


FIGURE 3.3 – Evolution du nombre de transistors dans les processeurs et conséquences

donc écrit intensivement au moment de sa création, puis lue pendant le reste du traitement. A titre d'illustration, la figure 3.3 expose le pipeline d'exécution d'un filtre de Canny qui permet de détecter les contours sur une image. Le pipeline d'exécution se décompose en filtres appliqués successivement, les résultats étant stockés dans des images intermédiaires. Chacune des images intermédiaire à partir du moment où elle est générée, devient ainsi en lecture durant le reste de l'exécution de l'algorithme.

La suite MiBench [34] est mise en œuvre et maintenue par l'université du Michigan et illustre quelques exemples d'applications utilisées dans les systèmes embarqués. Les applications sont divisées en six catégories : l'automatique et le contrôle industriel, le réseau, la sécurité, dispositifs consommateur, les bureautiques et les télécommunications. Toutes les sources des programmes sont accessibles en langage C. Ces applications sont séquentielles et sont fournies avec un petit et un large jeu de donnée. Le petit jeu de donnée permet d'exécuter de façon légère l'application et de valider son fonctionnement tandis que le jeu de donnée large fournit une version plus réaliste de l'exécution de cette application. L'analyse s'effectuera donc sur le jeu de donnée large.

Comme mentionné précédemment, nous souhaitons étudier plus précisément le domaine du traitement d'image. C'est pourquoi, on ajoute à la suite Mibench une suite constituée d'applications de traitement d'images d'applications développées en interne que nous appellerons la suite COTS. Cette suite d'applications est spécialisée dans le traitement d'image et les codes sont écrits en C et parallélisé en OpenMP. Cependant, pour cette étude, nous utiliserons les versions séquentielles de ces applications. La liste des applications de cette suite est résumée Tableau 3.1 avec les données d'entrées utilisées.

3.1.2 Résultats

Maintenant que nous avons détaillé la méthode d'expérimentation et les applications étudiées, les résultats de la détection sont présentés figure 3.4. Deux phénomènes peuvent être observés. Premièrement, les données en lecture seule représentent en moyenne 60% de l'ensemble des données utilisées et 24.7% des accès mémoires, ce taux de détection des données en lecture seule est donc important. Cela peut être expliqué en partie par le fait que les applications étudiées sont des

TABLE 3.1 – Description des applications parallèles de la suite COTS

Application	Donnée d'entrée	Description
matrix_multiply	tableau de 512x512	multiplication de matrices
deriche	image 1024x1024px	Filtre de Canny pour la détection de contour d'une image
rotate	image 1024x1024px	Algorithme de rotation d'une image
max33	image 1024x1024px	Algorithme de flou moyen appliqué à une image
median33	image 1024x1024px	Algorithme median appliqué sur une image
jacobi	matrices 1024x1024	Algorithme itératif qui résout un système d'équation linéaire
adi	tableaux à 256 éléments 10 itérations	Algorithme de résolution de systèmes d'équations non-linéaires
wodcam	1000 images (168x192px)	Application de reconnaissance de visage basée sur la méthode *eigenface*

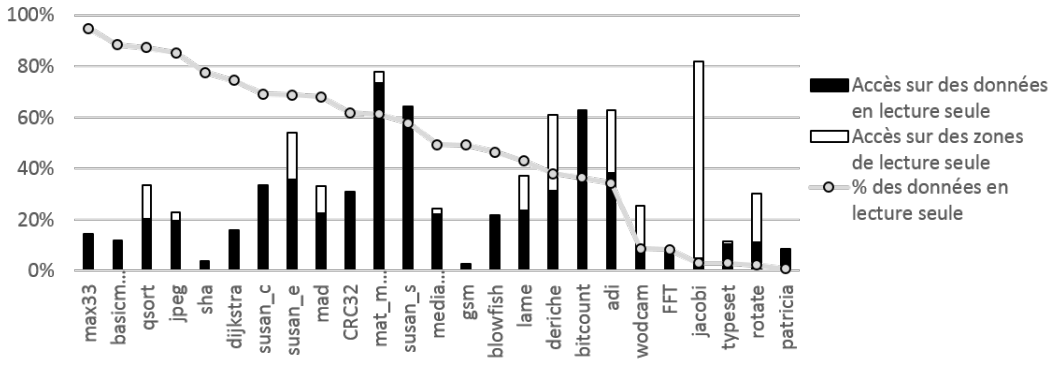


FIGURE 3.4 – Détection des données en lecture seule et des zones en lecture seule sur les suites Mibench et COTS

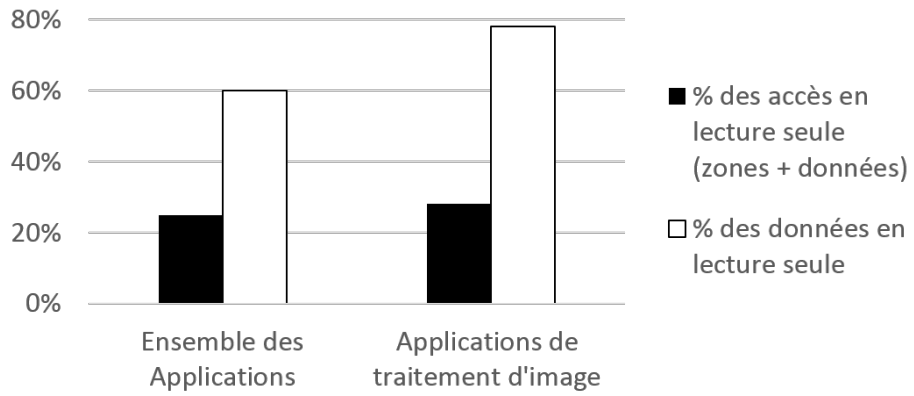


FIGURE 3.5 – Résultat de la détection pour les applications de traitement de signal et d'image

applications de tests et non pas des applications utilisées dans des environnements réels. Les données en entrées et en sorties de l'application ne sont pas réutilisées, les données d'entrées sont donc nécessairement des données en lecture seule. Par exemple, l'application *susan_edge* effectue une détection des contours sur une image et l'image d'entrée est donc considérée comme une donnée en lecture seule. Dans un système plus réaliste, ce calcul serait intégré dans une chaine plus complète, et l'image d'entrée viendrait d'un flux d'image comme une caméra, qui écrirait régulièrement le buffer associé à l'image d'entrée. Dans ce cas, l'image d'entrée ne serait pas considérée comme une zone mémoire en lecture seule pendant toute l'application mais posséderait des zones de lecture seule. C'est pourquoi il est important de considérer également les zones de lecture dans notre analyse. Deuxième, malgré la disparité des résultats, la tendance générale est une asymétrie importante entre la proportion des données en lecture seule et la proportion des accès qu'elles représentent. Cette asymétrie suggère que les données en lecture seule ne sont pas autant réutilisées que les autres données. De toutes ces applications étudiées, on forme un sous-groupe d'applications constituées d'applications de traitement de signal et d'image afin d'étudier les particularités de ce domaine applicatif. Ce sous-groupe est constitué de toutes applications de la suite COTS et des applications *susan*, *jpeg*, *mad*, *lame*, *fft*, *gsm* des différents algorithmes de manipulation des images au format tiff pour la suite Mibench. La figure 3.5 montre que l'asymétrie observée entre proportion d'accès et proportion de données est plus importante pour ce sous-groupe d'applications que pour l'ensemble des applications.

On constate également beaucoup d'applications pour lesquelles aucune zone de lecture seule n'est détectée. Ces dernière apparaissent essentiellement sur les applications de traitement d'image et elles correspondent dans ce cas à des images intermédiaires du pipeline d'exécution de ces applications tel que expliqué 3.1.1. Des expériences complémentaires ont été effectuées pour mesurer l'impact du seuil de détection des zones en lecture seule fixé jusqu'à présent à 1024 accès. Du fait

que les zones de lectures seule existantes sont principalement ces images intermédiaires, passer d'un seuil de détection de 1024 à 128 accès n'augmente que faiblement le nombre d'accès détecté sur des zones de lecture (2% en moyenne). En revanche, l'augmentation du seuil de détection jusqu'à 16384 accès ne permet plus de détecter ces images intermédiaires ce qui réduit drastiquement le taux de détection sur les applications de traitement d'image qui passe alors de 9,4% à moins de 3% en moyennes.

Cette analyse montre que les données et les zones en lecture seule représentent une part significative des accès effectués et des données manipulées par les applications dans les systèmes embarqués. Il s'agit maintenant d'illustrer plus précisément la différence de comportement des deux types de données par une analyse plus poussée de la localité des données.

3.2 Localité des données en lecture seule

Nous voulons maintenant étudier plus précisément la différence en termes de localité des données, entre les données en lecture seule et les données en lecture écriture. La localité des données est une propriété abstraite et des métriques ont été proposées dans la littérature afin de pouvoir l'évaluer. Cette métrique sera d'abord introduite puis appliquée à notre cas d'étude.

3.2.1 Introduction à la distance de réutilisation

Afin de pouvoir évaluer les propriétés de localité des données d'une application, de multiples métriques ont été proposées. Celle qui va nous intéresser ici est la distance de réutilisation. Elle a été introduite pour la première fois en 1970 par Mattson et al. [74], elle est définie comme le nombre d'accès à des données différentes entre deux accès à une même donnée. Le mot donnée ici peut être défini selon un grain plus ou moins important, on peut par exemple calculer la distance de réutilisation à l'échelle d'un bloc de cache. Dans ce cas, cette métrique permet de capturer et la localité spatiale et la localité temporelle des données dans le cache. Un exemple est proposé avec le tableau 3.2. Le bloc de cache considéré ici fait 16 octets. A noter que lorsque qu'une donnée est accédée pour la première fois, la distance de réutilisation associée est infinie. Dans l'exemple, on accède d'abord à la donnée à l'adresse 0, la distance de réutilisation est infinie car c'est le premier accès. Puis, on accède à la donnée située à l'adresse 8, cette donnée étant située sur la même ligne de cache que la donnée précédente, la distance de réutilisation est donc 0 lorsque la granularité est au niveau bloc de cache et infinie si on considère les adresses séparées. On voit alors que la granularité au niveau bloc de cache permet de rendre compte de la localité spatiale. Dans la suite de l'étude, la distance de réutilisation sera toujours calculée à l'échelle d'un bloc de cache de 64 octets ce qui correspond à une valeur standard utilisée actuellement sur les architectures

TABLE 3.2 – Exemple de calcul de la distance de réutilisation à une granularité de l'adresse et du bloc de cache

Numéro de l'accès mémoire	1	2	3	4	5	6	7
Adresse	0	8	16	96	8	16	104
Numéro de la ligne de cache	0	0	1	6	0	1	6
Distance de réutilisation (Grain : Adresse)	∞	∞	∞	∞	2	2	∞
Distance de réutilisation (Grain : bloc de cache)	∞	0	∞	∞	2	2	2

modernes embarquées.

Le résultat du calcul de la distance de réutilisation que l'on représente habituellement sous forme d'histogramme, permet de déduire directement et exactement le taux de défauts de cache pour n'importe quelle taille de cache pleinement associatif. De façon générale, les optimisations en localité s'emploient à à minimiser les distances de réutilisation, car les accès à une même donnée seront alors plus proche, il est donc plus probable que le bloc soit présent dans le cache. Le calcul des distances de réutilisation pour tous les accès mémoires représente cependant un calcul lourd à effectuer et c'est pourquoi beaucoup d'algorithmes ont été proposées pour la calculer efficacement. Pour notre part, nous utiliserons l'algorithme proposé par Bennett et al. [75]. Sa complexité est en $N * log(N)$ ou N est le nombre d'accès considéré, nous considérerons cette complexité suffisamment faible pour la taille de nos traces mémoires.

Pour pouvoir évaluer la différence en termes de localité de la séparation proposée, on calculera la moyenne algébrique des distances de réutilisation sur chacune des traces mémoires de l'expérimentation précédente. Les accès avec une distance de réutilisation infinie correspondent au premier accès à la donnée, et se traduiront par des défauts de cache obligatoires. Ces défauts ne peuvent être résolus qu'à l'aide de pré-chargement et la séparation du flux mémoire ne peut rien pour résoudre ces défauts de caches, ils ne seront donc pas étudiés ici.

De façon générale, séparer la trace mémoire principale en plusieurs sous-ensembles diminue nécessairement les distances de réutilisations si les deux ensembles forment deux groupes de données exclusifs. Dans notre situation, ce n'est pas le cas, ce qui fait qu'une séparation peut abimer la réutilisation des données lors du changement de statut de la donnée. Il faut donc éviter cette situation en considérant des plages de lecture seule les plus longues possibles.

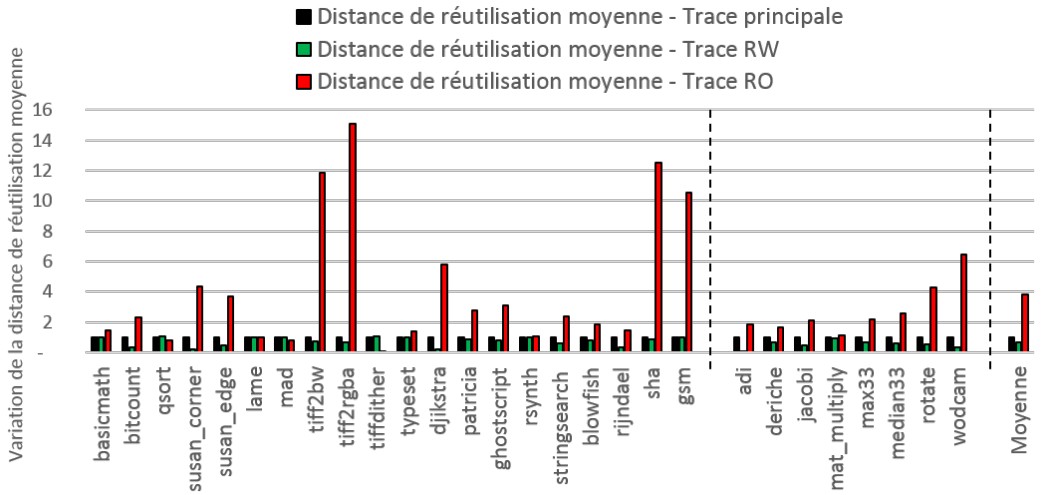


FIGURE 3.6 – Calcul des distances de réutilisation moyennes sur chacune des traces mémoires

3.2.2 Évaluation

Pour les applications des suites COTS et Mibench, on calcule la distance de réutilisation moyenne sur la trace complète, la trace RO et la trace RW et celles des traces RO et RW sont rapportées par rapport à celle de la trace complète. Les résultats présentés sur la figure 3.6 permettent d'observe que pour la trace RO, les distance de réutilisation moyenne sont 3,8 fois supérieure en moyenne par rapport à la trace complète alors qu'elles sont diminuées de 32% pour la trace RW. Cet effet est observé aussi bien pour les applications séquentielles de la suite Mibench que pour les applications parallèles de la suite COTS. Le résultat met en évidence un phénomène de pollution de la part des données en lecture seule. Ce phénomène survient lorsque des données sont importées dans un cache et ne sont pas réutilisées dans le cache avant d'en être évincées du fait d'une distance de réutilisation trop importante. Ce phénomène est un cas pathologique de l'utilisation d'un cache et doit être évité. Comme les données en lecture seule ont une distance de réutilisation importante, elles utilisent moins efficacement la ressource en cache par rapport à d'autres données et le fait de les retirer du flux mémoire principal permet d'augmenter la localité des données présentes et donc d'éviter des phénomènes de pollution. On observe donc deux types de données qui se comportent de façon bien distinctes en termes de localité.

3.2.3 Analyse qualitative avec un exemple

Afin d'illustrer comment se traduit, dans le code, cette différence de localité peu intuitive, on s'intéresse plus précisément à l'algorithme *rotate* de la suite COTS. Le pseudo-code de l'algorithme présenté figure 3.1, correspond à un algorithme de rotation d'une image d'entrée *img_in*, le résultat étant écrit sur l'image de sortie *img_out*, l'angle de rotation étant ici fixé à 45°. L'algorithme détermine d'abord la taille de l'image de sortie et l'alloue en mémoire. Puis, une boucle itère sur chaque pixel de l'image de sortie, la rotation inverse est calculée afin de déterminer les coordonnées en flottant de la position du pixel de l'image d'entrée correspondante. Enfin, une interpolation est effectuée à partir des quatre pixels de l'image d'entrée les plus proches du point trouvé, pour obtenir la valeur du pixel de l'image de sortie. En termes d'analyse des accès mémoire, pour chaque itération de boucles, l'image d'entrée est accédée quatre fois en lecture et l'image de sortie une fois en écriture. Le compilateur est généralement capable de placer les différentes variables d'index x et y dans des registres, car elles sont accédées très souvent. Les autres accès mémoires effectués sont les variables scalaires de la pile : *angle*, *pi*, *pj* ... L'image de sortie est donc accédée de façon uniforme, pixel par pixel, alors que les accès à l'image d'entrée sont dépendants de l'angle de rotation défini. L'algorithme de détection identifie une zone de lecture sur l'image d'entrée durant l'exécution de cette boucle, la trace RO est donc essentiellement composée des accès à l'image d'entrée dans ce cas.

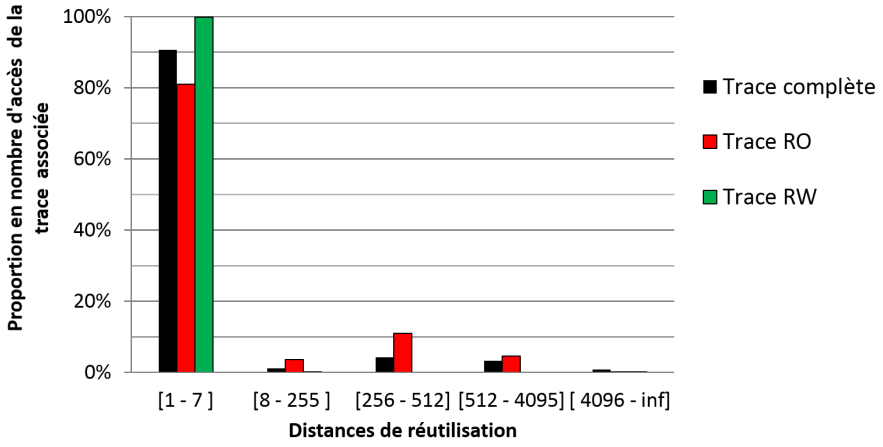
Listing 3.1 – Pseudo-code de l'algorithme *rotate*

```
int img_in[SIZE][SIZE];
//Lecture de l image pour remplir img_in
...
//Allocation de l image de sortie et calcul des index
int x0, x1, y0, y1;
int *img_out = allocOutputMatrix(angle, &x0, &x1, &y0, &y1);
...
#pragma omp parallel for ...
for(int x =x0 ; x <= x1 ; x++)
    for(int y =y0 ; y <= y1 ; y++)
        //Calcul des coordonnees flottantes dans l image
        //d entree du pixel de sortie
        int i = (int) (x * cos(-angle) - y * sin(-angle))
        int j = (int) ( y * sin(-angle) + x * cos(-angle))

        //Interpolation a partir des 4 pixels les plus
        //proches de l image d entree
        img_out[x][y] = (img_in[i][j] + img_in[i][j+1] \\
                + img_in[i+1][j] + img_in[i+1][j+1])>>2;
```

Sur l'histogramme de la trace complète de la figure 3.7, on peut voir que 92% des accès mémoires ont une distance de réutilisation inférieur à 7, les 8% restants possèdent une distance de réutilisation importante, cette part d'accès correspond

FIGURE 3.7 – Histogrammes des distances de réutilisation sur chacune des traces mémoires pour l'algorithme *rotate*



aux accès irréguliers faits à l'image d'entrée. En effet, on observe que ces accès sont essentiellement reportés sur la trace RO qui ne contient que des accès faits sur l'image d'entrée. Le fait de séparer les accès RO permet alors d'enlever ces accès de la trace principale et de ne conserver les accès avec les distances de réutilisation très faibles. C'est pourquoi dans ce cas, la distance de réutilisation moyenne des données de la trace RO est plus élevée. En termes de défauts de cache, un cache 32kO pleinement associatif générera des défauts de cache pour tout accès dont la distance de réutilisation est supérieur à 512. La trace complète sur un tel cache générera un taux de défauts de cache de 8% alors que la trace RW seulement 0.1% (sans compter les défauts de cache obligatoires). Cela permet d'illustrer la pollution engendrée par les données en lecture seule sur le flux mémoire principal. De plus, les accès RO étant isolés du reste du flux mémoire, cela permet de réduire leurs distances de réutilisation avec potentiellement une résolution des défauts de caches associés à ces accès.

3.3 Variation des résultats

L'expérience proposée précédemment illustre des distances de réutilisations significativement plus importantes pour les données en lecture seule que le reste des données. Afin d'élargir la portée de cette étude et de renforcer notre conclusion, on propose dans cette section d'approfondir ce constat en étudiant les variations à certains paramètres expérimentaux. Parmi les nombreux paramètres dont dépendent cette expérience, on peut lister :

— La taille des données d'entrée ou le volume du problème que l'on donne à résoudre à l'application
— L'implémentation particulière des algorithmes utilisés

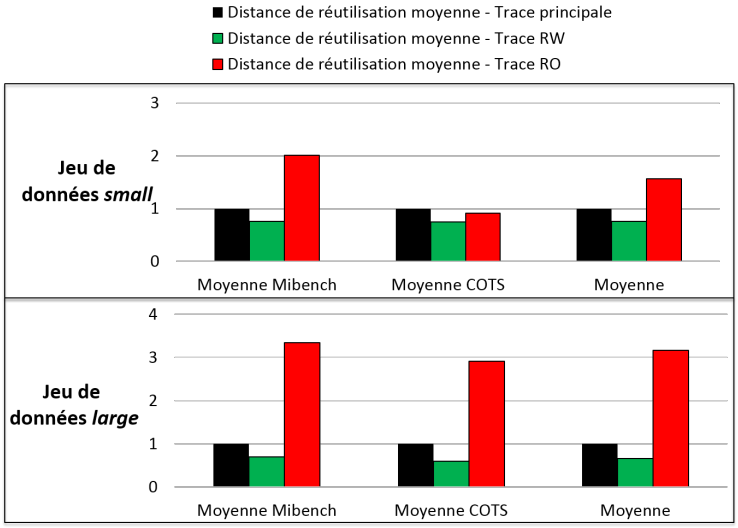


FIGURE 3.8 – Variation des résultats selon la taille du problème

— Les données d'entrées : selon les données d'entrées, les chemins d'exécution dans l'application peuvent être différents
— La façon dont est écrit le code par l'utilisateur ou la façon dont il est transformé par le compilateur.
— Le jeu d'instruction utilisé

Nous étudions dans la suite de ce chapitre d'étudier plus en détails quelques-uns de ces paramètres à savoir le problème de la taille des données d'entrées et la variation aux optimisations du compilateur.

3.3.1 Variation des résultats avec la taille du jeu de données d'entrées

L'évaluation des résultats pour des volumes plus importants de données est limitée par notre méthode d'expérimentation qui utilise des traces mémoires. Le problème des traces mémoires est que même compressées, elles deviennent rapidement volumineuses et il est assez difficile de représenter un nombre d'accès important. Les applications étudiées ici effectuent quelques milliard d'accès mémoires au mieux ce qui représente au moins un ordre de grandeur inférieur à des applications actuelles [76]. Afin de comprendre comment les résultats s'extrapolent pour des volumes d'accès mémoire plus importants, on effectue deux points de mesures à partir de deux jeux de données d'entrées. Cela permet d'indiquer une tendance quant à l'évolution des résultats par rapport à ce paramètre. On utilise pour la suite Mibench, les deux jeux de données d'entrées proposés avec chaque application, le jeu de donnée *large* et *small*. Pour la suite COTS, un deuxième set de données d'entrées est plus petit est fabriqué à partir de taille d'image plus petites.

Les résultats de l'analyse des distances de réutilisation montrent une nette

augmentation des distances de réutilisation avec l'augmentation du jeu de données d'entrées. Pour le petit jeu de données, la distance de réutilisation moyenne de la trace RO est 56% plus élevé que la trace principale alors que pour le jeu de donnée large, ce même rapport est de 216%. On peut identifier plusieurs types d'accès. Les motifs d'accès réguliers comme celui de l'image de sortie de l'algorithme *rotate* du code 3.1 ne dépendent pas directement de la taille des zones parcourues. Pour les motifs d'accès irréguliers, tous les codes ne sont pas impactés de la même façon mais dans de nombreux cas comme dans celui de la matrice accédée en colonne de l'algorithme de multiplication de matrice, augmenter les données aura tendance à augmenter les distances de réutilisation. Ce qui a donc tendance à exacerber les différences entre distances de réutilisation ce qui permet donc de renforcer les conclusions de l'étude précédente.

3.3.2 Variation des résultats avec différentes optimisations de localité

Comme nous avons pu le voir dans le chapitre précédent, le compilateur peut effectuer des transformations de code pour améliorer la localité des données. Selon les techniques de transformations utilisées, on observe que cela peut avoir un impact important sur le taux de détection des données et des zones de lecture seule, et également sur les distances de réutilisation. Dans cette section, on étudie l'impact de telles transformations sur les résultats obtenus par notre analyse.

On peut observer par exemple deux versions du code source du noyau de l'application *jacobi*, une version simple et une version transformée et optimisée pour la localité via l'outil PLuTo [77], outil de transformation de code basé sur le modèle polyédrique. Les deux codes produisent les mêmes résultats, et pourtant le résultat de l'analyse des données en lecture seule est très différent. Sur la version du code de la figure3.9-a, le calcul est décomposé en deux sous nids de boucles, et l'étude détecte pour une valeur de N suffisamment grande, une zone de lecture seule pour le tableau A, pour le premier nid de boucle et une zone de lecture pour le tableau B dans le deuxième nid de boucle. Sur la version optimisée, les deux tableaux sont détectés en lecture-écriture, et donc aucune zone de lecture seule ne peut être détectée. La version optimisée ici combine plusieurs techniques de transformations de boucles dont un changement de variable d'index qui permet de retirer les dépendances entre itérations de la boucle imbriquée ce qui permet ensuite d'effectuer une fusion de boucles puis un tuilage selon les différentes dimensions du tableau. Il s'agit donc ici d'un cas pathologique de l'étude ou selon la version du code étudiée, les conclusions peuvent s'opposer.

Toutes les optimisations n'influencent donc pas les résultats de la même manière et les résultats obtenus doivent effectivement être considérés selon le niveau d'optimisation proposé. Dans le cas de l'algorithme *jacobi*, c'est la fusion des deux boucles qui empêche toute détection des données en lecture seule. Parmi toutes les optimisations en localité existante, le tuilage de boucles est plus précisément étu-

```
for(t = 0 ; t < T ; t++) {

 for(i = 2; i < N; i++)
  for(j = 2; j < N; j++)
   B[i][j] = 0.2*( A[i][j]+A[i][j-1]+A[i-1][j]+\
                   A[i][j+1]+A[i+1][j]);

for(i = 2; i < N; i++)
  for(j = 2; j < N; j++)
   A[i][j] = B[i][j];
}
```

(a) Jacobi-2D

```
for(tT ...)
 for(iT ... )
  for(jT ... )
   for(t ... )
    for(i ... )
     for(j ... )
          B[-2*t+i][-2*t+j] = 0.2*( A[-2*t+i][-2*t+j]+...-
          A[-2*t+i][-2*t+j-1] = B[-2*t+i-1][-2*t+j-1];
```

(b) Jacobi -2D après transformation avec PLuTo

FIGURE 3.9 – Deux versions du code de *jacobi*, (a) Version simple et (b) Version optimisée en terme de localité des données avec PLuTo

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  for (k = 0; k < N; k++)
   C[i][j] += A[i][k]*B[k][j];
```

(a) Simple

```
for (jT = 0; jT < N/J; j++)
 for (k = 0; k < N/K; k++)
  for (i = 0; i < N; i++)
   for (j = jT; j < (jT+1)*N ; j++)
    for (k = kT; j < (kT+1)*N; k++)
     C[i][j] += A[i][k]*B[k][j];
```

(b) Tuilage 2D

```
for (iT = 0; iT < N/I; j++)
 for (jT = 0; jT < N/J; j++)
  for (k = 0; k < N/K; k++)
   for (i = iT; i < (iT+1)*N; i++)
    for (j = jT; j < (jT+1)*N ; j++)
     for (k = kT; j < (kT+1)*N; k++)
      C[i][j] += A[i][k]*B[k][j];
```

(c) Tuilage 3D

FIGURE 3.10 – Les versions de *matmul*, (a) Simple (b) Tuilage 2D et (c) Tuilage 3D

dié. Dans ce cas, ce sont les distances de réutilisations sont modifiées plutôt que la détection des données en lecture seule et nous allons nous intéresser dans la suite de cette section plus précisément à l'impact du tuilage sur la différence relative de localité observé entre les données en lecture seule par rapport aux autres. On regarde l'influence de cette technique sur l'algorithme de multiplication de matrices *matmul*. Ainsi, plusieurs versions de l'algorithme ont été implémentées : une version simple, une version avec un tuilage sur deux dimensions et une version avec tuilage sur trois dimensions. Le code est reporté sur la figure 3.10. En terme de détection de données en lecture seule, dans les trois exemples, les résultats sont identiques à savoir les deux matrices d'entrées A et B sont détectées en lecture seule et C est en lecture-écriture.

Sur la figure 3.11, on applique l'analyse précécente avec différentes formes de tuiles. Dans tous les scénarios, on observe que la différence de localité qui existe pour la version simple (cf figure 3.6) est également observée pour les versions avec tuilage avec toutes les formes de tuile évaluées. La différence des distances de réutilisation entre les données en lecture seule et les autres est dû à la construction de l'algorithme. L'implémentation étant faite en C, les tableaux sont donc stockés ligne par ligne dans la mémoire. Donc dans ce cas, les accès mémoires produisant une distance de réutilisation importante sont ceux effectués sur l'image B entre la réutilisation de la ligne de cache entre les accès B[i][j] et B[i][j+1], le tuilage permet

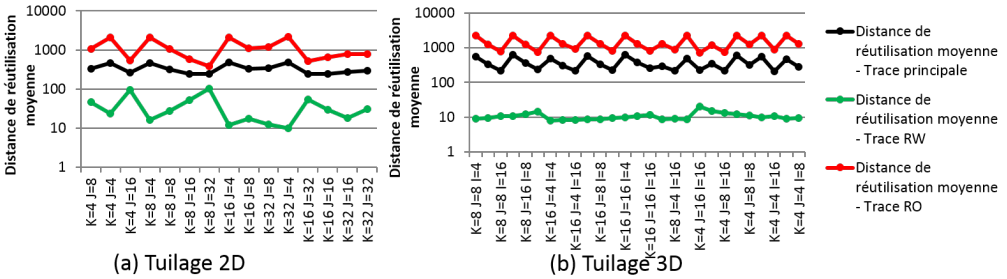


FIGURE 3.11 – Variation des distances de réutilisation en fonctions des paramètres de tuiles K et J, pour le tuilage 2D et K,J,I pour le tuilage 3D

de réduire cette distance. Dans la version simple, ces accès sont séparés par N itérations alors que dans les versions avec tuilage, ils ne sont plus séparés par K itérations. Cependant, pour les autres matrices, on peut observer une réutilisation des valeurs de C[i][j] et A[i][j] sur la boucle la plus profonde entre chaque itération (localité temporelle). Donc par construction, il y aura toujours une distance de réutilisation moyenne supérieure sur les données en lecture seule du fait du motif d'accès à B et ceci quel que soit la forme de la tuile adoptée. Le tuilage en tant qu'optimisation de la localité des données ne remet donc pas en question les conclusions apportées par l'étude précédente et ce indifféremment de la forme de tuile adoptée pour le tuilage.

La détection des données en lecture seule et l'analyse des distances de réutilisation dépend nécessairement de la façon dont le code est écrit, et les résultats proposés ne sont pas invariants aux différentes transformations de code. Cependant, les optimisations de localité appliquées n'invalident pas nécessairement les conclusions de l'étude précédente. Afin de s'affranchir de la transformation effectivement réalisée, on peut poser la question de d'ordonnancement des instructions qui peuvent former ou non une zone de lecture seule sur une région de mémoire selon la façon dont elles sont ordonnancées. Cet ordonnancement est limité par les dépendances, ou les points de synchronisation de l'application comme cela peut exister sur les pipelines d'exécution. Ainsi, si l'exemple de jacobi utilisé plus haut 3.9, les vecteurs de dépendances expriment le fait que l'instruction d'écriture de A[i][j] doit impérativement être effectuées après la lecture des points A[i+1][j],A[i][j+1], A[i-1][j], et A[i][j-1] et ce qu'importe les transformations finalement effectuée. De même, cette fusion de boucle ne serait pas possible si l'on imagine une version parallèle du code avec des points de synchronisation entre les deux boucles. De par la diversité des codes étudiés et l'étude du tuilage, on considère ces résultats suffisamment généraux pour notre étude.

3.4 Conclusion

Nous avons caractérisé dans ce chapitre, le comportement des données en lecture seule sur un ensemble d'applications Notre étude nous permet de conclure sur plusieurs points :

— Les données et les zones en lecture seule représentent une part importante de l'ensemble des données utilisée ainsi qu'une quantité d'accès mémoire significative

— Il a été mis en évidence un comportement bien distinct en termes de localité des données en lecture seule. Les données en lecture seule présentent des distances de réutilisation plus importantes que les autres et créent donc une pollution sur le flux mémoire principal

De cette étude ressort qu'une proposition de gestion spécifique dans la hiérarchie mémoire des accès sur les données et les zones en lecture seule est pertinente car elle permet d'améliorer la localité des données de chacun des deux flux mémoires. Maintenant que nous avons caractérisé les données en lecture seule et illustré leurs comportements, la deuxième condition nécessaire pour qu'une solution architecturale puisse être proposée, est d'être capable d'identifier et de placer ces accès mémoire sur le bon chemin de données de façon automatique afin que cette solution puisse être utilisée de façon transparente pour l'utilisateur.

Chapitre 4

Classification des données en lecture seule pour des architectures parallèles

Sommaire

"Science is the systematic classification of experience"

George Henry Lewes

Nous avons pu voir dans le chapitre précédent une étude hors-ligne des données en lecture seule, le but était d'illustrer le comportement spécifique des données en lecture. Afin de pouvoir proposer une solution matérielle permettant une gestion différenciée dans la hiérarchie mémoire de ces données, il faut être capable d'opérer une classification des données soit à la compilation soit à l'exécution afin que l'architecture soit consciente du type des données manipulées.

4.1 Introduction

Comme nous avons pu voir dans le chapitre 2, plusieurs propositions de classification ont déjà été proposées dans ce sens dans la littérature pour plusieurs types de données notamment pour les données en lecture seule. La seule méthode de classification de données proposée à la compilation est celle proposée par Li

et al.[67, 65]. Cette méthode détaillée précédemment 2.2.2, effectue une classification des données privées à la compilation pour diminuer l'impact du protocole de cohérence. Elle illustre la difficulté que peut représenter une classification à la compilation avec la notion de donnée privée qui peut être difficile à prouver strictement à la compilation. Dans ce cas, le compilateur est obligé d'adopter des comportements conservatifs afin de garantir que les données détectées soient bien du type que l'on cherche à isoler. Une détection plus souple peut donc être envisagée si le taux de détection est trop faible. Elle permet de passer outre les limitations du compilateur et d'augmenter l'efficacité de la détection mais oblige à penser des mécanismes de rétablissement matériel en cas d'erreur. Cependant, l'erreur effectuée sur la détection et le cout du mécanisme de rétablissement en cas d'erreur doivent rester faible afin d'utiliser correctement ce genre d'approche. Cette approche utilisée dans la solution de Li et al.[67] pour la détection des données privées, des hypothèses sont émises quant à la façon dont les données privées sont accédées ce qui permet d'augmenter le taux de détection des données privées, passant de 2% à plus de 50% en moyenne avec un taux d'erreur sur la détection restant faible. Une détection souple est donc à considérer si l'approche conservative n'atteint pas des taux de détection suffisants.

Ces travaux proposent une détection des données en lecture seule à la compilation, avec cette propriété dynamique vue précédemment, cela revient donc plutôt à détecter des zones de mémoires en lecture sur certaines portions du code. L'avantage d'une telle méthode est qu'elle permet une détection plus précise des données classifiées qu'au niveau matériel ou les solutions contraintes en terme de surface ne peuvent proposer que des méthodes simples de classification et de changement de statut. Avec un compilateur, ces derniers peuvent être analysés de façon plus précise. De plus, nous avons précédemment montré l'intérêt particulier des applications de traitement du signal et d'image, dans la séparation des données en lecture du chemin de données principal. Ce domaine applicatif effectue principalement ses calculs dans des boucles et comme vu précédemment 2.1.1, le domaine de l'analyses de boucle par le compilateur a été largement étudié notamment pour améliorer la localité des données. De plus, un compilateur avec une interface appropriée avec l'architecture permet de travailler à une granularité du bloc de cache.

4.1.1 Objectif

Dans ce chapitre, l'objectif est donc de proposer une méthode de classification des données en lecture seule à la compilation afin d'obtenir un niveau de détection suffisant par rapport à l'analyse hors-ligne. L'analyse hors-ligne développée dans le chapitre précédent pour la détection des données en lecture seule permet d'observer un certain niveau de détection des données en lecture seule. Si ce niveau de détection ne constitue pas un niveau théoriquement optimal, car il dépend de la taille des zones de lecture seule que l'on veut considérer, il peut cependant être

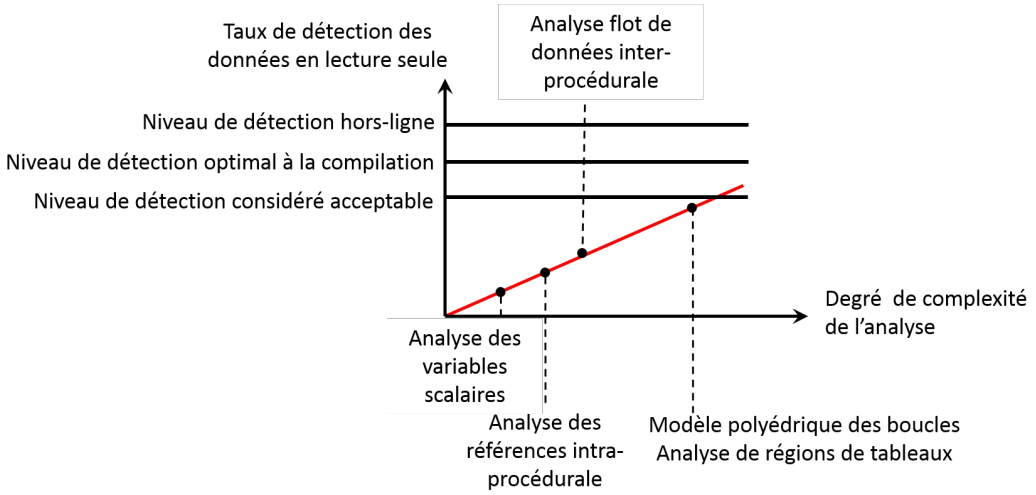


FIGURE 4.1 – Evolution de la courbe de détection en fonction du niveau de complexité d'analyse effectué (les niveaux indiqués sont indicatifs)

utilisé comme référence par rapport au taux de détection d'une solution proposée par le compilateur. Pour cela, dans la prochaine section, différentes méthodes d'analyse flots de données sont proposées et dans la dernière section, notre implémentation sous GCC sera présentée. Le but ici n'est donc pas de développer de nouvelle analyses de flots de données mais plutôt de s'appuyer sur des techniques existantes à la compilation déjà maitrisées pour montrer la pertinence de l'utilisation du compilateur pour la détection des données en lecture seule.

4.2 Classification des données en lecture seule à la compilation

Nous proposons de diviser cette classification des données en deux étapes. La première constitue la détection des données en lecture seule proprement dite et la seconde étape propose un mécanisme permettant de transmettre l'information de classification à l'architecture. Cette interface compilateur/architecture conditionne la granularité de la détection.

4.2.1 Introduction à la classification des données à la compilation

Il existe une littérature importante sur les analyses de flots de données et c'est un bon point d'entrée pour notre analyse. Le rôle de ces analyses est d'étudier à la compilation, le comportement des données au cours de l'exécution de l'application. Le sujet a été beaucoup étudié dans les années 70 et 80 car il permet un grand nombre d'optimisations standards comme la propagation de constantes ou la suppression de codes morts. Ces analyses sont généralement limitées par

la complexité algorithmique des solutions proposées. En effet, il a été montré par exemple, qu'une analyse flots de données comme l'analyse des alias de pointeurs que la complexité algorithmique devient rapidement NP-difficile, au fur à mesure que l'on autorise la gestion de fonctionnalités plus complexes comme des pointeurs multi-niveaux [78]. La solution à ce problème est de limiter la portée de l'analyse ou la précision des résultats. Par exemple, on peut demander si deux pointeurs sont des alias l'un de l'autre de façon précise (*must-alias*) de façon plus souple (*may-alias*). Dans le dernier cas, la réponse peut produire des faux positifs mais est moins complexe à mettre en oeuvre. C'est l'approche utilisée couramment dans les compilateurs de production tel LLVM ou GCC car ils doivent être capable d'analyser des codes très volumineux ne peuvent pas par conséquent implémenter des algorithmes NP-difficiles. Il est important de comprendre ces limitations car notre détection des données en lecture partagera les mêmes limitations que les analyses de flots de données sur lesquelles on s'appuiera.

La figure 4.1 représente schématiquement la courbe du taux de détection des données en lecture seule qui augmente au fur à mesure que les analyses utilisées sont complexes. La courbe de détection dépend également évidemment du code de l'application étudiée mais ce taux de détection reste nécessairement inférieur ou égal à celui de l'analyse hors-ligne. Ceci est du fait qu'une détection à la compilation sera de façon inhérente limitée dans son analyse par des indéterminations résolues uniquement à l'exécution.

Le niveau d'analyse le plus simple pour étudier le problème de détection des données en lecture seule s'intéresse aux variables scalaires. Afin de décrire les différents accès en lecture/écriture d'une variable, le compilateur construit des chaines de définition-utilisation [1]. Ces chaines permettent d'associer pour chaque définition (son écriture) de chaque variable, ses utilisations (ses lectures). Ce type de structure peut donc être utiliser pour déterminer les données en lecture seule. Un deuxième niveau d'analyse étudie le problème de la détection des périodes de lecture seule sur les références mémoire. Le problème revient à s'assurer que la région mémoire pointée n'est utilisée que par des références en lecture seule. Cette situation est directement reliée à celui de l'étude des alias de pointeurs. Ce type d'analyse est mené pour permettre des optimisations de code et de permettre la détection d'erreurs. Durant l'analyse des alias, les régions mémoires (dont celles en lecture seules) sont considérées comme des régions de mémoire homogènes et donc une seule écriture sur une région mémoire détectée comme étant en lecture seule suffit à invalider toute la zone. Afin de détecter plus finement les zones de lecture seule et éviter ce genre de situation, l'étude des régions de tableaux a également été proposée dans la littérature. Ces méthodes ont été développées dans le cadre d'optimisations comme la privatisation de portions de tableaux ou la génération de communication sur des architectures distribuées. Par exemple, à travers le modèle

1. *Def-Use chain*

polyédrique, Creusillet et al.[79] décrivent une méthode permettant de calculer par approximation des ensembles d'éléments de tableaux accédées soit en lecture (READ) ou en écriture (WRITE) durant l'exécution d'une boucle.

D'autres niveaux d'études peuvent également être considérés comme par exemple le fait que certains langages comme le langage C compilent fichier par fichier, la portée de la détection est donc limitée à l'unité de compilation courante. Cependant, certains compilateurs autorisent des optimisations directement durant la période d'éditions des liens où tous les fichiers objets sont fusionnés à l'intérieur d'un même fichier (l'exécutable), la représentation intermédiaire est alors reformée à partir du code objet et permet des optimisations avec une portée plus grande. On parle alors d'analyses en programme entier ou d'analyse LTO [2].

Plusieurs niveaux peuvent donc être envisagés pour effectuer une détection des données en lecture seule avec à chaque fois une analyse . Il s'agit ainsi de proposer une analyse pertinente vis-à-vis des applications que l'on souhaite étudier.

4.2.2 Interface Compilateur/Architecture

A partir des résultats de la détection, il s'agit d'être capable de transmettre cette information à l'architecture afin de placer la donnée sur le bon chemin de données. Il s'agit de rendre les différentes mémoires de la hiérarchie conscientes de la classification des données manipulées. Parmi les solutions étudiées dans le chapitre 2, beaucoup de solutions encodent l'information de classification des données au niveau page du système d'exploitation. Nous proposons ici d'utiliser une technique reposant sur un principe différent, introduite par la famille des architectures dites EPIC (*Explicit Parallel Instruction Computing*). La particularité de cette famille d'architecture est qu'elle repose sur un réordonnancement des instructions par le compilateur plutôt qu'au niveau matériel via le pipeline d'instruction. Une grande partie de la bonne utilisation de ces architectures reposent donc sur des optimisations à la compilation. Cette famille de processeurs est quasiment exclusivement représentée par les processeurs Itanium [80] développé par Intel et HP qui utilisent le jeu d'instruction particulier IA64 [81].

Ce jeu d'instruction utilise plusieurs mécanismes pour permettre au compilateur de transmettre au processeur des informations calculées à la compilation. Notamment, chaque requête de lecture/écriture utilise un champ de 2 bits dans lequel le compilateur encode une prédiction de la localité de la donnée accédée. Un processeur utilisant cet ISA (*Instruction Set Architecture*) peut alors utiliser cette information pour déterminer un placement des blocs de cache dans la hiérarchie de cache. Dans la littérature, ce mécanisme est appelé technique de cache collaboratif (*collaborative caching*) ou indication de cache (*cache hint*). Il a été montré que ce type d'interface permet une meilleure utilisation de la hiérarchie et réduit le taux de défauts de cache de 34% [82]. Dans la littérature, ce type d'interface a été étudié pour

2. *Link-Time Optimization*

d'autres optimisations que le placement des données dans la hiérarchie. Ainsi, Gu et al. [83] utilisent cette interface pour donner des indications au cache quant au choix de politique de remplacement à adopter pour le bloc de cache chargé. Wang et al. [84] s'en servent pour implémenter une solution permettant à certains accès de contourner le cache. Mukkara et al. [85] s'en servent pour proposer une solution au problème de placement des données sur les caches NUCA.

Nous proposons ici d'utiliser cette technique pour permettre la classification des données en lecture seule. A chaque requête de lecture/écriture, il est ajouté un bit permettant de déterminer si l'accès est effectué sur une donnée détectée comme étant en lecture seule par l'analyse statique détaillée précédemment. Cette technique de cache collaboratif a plusieurs avantages par rapport à la solution de Li et al. [67] vue précédemment qui à partir d'une analyse statique encode l'information de la classification dans les pages du système d'exploitation. Le premier avantage est que cela permet de travailler à une granularité du bloc de cache pour la classification, ce qui permet d'augmenter le taux de données détectées en lecture seule. Une des principales limitations de l'utilisation de cette technique dans les solutions proposées ci-dessus est que l'information transmise est alors principalement une information de localité qui dépend du contexte d'exécution tels que les paramètres d'entrées. Cette dépendance peut conduire à fournir à l'architecture des informations peu précises, les solutions implémentées perdent donc en efficacité. Dans notre situation, l'information fournie est avant tout sémantique, c'est-à-dire qu'elle dépend essentiellement de la façon dont le code est généré, ce qui est donc plus adapté à notre approche

Nous avons jusqu'à présent exposé quelques analyses possibles pour la détection des données en lecture seule et proposé une interface entre le compilateur et l'architecture. Nous allons maintenant développer une implémentation de cette détection dans le compilateur de production GCC.

4.3 Implémentation d'une classification des données en lecture seule sous GCC

Comme nous avons pu le voir précédemment, une détection des données en lecture seule à la compilation peut s'appuyer sur des analyses flots de données déjà implémentées dans le compilateur. Le choix de ce dernier est donc important dans les limitations apportées par la solution. Le compilateur GCC est choisi pour notre étude car il est très utilisé par la communauté et cela permettra d'étudier le potentiel de notre approche dans un compilateur de production qui se doit de posséder des analyses fortement contraintes en terme en complexité algorithmique. Dans cette section, nous détaillerons l'analyse implémentée dans GCC et les limitations de cette dernière.

4.3.1 Structure de GCC

Nous détaillerons d'abord succinctement le fonctionnement interne de GCC. Il s'agit d'un compilateur libre crée par le projet GNU, capable de compiler divers langages de programmation dont C/C++, Fortran, Java, Ada. De la même façon, il est porté pour un nombre importants d'architectures dont x86, ARM, SPARC, PowerPC. La structure de GCC est représentée figure 4.2. Sous GCC, chaque langage transforme sa représentation interne en une représentation intermédiaire appelée GENERIC [86]. Ce langage permet de décrire le code sous la forme d'un arbre de syntaxe abstrait (AST), indépendant du langage d'entrée utilisé. Ensuite, l'arbre décrit selon le langage GENERIC est traduit en langage GIMPLE par décomposition des expressions sous la forme d'un code à trois adresses, c'est-à-dire que chaque instruction utilise au plus trois opérandes. Ensuite, le graphe de flot de contrôle et la forme SSA *Static Single Assignement* [87] sont construits sur cette représentation.

Les différentes fonctionnalités ont été encodées via l'interface de plugin proposée par GCC, disponible depuis la version 4.5. Elle a pour but de permettre le développement de nouvelles fonctionnalités dans GCC sans avoir à modifier la chaine de compilations. Les plugins sont activés par le compilateur par des événements spécifiques qui fonctionne sur un système de fonction de rappel (*callback*). Cela permet ainsi de développer des fonctionnalités à différents niveaux de la chaine de compilation de GCC. Les analyses sont développées dans la partie optimisateur de code, ce qui permet d'obtenir une solution indépendante du langage.

4.3.2 Algorithme de détection des données en lecture seule

L'algorithme développé ici pour la détection des données en lecture, dépend largement des comportements observés dans les applications que l'on souhaite à étudier à savoir les applications de la suite COTS 3.1.1. Ces applications manipulent essentiellement tableaux et pointeurs dans des boucles, ces structures de données étant utilisées de façon homogène. L'algorithme de détection utilisé se base avant tout sur une analyse des références mémoire et peut s'appuyer sur une extension de l'analyse des alias avec une attention plus particulière sur les boucles imbriquées. L'algorithme 4.3.2 décrit l'analyse effectuée.

Pour chaque boucle, l'analyse parcourt le corps de cette dernière en maintenant à jour deux listes contentant les références mémoires utilisées, une pour les références en lecture seule et une autre pour les références en lecture-écriture. A la suite de cette étape, à partir de l'information de l'imbrication des boucles, les différentes listes sont fusionnées afin de retrouver pour chaque nid de boucles, les listes des références en lecture seule et en lecture-écriture. Enfin, l'analyse des alias nous permet de corréler aux références mémoires, des régions mémoires. Afin de considérer, une région mémoire comme étant en lecture seule, toutes les références qui pointent vers cette région doivent venir de la liste des références en lecture seule.

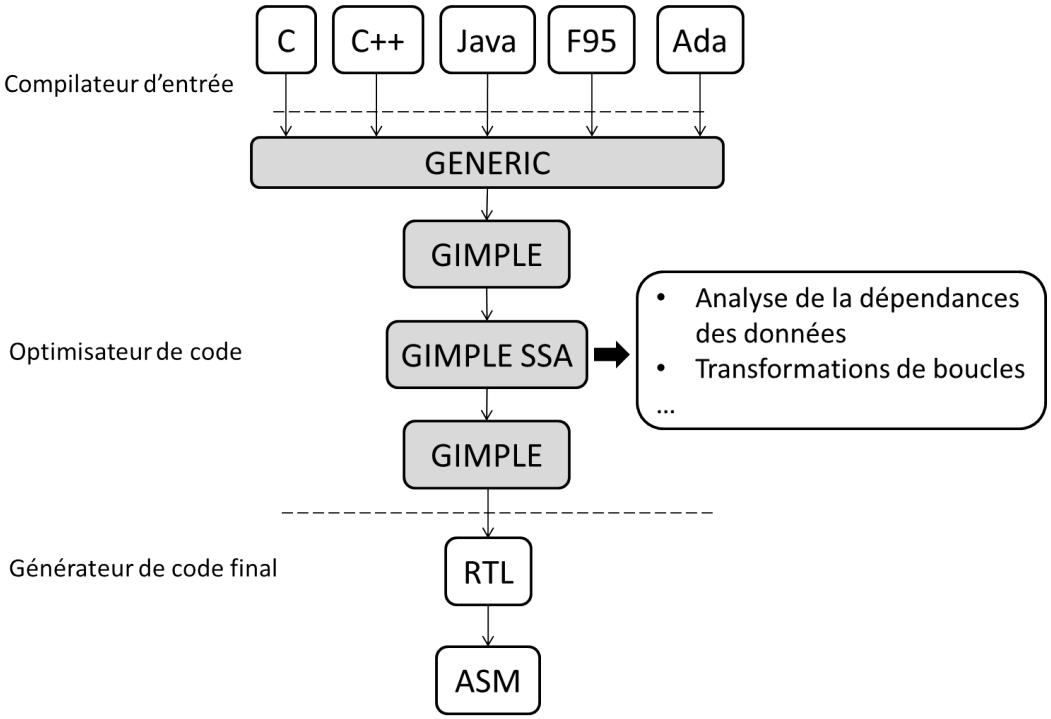


FIGURE 4.2 – Processus de compilation dans GCC

Algorithm 1 Algorithme de la passe intra-procédurale

```
1:  procedure INTRA-PROCEDURAL(Détection des données en lecture seule)
2:      for chaque instruction inst do
3:          for chaque operande op dans inst do
4:              if op est un accès tableau ou déférencement de pointeur then
5:                  base ← get_operand(op)
6:                  if base ∈ liste_ptr_RO ET op est ECRIT then
7:                      retire(base , liste_ptr_RO) ;
8:                      ajout( base , liste_ptr_RW) ;
9:                  else if base ∉ liste_ptr_RO ET op est LU then
10:                     ajout( base , liste_ptr_RO)
11:                 else if base ∉ list_ptr_RW ET op est ECRIT then
12:                     ajout( base , liste_ptr_RW)
13:                 end if
14:             end if
15:         end for
16:     end for
17: end procedure
```

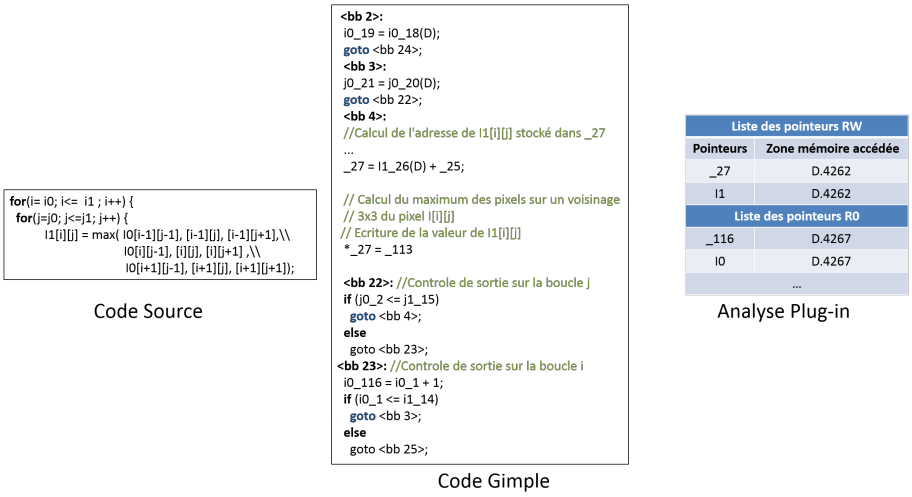


FIGURE 4.3 – Exemple d'application de l'analyse sur l'application *max33*

Un exemple d'utilisation de cet algorithme est développé figure 4.3 avec l'application *max33*. Le code source ainsi qu'une partie du code gimple sont représentés. A partir du code gimple, l'analyse parcourt les blocs de base de la boucle. Dans l'exemple, les deux boucles sont parfaitement imbriquées et le coeur de boucle est constitué des blocs basique de 4 à 21. Dans ces blocs sont détectées les différentes lectures/écritures des pointeurs. A partir de la liste des zones mémoires accédées, on peut déduire que la zone avec l'identifiant D.4267 adressée par I0 n'est utilisée que par des pointeurs en lecture est donc en lecture seule durant l'exécution de la boucle.

Il se pose également la question du placement de notre passe d'analyse par rapport aux autres passes du flot de compilation. Cette question reste importante car la représentation intermédiaire est largement modifiée au cours de ce flot et une zone mémoire détectée comme étant en lecture seule peut disparaitre à cause de ces transformations. Idéalement, notre passe intervient donc en dernier, juste avant l'étape de génération de code pour obtenir une analyse pertinente du code exécuté. Cependant, plus notre passe intervient tard, plus la représentation intermédiaire est de bas niveau et complexe à manipuler. De plus, notre analyse utilise plusieurs informations calculées par des passes précédentes comme le graphe de flot de contrôle, les informations de dominance des blocs (permet de retrouver les structures des boucles), et les informations d'aliasing. Dans des compilateurs comme LLVM, ce type de contraintes est renseigné au gestionnaire de passe qui décide alors de l'ordonnancement à effectuer pour minimiser l'espace mémoire nécessaire. Dans GCC, l'utilisateur doit décider explicitement du positionnement de sa passe. Notre passe est placée à la fin de la liste des passes spécifiques aux boucles (*tree-loonp*). D'une part car les informations nécessaires existent bien à ce moment de la compilation. Egalement, cette liste contient des passes importantes

de transformation de boucles notamment les passes associées à Graphite, plusieurs passes d'élimination de code morts spécifiques aux boucles, ou la passe d'optimisation de déroulement des boucles. Ces passes peuvent donc affecter notre analyse et on a donc tout intérêt à se placer après.

4.3.3 Analyse des alias de pointeurs sous GCC et limitations

L'algorithme développé dépend fortement de la capacité de l'analyse d'alias à décorreler les pointeurs, notre analyse partage donc les limitations de cette analyse. Il est détaillé maintenant le fonctionnement et les limitations que cette analyse nous apporte.

L'analyse des alias dans GCC est présente au niveau de l'optimisateur de code et au niveau RTL. Nous nous intéresserons ici à celle faite dans la partie optimisateur de code de GCC, car c'est à ce niveau que nos passes sont développées. Pour étudier l'aliasing entre deux pointeurs, GCC utilise principalement deux méthodes. La première se fait à travers les types des pointeurs. Cette méthode postule que deux pointeurs de type différents ne peuvent pas pointer vers la même zone mémoire, sinon cela conduit à un comportement indéfini. Par extension, cela est valable également pour chaque élément d'un objet composite. Une telle règle dépend du langage de programmation utilisé, et certains standards de langages imposent cette contrainte de programmation comme le C [88] ou le C++[89]. Cependant, cette règle est rarement strictement appliquée en pratique et c'est pourquoi cette analyse n'est pas activée par défaut dans GCC, il faut que l'option de compilation *-fstrict-aliasing* soit utilisée.

La deuxième méthode, la plus importante, construit pour chaque pointeur, un ensemble de régions mémoires que le pointeur peut effectivement référencer (*points-to set*). Le but de l'analyse est donc de réduire au maximum cet ensemble possible de régions mémoire que le pointeur peut effectivement adresser. Afin de déterminer cet ensemble, l'algorithme implémentée par Pearce et al. [90] dans GCC procède par l'étude de contraintes sur les ensembles (*Anderson-style*). Les principaux critères d'implémentation à prendre en compte dans le cadre d'une analyse des alias sont :

— La sensibilité au flot de contrôle : cela correspond à la prise en compte du déroulement de l'application dans le calcul des résultats. Ces résultats peuvent donc soit être calculés pour chaque point du programme (*flow-sensitive*), ou pour l'ensemble du programme
— La sensibilité au contexte : une analyse sensible au contexte ajoute de l'information sur la façon dont la fonction courante est appelée et permet de distinguer les appels des différentes fonctions appelantes.
— La modélisation du tas : décrit comment les régions mémoire allouées sur le tas sont modélisées.

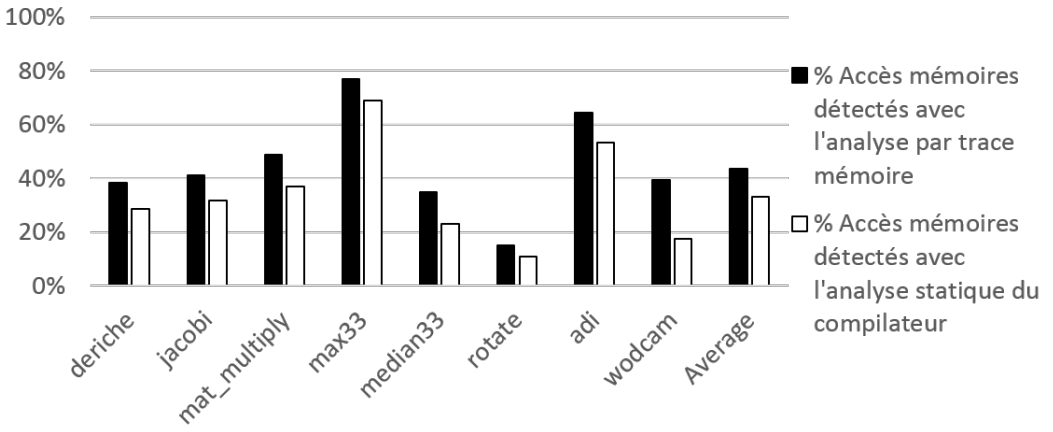


FIGURE 4.4 – Comparaison du nombre d'accès capturés entre l'analyse statique et l'analyse hors-ligne

— Analyse intra ou inter procédurale : l'analyse peut être effectuée soit sur une procédure ou sur l'ensemble de l'unité de compilation.
— La modélisation des structures de données agrégées : l'analyse peut faire le choix d'étudier séparémentou non chaque élément d'une structure composée comme les classes en C++.

Ces critères d'implémentations permettent d'effectuer un équilibre entre précision des informations calculés et complexité algorithmique de la solution. Dans le cas de GCC, l'analyse des pointeurs est intra-procédurale, non sensible au contexte et au flot de contrôle. Cependant, l'analyse prend en compte séparement les différents de structures agrégées. Les applications détectées ont été compilées suivant les limitations imposées par cette analyse.

4.3.4 Evaluation

Afin d'évaluer la qualité de la détection proposée, le taux de détection de l'analyse statique proposée dans ce chapitre est comparé à l'analyse hors-ligne effectuée dans le chapitre 3 sur les applications de la suite COTS. Les résultats présentés figure 4.4 montre un taux de détection importante, avec une différence entre l'analyse statique et l'analyse de 10,6% en moyenne. La différence de détection observée provient essentiellement de zones en lecture seule détectées par la détection hors-ligne qui arrive à former des zones de lecture seule en assemblant des accès en lecture éparpillés qui ne peuvent être étudiés à la compilation. Quelques-uns de ces artefacts subsistent malgré l'augmentation des seuils de détections des zones de lecture seule. Ce taux de détection apparait comme suffisant pour les applications que nous souhaitons étudier.

4.3.5 Emulation de la technique de cache collaboratif

Afin d'utiliser la technique de cache collaboratif proposée 4.2.2, il est nécessaire d'effectuer modification complexe du générateur de code x86 de GCC. Afin de limiter les temps de développement, une solution simplifiée à base d'instrumentation du code permettant de simuler un comportement similaire a été adoptée. Pour cela, on se propose d'encoder les zones de lecture seule détectée par l'analyse statique dans le binaire par la génération d'instructions spécifiques. Cette information peut être encodée en utilisant des instructions assembleur peu utilisées. Par exemple, le jeu d'instruction x86 possède 4 instructions assembleurs : *prefetchNTA*, *prefetchT0*, *prefetchT1*, *prefetchT2*, cela permet d'indiquer le placement dans la hiérarchie mémoire de la donnée à pré-charger. Ces instructions peuvent être soit écrites par l'utilisateur à l'aide de fonctions spécifiques ou générées par GCC via une passe de pré-chargement automatique. Parmi les quatre instructions existantes, la passe de préchargement automatique de GCC ne génère que des intructions *prefetchNTA* et *prefetchT0*. On se propose donc d'utiliser les deux instructions restantes afin d'encoder le début et la fin des régions de données en lecture seule. Pour encoder l'information d'une zone de lecture seule, on a besoin de transmettre les paramètres de l'adresse de début et l'adresse de fin de l'intervalle. Les deux instructions possèdent un seul opérande : l'adresse de la donnée à précharger. L'opérande de *prefetchT1* permet d'encoder l'adresse de début et celui de *prefetchT2* l'adresse de fin de l'intervalle en lecture seule. Ainsi, tous les accès mémoires effectués dans cet intervalle seront ensuite identifiés comme étant des accès sur des zones de lecture seule sur les outils suivants de la chaine. Pour repasser une zone de lecture seule à une zone de lecture-écriture, il suffit ainsi de redéclarer l'intervalle de la même façon. Un exemple de code après instrumentation est présenté dans le listing 4.1.

Listing 4.1 – Instrumentation d'une zone de lecture détectée sur A

```
//Declaration d'une periode de lecture seule pour A
//On renseigne le debut et fin de l intervalle
builtin_prefetch(&A ,0,2);
builtin_prefetch(&(A+size),0,3);

#pragma omp parallel for ...
for (i = 0; i < n; i++) {
 for (j = 0; j < n; j++) {
  ... = f(A)
    }
}
//Redeclaration du meme intervalle
//i.e. fermeture de l intervalle
builtin_prefetch(&A ,0,2);
builtin_prefetch(&(A+size),0,3);
```

Listing 4.2 – Instrumentation d'une zone de lecture détectée sur A

```
max33_ui8matrix_omp._omp_fn.0 (struct .omp_data_s.0 * .omp_data_i)
{
  I0_10 = .omp_data_i_9(D)->I0;
  i0_11 = .omp_data_i_9(D)->i0;
  i1_12 = .omp_data_i_9(D)->i1;
  j_13 = .omp_data_i_9(D)->j0;
  j1_14 = .omp_data_i_9(D)->j1;

   <bb 1> :
  //Fonctions d'instrumentations
  _3 = // retrouve addr debut de zone
  __builtin_prefetch (_3, 0, 2);
  _11 = // retrouve addr fin de zone
  __builtin_prefetch (_11, 0, 1);

  _15 = __builtin_omp_get_num_threads ();
  _16 = __builtin_omp_get_thread_num ();

  <bb 3> : // Debut de la boucle

  <bb 27>: // Phase d'instrumentation identique au bb 1

        return;
```

Procéder de cette façon possède l'avantage de pouvoir rester au optimisateur de code pour la génération de code (juste deux appels de fonctions à générer pour ouvrir/fermer un intervalle) et également d'obtenir des applications en sortie de compilation toujours exécutables sur une architecture x86 classique. Ce mécanisme présente cependant l'inconvénient de devoir renseigner explicitement la taille de la zone en lecture seule accédée. Pour cela, une analyse inter-procédurale a été effectuée afin de détecter dans le programme, les différentes allocations mémoire dont la taille peut être retrouvée à la compilation, cela comprend les allocations statiques et les allocations dynamiques de taille connue à la compilation. Un exemple d'utilisation est montré à travers l'exemple de la figure 4.5. D'abord, par fonction, toutes les allocations sont collectées, puis l'analyse développée parcourt le graphe de flot de contrôle complet à partir des références à la zone en lecture seule pour retrouver l'instruction d'allocation de la zone mémoire. Si la taille de cette allocation peut être retrouvée à la compilation, la zone peut alors être instrumentée.

Cette passe ne peut donc instrumenter que les zones en lecture seule dont la taille est connue à la compilation. Plusieurs méthodes sont possibles afin d'enlever cette limitation. Par exemple, une autre approche possible pour déterminer la taille de la zone de mémoire accédée serait donc d'utiliser des analyses d'évolution des variables d'induction. Une variable d'induction est une variable qui varie de façon linéaire entre chaque itération et qui permet généralement d'indexer les régions mémoires. En effet, pour les applications étudiées et plus généralement pour les applications de traitement d'image, les tableaux sont accédés à partir d'une rela-

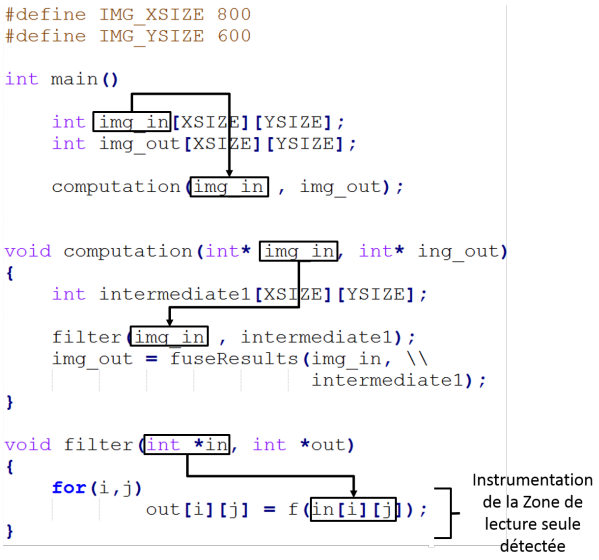


FIGURE 4.5 – Exemple de fonctionnement de l'analyse inter-procédurale. La taille de la région en lecture seule est retrouvée à partir des différents appels à cette fonction.

tion affine aux variables d'induction, de type *base* + *index* ∗ *pas* + *decalage*. Ainsi, à partir de l'analyse des bornes des variables d'induction, la zone de mémoire accédée peut être retrouvée de façon exacte ou approximée. Cette analyse des variables d'induction existe dans GCC mais notre analyse est ici limitée par le fait que le résultat de cette analyse n'est pas accessible via l'interface de plugin de GCC.

Cette limitation est inhérente au flot utilisé pour simuler la technique de cache collaboratif, elle ne correspond pas à une limite du compilateur. Cette contrainte s'avère peu limitante pour analyser les applications de la suite COTS ce qui correspond à l'objectif visé. Cependant, pour permettre la détection de données en lecture seule dynamiquement allouées dans le cas d'applications plus complexes, la proposition est renforcée à l'aide d'un pragma utilisateur permettant de compléter la détection. La sémantique du pragma est proche d'un pragma OpenMP. L'utilisateur peut alors donner une indication sur la présence d'une région de mémoire en lecture seule dans le bloc de calcul suivant (fonction ou boucle). L'adresse de base ainsi que la taille de la région mémoire doivent être indiquées. Cette indication permet de compléter l'analyse statique effectuée et permet surtout de donner l'information de taille de la région considérée, et selon les cas, cette information peut ne pas être prise en compte. L'information importante du pragma est surtout la taille de la région, car des transformations de code peuvent faire apparaitre ou disparaitre une zone qui apparait à l'utilisateur comme étant en lecture seule. L'analyse statique a donc une priorité plus importante que le pragma utilisateur sur la détection ou non des zones de lecture seule. Un exemple illustrant l'intérêt du pragma est développé avec le listing 4.3.

Listing 4.3 – Exemple d'utilisation du pragma permettant la détection de données en lecture seule

```
#define N 1024

int* a = alloc_array();
foo(a,b,c)
...
void foo(int* a){

        /* Declaration d'une zone de lecture seule sur a
         sur la boucle for */
        #pragma ro a N
        for( i = 0 ; i < N ; i++ ){
                out[i] = foo(a[i]);
        }
}
```

4.4 Conclusion

Nous avons développé dans ce chapitre une méthode de classification des données à la compilation décomposée en une phase de détection des données en lecture seule et une interface avec l'architecture permettant une détection à l'échelle du bloc de cache. Les analyses implémentées restent relativement simples comparées à des analyses de flots de données proposées à la compilation. Cependant, l'implémentation dans GCC montre des taux de détection des données en lecture seule importants ce qui met une évidence la pertinence d'une proposition de co-design compilateur architecture pour une gestion spécifique et automatique des données en lecture seule. Contrairement à l'approche de Li et al.[67], les taux de détection proposés avec une approche conservatives sont suffisants et il n'y a pas besoin de recourir à une détection plus souple qui autoriserait des erreurs de détections. Le compilateur garantit donc qu'aucune écriture ne passera par le chemin de données en lecture seule.

Dans la solution proposée, le compilateur place sur le chemin de donnée pour les données en lecture seule toutes les zones qu'il détecte. Une perspective peut être d'ajouter une étape entre la détection et l'instrumentation qui évalue l'intérêt de changer les données en lecture seule de chemin de données pour une prise de décision plus intelligente de la part du compilateur. Cela nécessite de la part du compilateur d'être capable de pouvoir modéliser le comportement en cache de ces données et d'évaluer le gain apporté à les changer de chemin de données. Des modèles de comportement des données en cache à la compilation qui peuvent servir de base théorique à une telle étude ont déjà été proposée dans la littérature.

L'interface proposée entre compilateur et architecture permet avec une granularité au niveau bloc de cache, de rendre consciente l'architecture des données

qu'elle manipule. Maintenant que cette classification est effectuée, il s'agit d'étudier les possibilités de cette gestion des données en lecture seule au sein d'une hiérarchie mémoire multi-coeurs.

Chapitre 5

Exploration architecturale pour une gestion spécifique des données en lecture seule

Sommaire

"Réfléchir, c'est fléchir deux fois."

Alain Damasio - La Zone du dehors

A travers les deux chapitres précédents, nous avons montré qu'une gestion séparée des données en lecture seule était pertinente et que cette classification des données pouvait être faite a priori à la compilation. Il s'agit donc maintenant d'étudier comment cette gestion peut être effectuée dans des hiérarchies mémoires pour systèmes parallèles embarqués. La méthode utilisée dans ce chapitre pour la gestion spécifique des données en lecture sera une exploration architecturale des différentes possibilités par simulation. Cette étude aurait pu également être envisagée à travers d'autres méthodes. Une méthode s'appuyant sur l'étude des distances de réutilisation ou plus généralement sur des métriques de localité peut être effectuée, cela reviendrait à améliorer l'analyse proposée dans le chapitre 3. Ces métriques proposées à la base pour des caches privés ont été améliorées pour

être capable de modéliser le comportement de toute la hiérarchie mémoire d'une architecture parallèle classique. De plus, elles présentent l'avantage d'abstraire au moins partiellement le matériel ce qui permet l'évaluation rapide d'un grand nombre de solutions architecturales différentes. Par exemple, la distance de réutilisation concurrente proposée par Jiang et al.[91] permet de rendre compte avec précision des phénomènes d'interférences lorsque plusieurs processeurs utilisent un même cache partagé. Toutes les combinaisons d'applications peuvent ainsi être évaluées beaucoup plus rapidement que par simulation. Cependant, ces métriques ne permettent pas de rendre compte de l'effet de certains paramètres d'implémentation importants plus complexes comme le protocole de cohérence par exemple.

5.1 Propositions d'architecture

Afin de proposer une gestion des données en lecture seule, on part d'une hiérarchie mémoire de référence représentée figure 5.1-a). Elle est constituée d'un niveau L1 privé séparé entre instructions et données et un niveau L2 unifié et partagé. Cela correspond à un modèle de hiérarchie classique que l'on retrouve dans la plupart des architectures INTEL et ARM, avec des variations sur le nombre de niveau dans la hiérarchie ou le niveau de partage des caches. La gestion des données en lecture seule se fera donc en parallèle d'une hiérarchie mémoire. La mémoire utilisée pour effectuer cette gestion spécifique, sera implémentée sous la forme d'un cache de technologie standard SRAM que nous appellerons par la suite, cache RO. Parmi les multiples paramètres à fixer pour le cache RO, on trouve :

- La propriété privé/partagé
- Le niveau de la hiérarchie auquel est ajouté le cache RO
- Le design du cache RO : taille, associativité, taille de bloc de cache, nombre de ports, ...
- Le mécanisme de cohérence vis-à-vis du reste de la hiérarchie mémoire
- La politique de remplacement

Ces paramètres, comme il a été montré en introduction 1.3.3, sont interdépendants et une solution optimale (quelques soit le critère choisi) ne peut pas étudier séparément chacun de ces critères. Afin de limiter la complexité de cette analyse, l'étude se propose d'abord de fixer deux propriétés importantes que sont la propriété de partage, ou non, du cache et le niveau auquel il est pertinent d'ajouter cette gestion. Cette étude se décline donc selon 3 possibilités tel que détaillées dans la figure 5.1. Le scénario A considère un cache spécifique privé au niveau L1, le scénario B un cache partagé au niveau L1, le scénario C un cache partagé au niveau L2.

La définition des données en lecture seule que nous utilisons est dynamique et autorise des changements de statuts au cours de l'exécution de l'application. Les données sont donc accédées par l'un des deux chemins de données selon le moment de l'exécution. Un problème de cohérence se pose entre deux caches

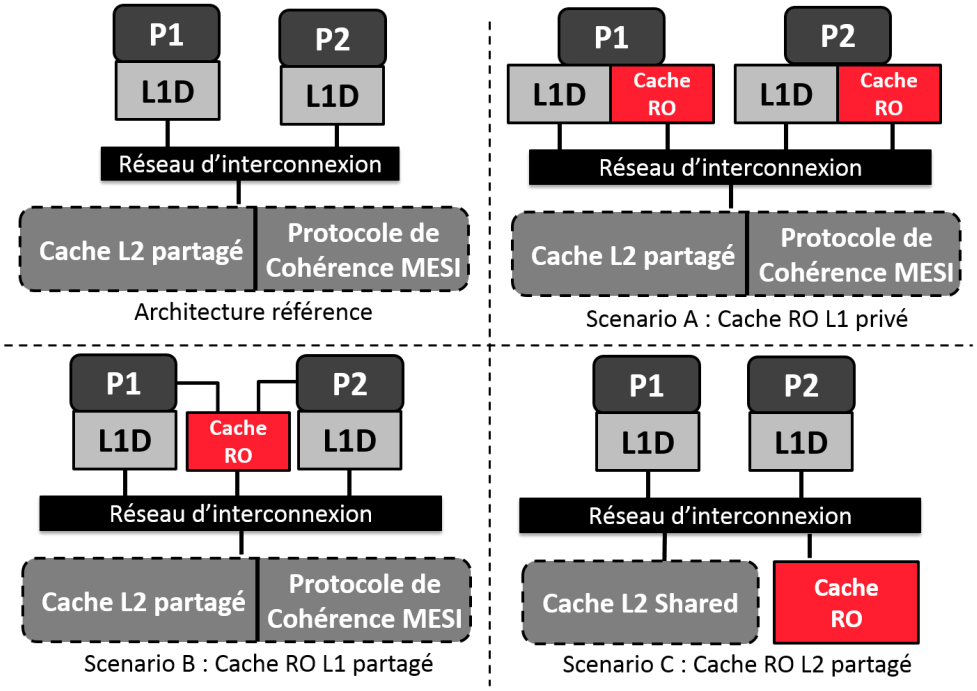


FIGURE 5.1 – Description des scénarios

de même niveau et des mécanismes doivent être alors utilisés afin d'invalider les blocs de caches entre lors des changements de statut des données. On décrit maintenant les différents scénarios et la solution utilisée pour résoudre ce problème de cohérence.

5.1.1 Scenario A

Le premier scenario considère un cache RO privé pour chaque processeur. Cette solution est similaire dans le principe à la gestion différenciée proposée entre instructions et données. Pour résoudre le problème de cohérence entre le cache L1D et le cache RO d'un même processeur, le cache L2 s'assure que le bloc de cache soit présent uniquement dans un des deux caches. La procédure illustrée figure 5.2, nécessite qu'une invalidation soit envoyée au cache L1D par le cache L2 s'il détecte un changement de statut du bloc de cache (ici lecture-écriture vers lecture seule) afin de garantir la bonne cohérence. Chaque entrée de la liste des processeurs partageant la donnée dans le cache L2 doit alors contenir un bit supplémentaire afin de stocker l'emplacement du bloc de cache (L1D ou RO) qui est déterminé selon la provenance du dernier accès à avoir accéder au bloc. Cette procédure est bloquante jusqu'à la réception de l'acquittement d'invalidation afin d'éviter les comportements indéfinis dû à l'arrivée de nouvelles requêtes sur le même bloc.

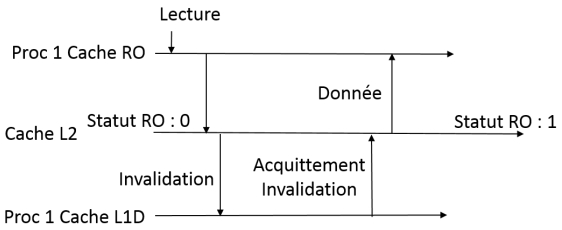


FIGURE 5.2 – Procédure de changement de statut d'un bloc de cache RO de lecture-écriture vers lecture seule dans le scénario A (le bloc est déjà présent dans le cache L1D)

5.1.2 Scenario B

Le scénario considère un cache RO partagé au niveau L1 entre les processeurs. Cette situation est similaire à une idée que l'on retrouve sur les architectures Kepler des processeurs graphiques NVIDIA [92] qui ajoute un cache en lecture seule au niveau L1 qui est partagé par un ensemble de 32 processeurs (un *Warp*). Le problème de cohérence entre les caches de données et le cache RO est résolu dans ce cas en utilisant pour le cache RO, la même machine d'état MESI qu'un cache de données classique. Du point de vue du protocole de cohérence, cela revient simplement à ajouter un cache de données supplémentaire. Le cache RO peut donc partager les blocs de caches en mode *Shared* (Partagé), avec les caches de données. Également, le cache RO ne possède qu'un seul port permettant de communiquer avec les processeurs. L'ajout de ports augmente de façon importante les couts d'accès au cache et doit donc être évité pour les caches proches processeurs. Les requêtes faites au cache RO dans ce cas-là sont donc séquentialisées dans une pile FIFO et le contrôleur du cache RO traite une requête par cycle. La latence d'accès au cache RO peut être plus élevée que pour un cache L1D du fait d'un conflit d'accès. De la même façon que les caches L1D, le cache RO est non bloquant, ce qui signifie que lorsqu'un défaut de cache se produit, une entrée est créée dans le registre MSHR (*Miss Status Holding Register*), avant de continuer à servir d'autres requêtes.

5.1.3 Scenario C

Le dernier scénario considère un cache RO partagé au niveau L2. Dans le cadre de ce scénario, le problème de cohérence entre le cache L2 et le cache RO est plus compliqué à résoudre du fait que les accès sont d'abord traités par les caches L1 qui filtrent les informations de classification. Le processus de changement de zone est décrit figure 5.3. Dans le cas d'un changement de statut, c'est à dire qu'il y a une différence entre le bit inséré dans la requête et le bit de statut dans le bloc de cache, le contrôleur L1 prévient le contrôleur L2 qui prévient les processeurs partageant ce bloc afin de mettre à jour l'information de statut. Ensuite, un changement du bloc de cache est effectué du cache L2 au cache RO (ou vice versa selon le changement de statut effectué). Cette opération n'invalide pas les blocs concernés du cache L1

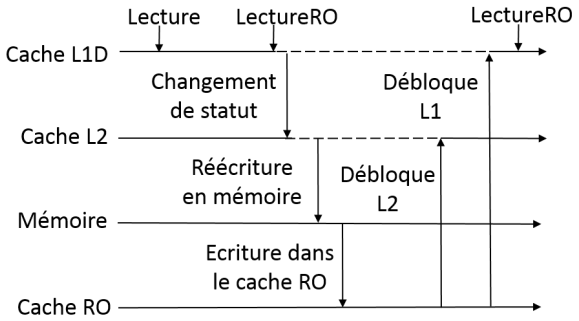


FIGURE 5.3 – Procédure de changement de statut d'un bloc de cache RO de lecture-écriture vers lecture seule dans le scénarion C(le bloc est déjà présent dans le cache L1D)

mais bloque la hiérarchie mémoire durant cette opération afin de garantir que la propriété d'inclusion du système est préservé. Le bloc ne pouvant être que présent que dans l'un des deux cache L2 ou RO.

Comme nous avons pu le voir, toutes les solutions présentent un surcout plus ou moins important du au changement de statut d'un bloc de données. La granularité de la détection à la compilation des données en lecture seule est supposée être suffisamment importante pour que ce surcout reste faible.

5.2 Architecture de simulation

Afin de simuler les architectures proposées, il s'agit de se munir d'une infrastructure de simulation afin des modéliser précisément les éléments importants du scénario. Ces derniers pour notre cas correspondent aux contrôleurs mémoires et les interactions sur le réseau d'interconnexion. Dans cette section, les simulateurs utilisés sont brièvement décrits, et les différents compromis effectués entre vitesse de simulation et précision sont étudiés pour comprendre le cadre de validité des résultats. Afin de choisir un simulateur adapté, on peut classer les simulateur selon leur niveau d'abstraction du matériel, une classification habituellement établie dans la littérature est proposée figure 5.4. Sans rentrer dans le détail des différents niveaux de simulation possible, nos expérience se situent à un niveau exploratoire, le but ici est d'effectuer une preuve de concept de l'intérêt de la gestion spécifique des données en lecture seule, avant de proposer une implémentation réelle. C'est également pour cela que les résultats présentés seront toujours exprimés en différentiels plutôt qu'en absolu.

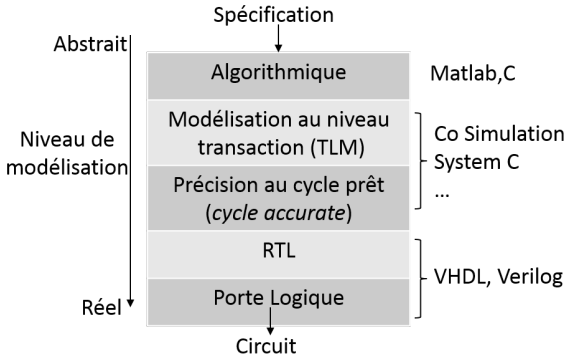


FIGURE 5.4 – Description des différents niveaux de simulations

5.2.1 Simulation fonctionnelle

Pour la simulation fonctionnelle de l'architecture, le simulateur Gem5 [93] est utilisé, il se situe au niveau *cycle accurate* de la figure 5.4. Issu de la fusion des simulateurs GEMS [94] et m5 [95], il propose une infrastructure de simulation open-source, qui supporte plusieurs jeu d'instructions et possède deux modes de simulations : le mode SE (*SystemCall Emulation*) et le mode FS (*Full system*). Le mode SC ne simule que la partie utilisateur, les appels systèmes sont émulés et permet de lancer des applications seules sans ordonnancement alors que le mode FS permet de démarrer un système complet avec un système d'exploitation, des interruptions matérielles, des exceptions, ... Gem5 inclut plusieurs simulateurs composés les uns aux autres permettant de simuler de façon précise les différentes parties de l'architecture. Afin de modéliser la hiérarchie mémoire et le protocole de cohérence, le simulateur gem5 [93] propose deux modèles : *Classique* ou *Ruby*. Le modèle *Classique* (issu de m5) fournit un modèle rapide et flexible à utiliser alors que *Ruby* (issu de GEMS) fournit une infrastructure plus précise et plus complète et permet en autre de décrire les contrôleurs de cache à l'aide d'un langage de domaine spécifique (DSL), le SLICC [1].

De plus, dans le simulateur Ruby, deux façons de modéliser le réseau d'interconnexion sont possibles, le modèle dit *Simple* et le modèle *Garnet* [96]. Le modèle *Simple* considère la latence des routeurs et des liens ainsi que la bande passante des liens. Le routeur est alors modélisé comme un entonnoir entre les buffers d'entrées et sorties avec une vérification sur les bandes passantes avant d'envoyer un message. Il ne modélise pas les interactions complexes qui ont lieu à l'intérieur du pipeline d'exécution d'un routeur comme par exemple la concurrence de ressources ou l'attribution des canaux. Ce modèle sert à des expérimentations qui requièrent le modèle détaillé des protocoles de cohérence de *Ruby* mais simplifie le fonctionnement du routeur afin de simuler plus rapidement. Le modèle *Garnet* [96],

1. Specification Language for Implementing Cache Coherence

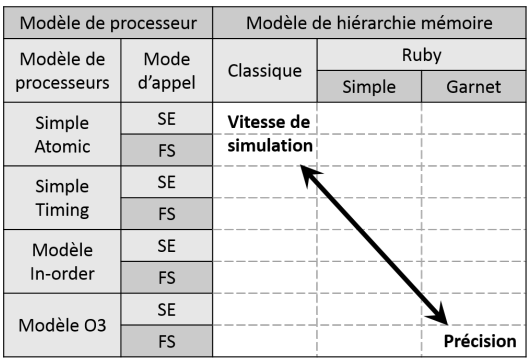


FIGURE 5.5 – Possibilités de simulation sous Gem5 pour un compromis Précision/-Vitesse de simulation

quant à lui, simule le pipeline d'exécution classique à 5 étages de chaque routeur. Les problématiques d'allocations des ressources ou l'attribution de canaux virtuels dans le routeur peuvent alors être étudiées. De plus, chaque paquet transmis sur le réseau est découpée en petites parties élémentaires appelées *flits*, le paquet complet étant reconstruit par le destinataire. Chaque *flit* est géré distinctement par le modèle. Le modèle *Garnet* permet de faire des explorations détaillées des réseaux d'interconnexion mais rajoute environ 21% de temps de simulation par rapport au modèle *Simple* [96].

Les compromis précision et vitesse de simulation sont représentés sur la figure 5.5. Au vu de nos besoins en terme de précisions, les simulations effectués se font donc en mode SE avec *Ruby* pour la simulation de la hiérarchie mémoire car une description précise des interactions mémoires est nécessaire. Cependant, pour le réseau d'interconnexion, le modèle *Simple* sera utilisé car le réseau intervient ici comme support de communication pour les requêtes et n'est pas fondamentalement différent entre scénarios à part l'ajout de quelques ressources nécessaire pour relier le cache RO au reste du réseau. Le type de processeur ayant une importance moindre dans nos simulations, le modèle *Simple Timing* sera utilisé pour nos simulations.

5.2.2 Modèlisation de la consommation

Pour la partie de modélisation de la consommation énergétique, le simulateur McPat 1.3 [97] est utilisé. Il permet la modélisation de la surface et de la consommation d'énergie statique et dynamique des différents composants de la puce : processeur, mémoires, réseau d'interconnexion ... Le modèle de transistor utilisé dans McPat se base sur des paramètres technologiques fournis par MASTAR [98], logiciel développé par ST Electronics dans le cadre de l'ITRS [2]. Pour la modélisation

2. *International Technology Roadmap for Semiconductors* : Ensemble de documents rédigés par un regroupement d'experts industriels internationaux dans le domaine des semi-conducteurs permettant

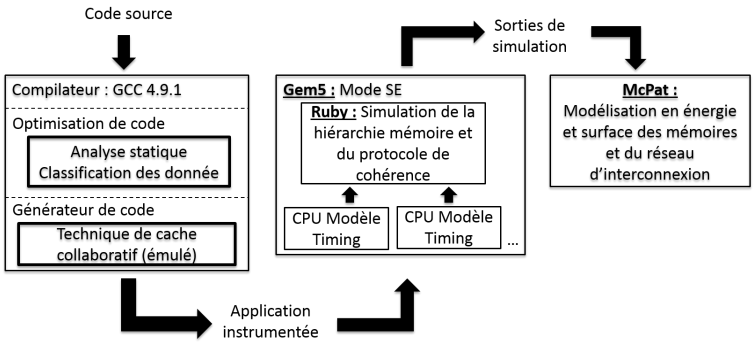


FIGURE 5.6 – Architecture de simulation des différents scénarios

des mémoires, McPat utilise une version intégrée du simulateur CACTI 6.5 [35]. Ce simulateur a été comparée à des architectures réelles et présentent une erreur de l'ordre de 20%[97].

On obtient donc l'architecture de simulation de la figure 5.6 avec Gem5 en simulateur fonctionnel et McPat pour les modèles de consommation.

5.3 Résultats

5.3.1 Paramètres de simulations

Les scénarios simulés utilisent un cache supplémentaire par rapport à l'architecture de référence. Afin de produire des résultats comparables, il s'agit de conserver le stockage constant. Pour chaque simulation, le stockage nécessaire pour le cache RO est attribué en retirant des voies au cache L2, le détail des tailles des caches L2 et RO sont affichés table 5.2. Dans ce cas, les simulations ne sont pas exactement constantes en terme de surface, mais plutôt en termes de stockage, car la surface nécessaire au contrôleur du cache RO est négligé. Le cache L2 est un cache de 256Ko à 8 voies, soit 32Ko par voies et donc on ne peut supprimer le stockage L2 que par voie de 32Ko, c'est pourquoi le cas du scénario B avec un cache RO de 16Ko considère un cache L2 de 224Ko plutôt que de 240Ko.

Les simulations s'effectuent sur une architecture à 4 coeurs. Du fait d'utiliser le mode SE de Gem5, il n'y a pas d'ordonnanceur, le nombre de threads est égal au nombre de processeur et la correspondance est statique entre les threads et les processeurs, le thread n°i s'exécute sur le processeur n°i, et il n'y a donc pas de changement de contexte au cours de l'application. On utilise les applications de la suite COTS parallélisées avec OpenMP et compilée en O3 avec GCC 4.9.1 et le plugin GCC présenté dans le chapitre 4 qui effectue la détection des données en

l'évaluation des technologies actuelles et les évolutions technologiques du domaine

TABLE 5.1 – Paramètres constants durant les simulations

	Paramètres de simulation
Processeurs	4 processeurs, Modèle *Simple Timing*, 1GHz 1 unité de lecture/écriture
Cache L1-D	32Ko, 2 voies, bloc de 64 octets
Réseau d'interconnexion	Bus de 64 octets, latence de 2 cycles
DRAM	512MO, un contrôleur mémoire at 12.8GO/s

TABLE 5.2 – Répartition des ressources en cache entre cache L2 et cache RO pour chaque scénario

	Référence	A	A	B	B	C	C
Cache RO	- -	8Ko 2 voies	16Ko 2 voies	16Ko 2 voies	32Ko 2 voies	64Ko 2 voies	128Ko 4 voies
Cache L2	256Ko 8 voies	224Ko 7 voies	192Ko 6 voies	224Ko 7 voies	224Ko 7 voies	192Ko 6 voies	128Ko 4 voies

lecture seule.

La performance de la simulation est observée à travers la métrique de l'AMAT [3] qui correspond au temps moyen que met la hiérarchie mémoire à répondre à une requête mémoire. Cette métrique permet de mesurer plus précisément les variations de performance du système, car les autres opérations impactant la performance du système (calcul, prédiction de branchement, ...) sont à priori constantes entre les différents scénarios. Cependant, la correspondance n'est pas directe entre la variation des performances et la variation de l'AMAT, cela dépend de la sensibilité des applications à la latence des accès mémoires. La figure 5.7 montre les résultats du point du vue de la consommation, de l'AMAT des différents scénarios. La variation du nombre de défauts de caches au niveau L1 est également représentée pour les scénarios A et B, ce nombre inclut les défauts effectués sur le cache RO, étant logiquement au niveau L1. Nous analysons maintenant les résultats pour les différents scénarios.

5.3.2 Scenario C

Le scénario C apporte peu de gain en réduction d'énergie et aucun gain en performance. De façon générale, les caches L2 utilisent un flux mémoire avec moins de localité des données et présentent des taux de défauts de caches plus important qu'un niveau L1. Séparer le flux mémoire au niveau L2 réduit le peu de réutilisation qu'un cache L2 unifié peut proposer. On n'observe donc aucune dépollution au niveau des caches. De plus, la séparation a pour effet une augmentation importante

3. *Average Memory Access Latency* : latence moyenne d'accès à la mémoire. Idéalement cette latence est la plus proche proche de la latence du cache L1

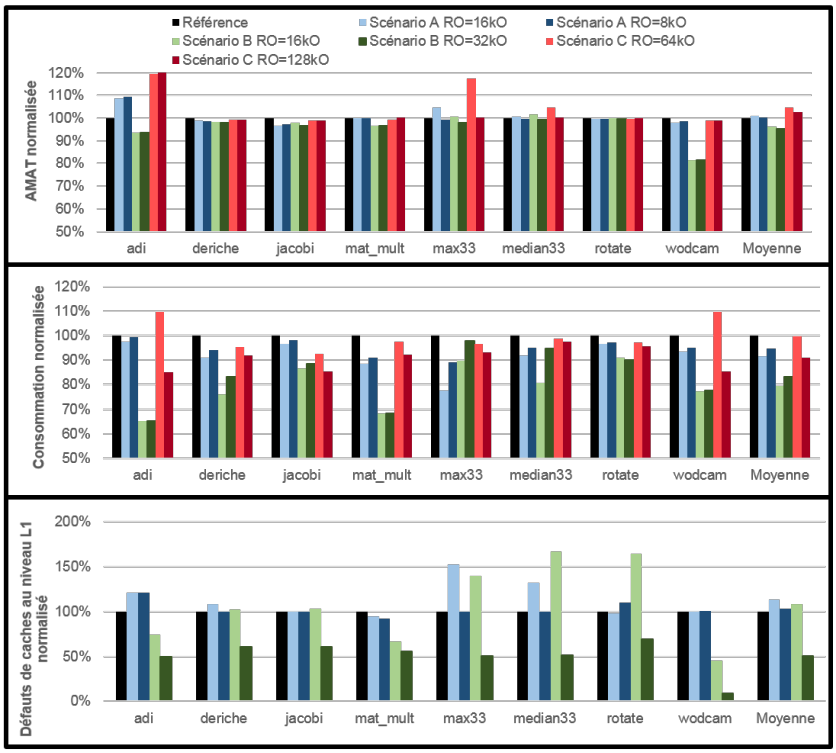


FIGURE 5.7 – a) Variation de l'AMAT normalisé et b) variation de la consommation d'énergie normalisée pour tous les scénarios. c)Variation du nombre de défauts de caches au niveau L1 pour les scénarios A and B

des accès à la mémoire principale, +46% avec un cache L2 RO de 128Ko et 70% pour un cache L2 RO de 64Ko. Pour ce scénario, la pénalité de changement de statut est très importante car elle oblige à repasser par la mémoire et le bloc de donnée n'est pas assez réutilisé au niveau des caches L2 pour que ce soit rentable. Une proposition d'amélioration de la solution peut être d'utiliser une technique de partitionnement de cache tel que celui proposé par Khan et al. [99]. De plus, cette solution peut être pertinente pour des applications présentant plus de réutilisation au niveau du cache L2.

5.3.3 Scenario A

Dans le scénario A, le nombre de défauts de caches reste relativement constant par rapport à l'architecture référence alors que dans le même temps, le taux de défauts sur les caches de données diminue en moyenne de 17.56% sur les caches L1D. La figure 5.8 représente le taux de défauts du cache L1D du processeur 0 [4]. Avec la plupart des applications, on observe bien un taux de défaut plus faible sur le cache L1D synonyme de dépollution. Chaque cache RO a peu d'accès à traiter, les accès sur les données en lecture seule représentent 28,5% du nombre d'accès en moyenne répartis sur les 4 caches RO, mais provoquent un nombre important de défauts de cache. Mais ici, les accès sur les données en lecture seule qui créaient des défauts de cache sur le chemin de données sont simplement reportés sur le chemin des données en lecture seule, sans que la séparation proposée ne permette de résoudre ces défauts de cache. Les résultats du chapitre 3 montrent en effet, que les distances de réutilisation des données en lecture seule étant en moyenne supérieure aux autres données, il faut donc proposer un cache RO de taille plus importante pour pouvoir résoudre les défauts crées par ces accès. On peut cependant observer un gain sur l'énergie du au fait que les accès en lecture seule sont gérés dans le cache RO, plus petit qu'un cache de données et propose donc un cout par accès plus faible.

5.3.4 Scenario B

Un cache partagé permet, de façon générale, une meilleure utilisation de la ressource en cache, mais peut également conduire des dégradations de performances en cas de conflits d'accès à la ressource. De plus, l'utilisation d'un cache partagé entraine l'apparition d'un phénomène qui aura une importance ici : les interférences constructives ou destructives. Ce phénomène est détaillé avec la figure 5.9. Le fait d'utiliser un cache partagé modifie les distances de réutilisation des accès et cette interférence est dite destructive lorsque le fait de partager le cache entre plusieurs processeurs entraine une augmentation des défauts de cache dû au fait qu'un processeur écrase un bloc de cache utilisé par un autre processeur. Au contrainte, elle

4. Le taux de défauts entre les différents caches L1D du système sont relativement similaires car la partie séquentielle de l'application est négligeable et que les processeurs tous les mêmes sections parallèles.

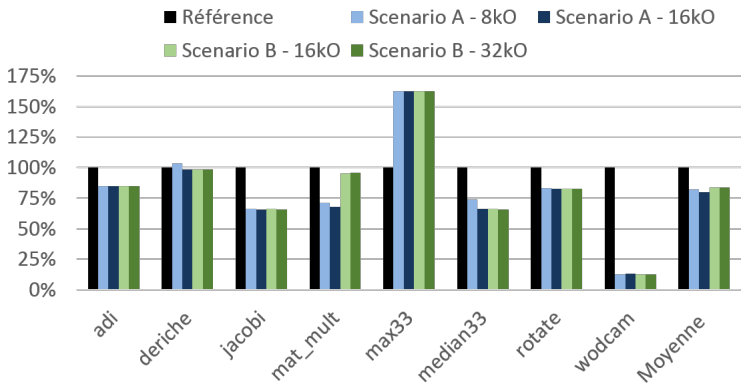


FIGURE 5.8 – Variation du taux de défaut de cache sur le cache L1D en fonction des scénarios

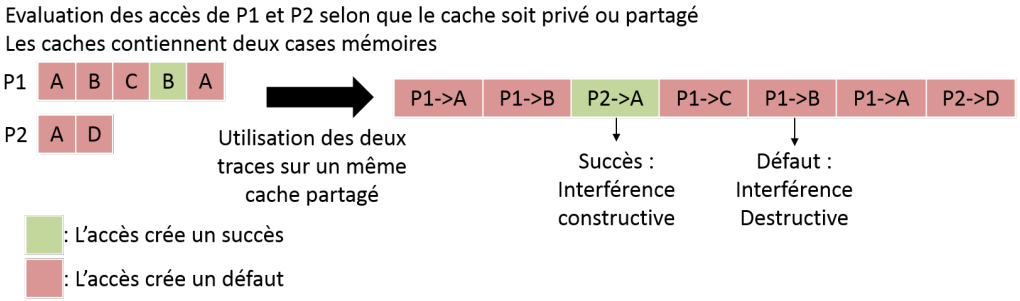


FIGURE 5.9 – Phénomène d'interférence dans un cache partagé. Le fait de combiner les accès de P1 et P2 modifient les performances du cache.

est dite constructives lorsque, cela permet une meilleure réutilisation des blocs de données dans le cache dû au fait que plusieurs processeurs peuvent travailler sur le même bloc de cache ce qui réduit leurs distances de réutilisation.

Le scénario B présente les meilleurs résultats que ce soit en termes de consommation, réduction du nombre de défauts de cache ou même le temps de réponse. L'avantage de ce scénario par rapport au scénario A est double. Pour ce dernier, on avait conclu qu'une part des défauts de cache provenait de certains accès fait sur les données en lecture seule qui étaient simplement reportés sur le cache RO. La ressource en cache RO dans le scénario A est donc nécessairement gaspillées du au fait qu'elle ne peut pas résoudre ces défauts de cache. La mutualisation de la ressource en cache RO du scénario B permet de limiter ce gaspillage de ressources et également l'impact de la réduction du cache L2 sur le système. Le deuxième facteur d'amélioration est le fait que d'autoriser le partage des données en lecture seule entre processeur au niveau L1 ce qui permet de produire des interférences constructives entre processeurs dans le cache RO et ainsi résoudre certains de ces défauts de cache.

L'augmentation des défauts de cache pour *rotate*, *max33* et *median33* s'explique par le fait que ces applications ont une utilisation du cache L1D déjà très efficace avec un taux de défauts inférieur à 1%. Une part importante des défauts de cache correspond à des défauts obligatoire et le fait d'ajouter un cache en lecture seule ne permet pas de résoudre ces défauts. Cependant, l'impact en termes de consommation de ce phénomène reste largement compensé par le fait d'avoir un cout par accès plus faible pour le cache RO sur ces applications. Les applications *adi*, *mat_multiply* et *wodcam* ont une utilisation différente du cache RO du fait qu'une part importante des données accédées via le cache RO sont partagées entre plusieurs processeurs ce qui permet une réduction importante du nombre de défauts de cache même pour un cache RO de 16Ko. Ce type de comportement de comportement bénéficie de façon importante d'un tel scénario.

Pour la suite d'applications considérées, le scénario B avec un cache de 16Ko donne le meilleur résultat en termes de consommation d'énergie, notre critère principal. Cette solution est donc détaillée plus précisément avec des simulations complémentaires.

5.4 Simulations complémentaires

5.4.1 Impact de la latence du cache RO

Dans les simulations précédentes, la latence d'accès au cache RO est la même que celle d'un cache L1D à savoir 3 cycles. Dans une implémentation plus proche de la réalité, cette latence pourrait être plus importante du fait que le cache RO doit être partagé entre tous les coeurs. De plus, la latence de ce cache est plus critique que celle d'un cache de donnée privé parce qu'il peut potentiellement bloquer plusieurs processeurs en même temps. C'est pourquoi, une expérience est menée dans laquelle on fait varier la latence d'accès au cache RO de 3 à 12 cycles (12 cycles étant la latence d'accès au cache L2).

Pour chaque application, la dégradation de l'AMAT se modélise selon une variation linéaire qui dépend de l'application. En moyenne, cette dégradation est de 1,5% par cycle supplémentaire nécessaire pour accéder au cache RO. Ainsi, pour une latence de 5 cycles, l'AMAT du scénario B correspond à l'AMAT de l'architecture de référence et la barrière de dégradation de 5% est franchie pour une latence de 9 cycles. On peut observer que bien que la variation soit affine pour toutes les applications, elle n'influence pas de la même façon chaque application. Cela permet d'observer la sensibilité de chaque application au chemin de données en lecture seule. La variation de cette sensibilité s'explique avant tout par le nombre d'accès détecté en lecture seule, par exemple *max33* possède une sensibilité plus importante que *rotate* du fait que le cache RO traite plus d'accès (68% pour *max33* et 10,8% pour *rotate*). Ces résultats montrent qu'une certaine flexibilité est possible

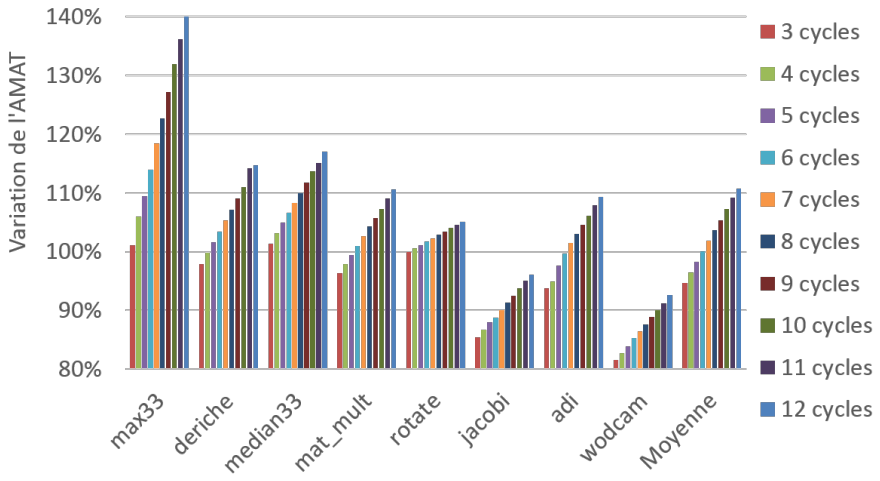


FIGURE 5.10 – Impact sur l'AMAT de la variation de la latence du cache RO. (l'AMAT est normalisé selon l'AMAT de l'architecture référence)

sur la latence du cache RO et les dégradations de performances ne sont observées que pour des scénarios très pessimistes.

5.4.2 Passage à l'échelle de la solution

Une autre expérimentation a également été menée pour étudier l'impact du nombre de coeurs sur le scénario B. Il est attendu que l'AMAT du système augmente avec le nombre de coeurs car dans cette situation, la gestion de la cohérence devient de plus en plus complexe et nécessite d'envoyer plus de messages de contrôles ce qui augmente le temps de réponse. La figure 5.11 représente la variation de l'AMAT de l'architecture de référence et de notre solution. Les différents AMAT sont normalisés par rapport à l'AMAT du scénario de référence à un coeur.

On peut observer ce phénomène sur l'architecture de référence, ce phénomène est moins important avec un cache RO permettant ainsi un meilleur passage à l'échelle de telles architectures, cet effet étant observé dans des proportions similaires pour chacune des applications. Pour un coeur, on constate que l'AMAT est plus élevé pour le cache RO, ce résultat est dû aux phénomènes de changement de zones qui demeurent importants. Cette conclusion est cependant limitée pour deux raisons. Premièrement, le nombre de coeurs testés reste faible au regard des challenges de passage à l'échelle adressés aujourd'hui dans les architectures. Par exemple, la solution proposée par Kaxiras et al.[100] travaille sur le passage à l'échelle de solutions de plusieurs centaines voire plusieurs milliers de coeurs. L'étude sur le nombre de processeurs est ici limitée par notre suite d'applications

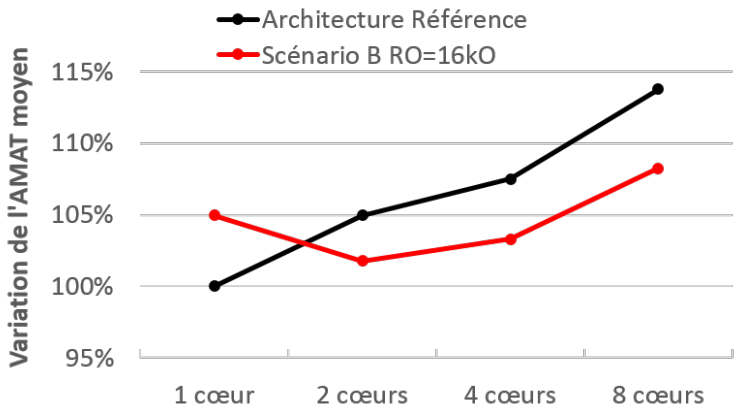


FIGURE 5.11 – Variation de l'AMAT

testées qui présentent un niveau de parallélisme peu élevé. Deuxièmement, à partir d'un certain nombre de processeurs, on s'attend à observer des pénalités en termes de performances sur le cache RO, parce qu'il devra gérer un nombre plus important de requête de façon simultanée et des interférences destructives peuvent diminuer son efficacité. Il y a donc une limite dans le nombre de coeurs que la solution du scénario B peut accepter, on peut cependant conclure que ces effets restent négligeables pour 8 coeurs.

5.5 Conclusion

Dans ce chapitre, nous avons étudié différentes possibilités de gestion des données en lecture seule à partir d'une hiérarchie mémoire classiquement utilisée dans des systèmes embarquées. Une exploration architecturale des diverses solutions montre qu'une gestion spécifique des données en lecture seule à partir d'un cache RO partagé au niveau L1 apporte une réduction de la consommation d'énergie de 20,6% et une réduction du temps de réponse de la hiérarchie mémoire de 6%. Ces résultats permettent de montrer l'intérêt d'une gestion particulière des données en lecture seule. Mes résultats et la proposition finale sont dépendantes des applications testées. Cependant, cette méthodologie peut être appliquée à d'autres suites d'applications de la même façon. La gestion spécifique apporte non seulement déjà des gains significatifs en termes de performance mais ouvre des possibilités sur des optimisations que l'on peut appliquer au cache RO. L'idée est donc de s'appuyer sur la proposition architecturale du scénario B pour améliorer les résultats obtenus en étudiant diverses possibilités d'optimisations spécifique au cache RO.

Chapitre 6

Evaluation de différentes optimisations spécifique au cache RO

Sommaire

"La connaissance scientifique possède en quelque sorte des propriétés fractales : nous aurons beau accroître notre savoir, le reste, si infime soit-il, sera toujours aussi infiniment complexe que l'ensemble de départ."

Isaac Asimov

A partir de la proposition architecturale effectuée dans le chapitre précédent du scénario B avec un cache RO de 16kO. Il s'agit d'un cache spécifique pour gérer les données en lecture seule situé au niveau L1, partagé entre tous les processeurs. Le fait d'ajouter plus de caches dans une hiérarchie mémoire augmente la complexité de la solution en termes de design mais permet également des degrés de liberté supplémentaires pour des optimisations spécifiques au cache RO. En effet, jusqu'ici, nous avons considéré un cache RO identique à un cache de données. Ce chapitre explore plusieurs piste d'optimisation spécifique au cache RO selon 3 axes : la gestion de la cohérence, la politique de gestion du cache et l'utilisation de technologie NVM.

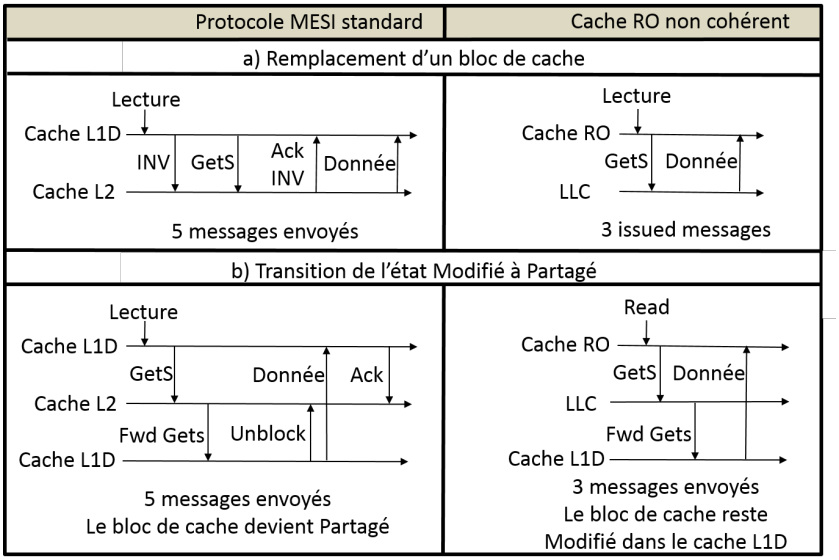


FIGURE 6.1 – Exemples de différence de traitement entre un cache RO cohérent et un cache RO non cohérent

6.1 Réduction de l'impact de la cohérence du système

6.1.1 Vers un cache RO non cohérent

Une première source d'optimisation appliquée au cache RO proposé est proposée par le fait que l'analyse statique effectuée à la compilation permet de garantir que le cache RO ne traite que des accès en lecture. Le suivi des données issues du cache RO n'est donc pas nécessaire d'un point de vue du protocole de cohérence MESI du système de la même façon que pour les caches d'instruction. Ce type d'optimisation renvoie à une littérature importante[64, 19, 70, 21, 20] étudiée 2.2 qui propose de réduire l'impact des protocoles de cohérence en activant ces mécanismes couteux uniquement pour les données partagées en écriture. Ce type de travaux d'abord effectué pour les données privées a été étendu pour les données en lecture seule. Une approche avec un cache RO constitue donc une approche orthogonale, ou les données considérées aussi bien que les objectifs diffèrent quelques peu. D'abord, nous considérons que

La machine d'état des contrôleurs des caches L2 et RO ont été modifiés afin d'éviter le suivi des blocs de cache en lecture seule de façon similaire à ce qui existe pour les caches d'instruction. Afin de comprendre l'intérêt d'une telle optimisation, deux exemples sont montrés figure 6.1. Dans le premier cas, une requête de lecture déclenche un défaut de capacité du cache RO. Avec un cache de donnée classique, cette invalidation requiert un échange de message d'auto invalidation avec acquittement doit être effectué avec le cache L2 même si le bloc n'est pas modifié. Cette contrainte est nécessaire afin que le cache L2 mette à jour la liste des processeurs partageant le bloc et garantir la propriété inclusive du système.

Cette précaution n'est pas nécessaire avec le cache RO du fait que le bloc de cache n'est jamais modifié, ce qui conduit à deux avantages. Premièrement, cette opération requiert moins de messages car le bloc n'aura jamais besoin d'être réécrit en cache L2. Deuxièmement, le système n'a plus besoin d'être inclusif car des blocs de cache peuvent être évincés du cache L2 tout en restant dans le cache RO. Cette contrainte d'inclusivité est une contrainte forte dans les hiérarchies mémoires car une part importante de l'espace mémoire est ainsi occupée par des duplications de bloc de cache, assouplir cette contrainte peut permettre d'augmenter l'efficacité des ressources en caches.

Le deuxième exemple de la figure 6.1-b illustre un cas de transition d'un bloc de cache Modifié sur un processeur à un état Partagé sur plusieurs processeurs. Dans le cas classique, un bloc de cache ne peut être partagé entre plusieurs processeurs qu'avec les droits de lecture afin de préserver la cohérence du système. Dans le cas d'un partage d'un bloc entre le cache RO un cache L1D, ce dernier peut transmettre la version conserver les droits en lecture-écriture, une telle transaction nécessite moins de messages et le cache L1D possédant le bloc conserve les droits en lecture-écriture, ce qui permet de ne pas créer de défaut de cache en cas d'écriture future sur ce bloc. D'autres exemples de transactions également être trouvé ou le cache RO apporte une gestion plus simple des blocs du fait que l'on peut anticiper que ce cache ne modifiera pas ces derniers. Si le cache RO n'a pas besoin d'être suivi par la cohérence du système comme le montre les exemples ci-dessus, il y a tout de même une gestion à conserver entre les caches de données et le cache RO lors des changements de statuts des blocs. Le passage de lecture-seule vers lecture-écriture est transparent car le cache L2 possède toujours la bonne version, pour la transition inverse, il y a un risque que le bloc soit modifié dans le cache L1D sans que le cache RO soit informé, c'est pourquoi les blocs de caches dans le cache RO sont systématiquement invalidés après chaque fin de zone de lecture seule (ce qui n'implique aucun envoi de message, cf exemple 6.1-a) à l'aide d'une instruction spécifique générée à la compilation.

Ce type d'optimisation facilite surtout la communication entre les mémoires du fait que moins de contrôle est nécessaire. Cette optimisation permet la simplification des communications entre mémoires qui nécessitent moins de messages mais la latence des transactions n'est cependant pas directement améliorée. En effet, dans l'exemple développé 6.1-a, les deux transactions ont la même latence et le processeur est stoppé durant le même nombre de cycles. L'effet sur la latence pourra plutôt être observé en termes de pression sur la bande passante du système qui sera alors moins forte, des gains de latence peuvent alors être observé si l'application est sensible à la bande passante.

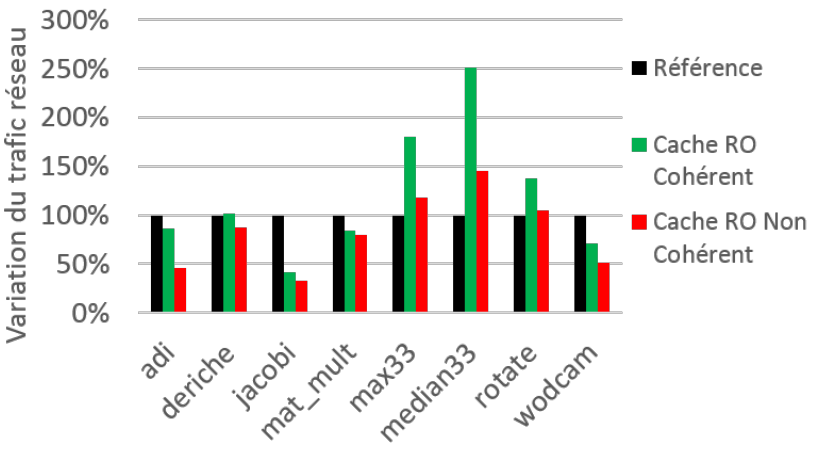


FIGURE 6.2 – Variation du nombre de messages envoyés sur le réseau entre un cache RO cohérent et un cache RO incohérent

6.1.2 Simulations et résultats

Cette variation du protocole MESI qui prend en compte le cache RO non cohérent a été implémenté dans Gem5 et le processus expérimental ainsi que les différents paramètres de simulation sont repris du chapitre précédent, détaillé 5.3.1. La figure 6.2 présente les résultats en terme de messages envoyés sur le réseau. Le fait d'adopter un cache RO cohérent peut créer une augmentation importante du nombre de messages, notamment pour les applications *max33* et *median33*, dû à l'augmentation du nombre de défauts de caches déjà commenté précédemment. Adopter un cache RO non-cohérent diminue donc le nombre de message envoyés sur toutes les applications en moyenne de 27%. Ce résultat permet d'améliorer légèrement le gain en consommation d'énergie déjà obtenu précédemment dû à une consommation du réseau plus faible sans toucher aux performances.

Cependant, cette optimisation n'améliore pas l'efficacité des caches dans la majorité des applications. La situation décrite dans le cas de figure 6.1-b) ou le cache RO reçoit le bloc de cache d'un autre cache L1 plutôt que du cache L2, existe surtout avec l'application *adi*. En effet, avec cette application, le cache RO reçoit en effet 6 fois plus de blocs de la part des caches L1D que de la part du cache L2, car les blocs de cache en lecture seule de cette application sont fortement partagées. L'optimisation d'un cache RO non cohérent dans cette situation permet alors d'améliorer de façon importante l'efficacité des caches. Le taux de défauts du cache RO passe alors de 11,1% à 1,7%. Le gain observé sur l'AMAT est de 22% et la consommation d'énergie est réduite de 11,3% par rapport à une solution avec un cache RO cohérent.

6.2 Politique de gestion spécifique au cache RO

6.2.1 Vers une politique de gestion spécifique au cache RO

Jusqu'à présent, nous avons envisagé qu'une politique LRU [1] pour le cache RO. Il s'agit de la politique traditionnellement implémentée sur tous les caches de la hiérarchie. Afin de réduire le stockage nécessaire d'information que requiert cette politique, on trouve la NRU qui fonctionne de la même façon que la LRU mais avec moins de bits sont utilisés pour encoder la localité. Ce type de politique repose sur le fait que les applications ont une localité temporelle et spatiale importante ce qui est moins vrai pour les caches plus éloignés de la hiérarchie dû au fait que les propriétés de localité sont alors filtrées par les caches proches processeurs. C'est pourquoi dans l'état de l'art actuel, des propositions de politiques de caches ont été faites pour les caches de dernier niveau, qui se basent sur des hypothèses moins fortes en termes de localité des données. Dans ce contexte, des solutions importantes sont la politique DIP [2] proposées par Qureshi et al. [101] et la politique RRIP [3] proposée par Jaleel [102] dont les algorithmes sont présentés dans le tableau 6.1. Contrairement à la politique LRU, la politique DIP place chaque bloc lors de son insertion en position LRU et il doit être accédé une seconde fois pour passer en MRU. La politique RRIP, quant à elle, se base sur une prédiction de l'intervalle de réutilisation. Ces solutions ont des propriétés intéressantes car elles permettent de se prémunir contre certains motifs d'accès comme les accès dits *streaming* [4], qui créent des cas pathologiques d'utilisation des caches.

Un changement de la politique de gestion de cache pour le cache RO peut être étudié pour deux raisons. Premièrement, comme nous l'avons montré dans le chapitre 3, les données en lecture seule présentent ont des propriété de localité spatiale et temporelle nettement moins fortes que les autres données et on peut vérifier si une politique LRU est toujours pertinente pour le cache RO. Deuxièmement, les résultats obtenus jusqu'à présent, montrent que la réutilisation des données dans le cache RO est un facteur important de bonne utilisation de la solution. Des politiques de caches spécifiques peuvent donc être envisagées afin d'augmenter la durée de vie des blocs partagés dans le cache RO.

6.2.2 Simulations et résultats

Les différentes politiques testées sont comparées à la politique de gestion de Belady [103] définie comme optimale. Cet algorithme évince le bloc de cache qui ne sera pas utilisée pour la plus grande période de temps. Cette politique reste purement théorique mais qui constitue néanmoins un moyen afin de mesurer

1. Politique *Least Recently Used* : le bloc évincé est celui est qui a été utilisé en dernier qui est dit en position LRU, le bloc accédé le plus récemment est dit en position MRU *Most Recently Used*
2. *Dynamic Insertion Policy*
3. *Re-Reference Interval Prediction*
4. Succession d'accès mémoires qui ne présente aucune réutilisation

TABLE 6.1 – Algorithmes des politiques de remplacement évaluées

LRU	**NRU**
Succès : (i) Le bloc est placé en MRU Défaut : (i) Remplace le bloc LRU (ii) Place le nouveau bloc en MRU	Succès : (i) Le bit NRU du bloc est mis à '0' Défaut : (i) Chercher le premier '1' (ii) Si trouvé, aller en (v) (iii) Mettre tous les bits NRU à '1' (iv) aller en (ii) (v) Remplacement du bloc et mise à '1' du NRU bit
DIP[101]	**RRIP**[102]
Succès : (i) Le bloc est placé en MRU Défaut : (i) Remplace le bloc LRU (ii) Place le nouveau bloc en LRU	Succès : (i) Met le RRVP du bloc à '0' Défaut : (i) Recherche du premier '3' (ii) Si un '3' est trouvé, aller au (v) (iii) Incrément de tous les RRPV (iv) Aller au (i) (v) Remplace le bloc et mettre RRVP à 2

l'efficacité d'un algorithme de remplacement en fournissant une référence. Les différentes politiques décrites dans le tableau 6.1 ont été implémentées et évaluées sous Gem5 avec un protocole expérimental identique à celui proposé dans le chapitre précédent 5.3.1 pour 4 coeurs. Les résultats illustrés figure 6.2.2 montrent qu'une politique LRU n'est pas toujours la meilleure solution et sur l'ensemble des politiques testées, c'est la politique NRU est celle qui se comporte le mieux en plus d'être celle qui requiert le moins de stockage car juste un 1 bit supplémentaire est nécessaire par bloc. La réduction moyenne du taux de défauts sur le cache RO est de 2,2% pour la politique NRU et de 2,1% pour la politique RRIP par rapport à une politique LRU. Cette modeste amélioration du cache RO permet d'améliorer les gains déjà obtenus en consommation et performance de la solution architecturale présentée dans le chapitre précédent.

6.2.3 Discussion

On trouve également dans la littérature des techniques permettant de conserver plus longtemps les blocs de caches partagés. Natarajan et al.[104] mettent en évidence l'intérêt de prendre en compte le partage dans l'algorithme de remplacement mais ne parviennent pas à une implémentation réaliste d'une telle solution dû à la difficulté de la détection de ces blocs partagées. Chen et al. [105] proposent un algorithme de partitionnement du cache de dernier niveau qui effectue l'attribution des voies du cache selon que le bloc est privé ou partagé. Une place plus importante est

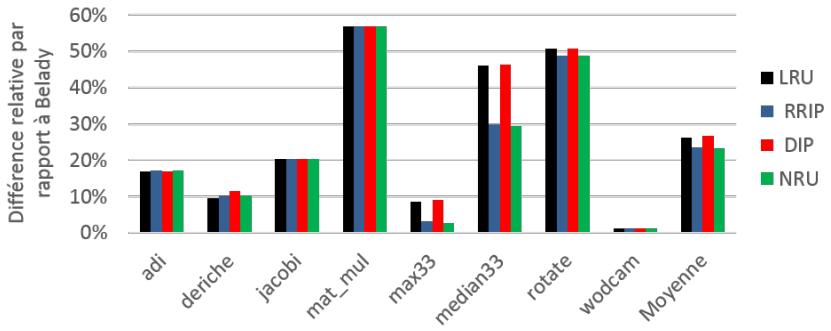


FIGURE 6.3 – Variation du taux de défauts du cache RO selon les différentes politiques de gestion rapportées à la politique optimale de Belady

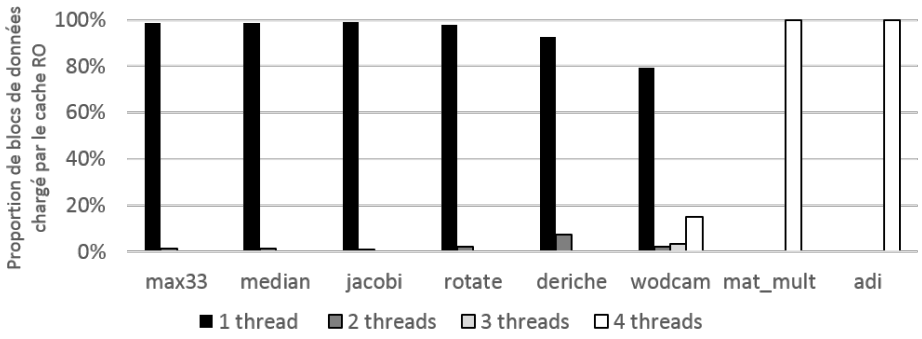


FIGURE 6.4 – Partage des blocs de cache chargé dans le cache RO

donc attribuée aux blocs partagés. Dans la même idée, Panda et al. [106] proposent une extension de la politique LRU pour donner une priorité plus importante aux blocs partagées. Ces solutions se basent sur les informations de cohérence stockées dans le répertoire du cache pour déterminer les informations de partage du bloc. Appliquer de tels mécanismes pour le cache RO ne présente que peu d'intérêt. En effet, dans ces situations, le cache manipule des blocs de données privés et partagés en même temps et les solutions permettent alors de consacrer plus de ressources en cache ou d'augmenter la durée de vie des blocs partagés. Notre situation est différente ou ces deux types de blocs ne cohabitent pas en même temps dans le cache RO. En effet, la figure 6.4 montre le motif de partage des blocs dans le cache RO, on peut observer que sur les applications évaluées jusqu'à présent, les blocs de données sont soit essentiellement en mode partagé entre tous les coeurs ou soit essentiellement en mode privé, on n'observe jamais les deux comportements en même temps.

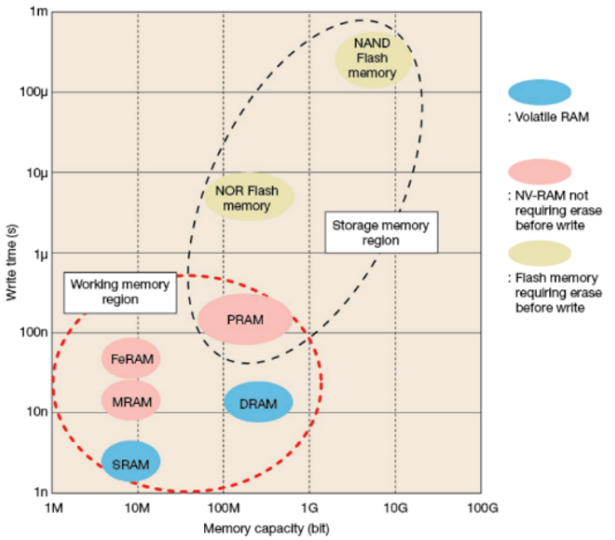


FIGURE 6.5 – Comparaison des différentes technologies NVM[6]

6.3 Cas d'utilisation d'hybridation technologique avec le cache RO

6.3.1 Utilisation des technologies NVM dans la hiérarchie mémoire

Comme nous l'avons mentionné en introduction, une tendance importante actuellement de l'évolution des hiérarchies mémoires est l'utilisation de nouvelles technologies mémoire conjointement avec l'utilisation la technologie SRAM habituellement utilisées pour l'implémentation des caches. Des travaux récents [28, 29] suggèrent des gains importants en terme de consommations des solutions qui utilisent des technologies SRAM et NVM pour un même cache, ce qui permet de tirer le meilleur parti des deux technologies. Parmi l'éventail des technologies NVM existantes, la figure 6.5 montre que celle qui s'approche le plus de la technologie SRAM en terme de latence est la technologie Magnetic-RAM ou MRAM, elle est donc la meilleure candidate pour être utilisé pour des caches proches processeurs pour lesquelles la latence est un facteur critique.

De façon générale, la technologie MRAM a une densité plus importante que la SRAM, une consommation statique d'un ordre de grandeur plus faible mais présentent une latence asymétrique entre la lecture et l'écriture, l'écriture étant un ordre de grandeur supérieur à la SRAM que ce soit en latence ou en énergie. Egalement, avec les technologies NVM et la MRAM en particulier, l'endurance de la mémoire peut devenir un critère d'optimisation car certains types de technologies possèdent une endurance de plusieurs ordres de grandeurs supérieure à la SRAM.

Une cellule MRAM est constitué de deux couches ferromagnétiques séparées par une couche mince d'isolant. Elle exploite l'effet dit de magnétorésistance à effet

TABLE 6.2 – Paramètres de la cellule de Base MRAM pour les différentes technologies

Technologie	Taille de cellule (F^2)	Temps d'accès lecture/écriture	Courant Ecriture	Endurance	Maturité
Toggle MRAM [107, 108]	50	35ns/35ns	>30mA	10^{15}	Commercialisé
TAS-MRAM [109]	<50	30ns/30ns	a few mA	10^{15}	Circuit Test
STT-MRAM [110, 111]	<50	2-20ns/2-20ns	50uA	$> 10^{16}$	Circuit Test
SOT-MRAM [112, 113]	<50	a few ns	< 100uA	$> 10^{16}$	Prototype

tunnel pour faire varier la résistance en fonction des champs magnétiques entre les deux aimants. Si ces deux champs sont parallèles, alors la résistance de l'isolant est alors faible. Si les champs sont anti-parallèles, la résistance devient maximale. Cette propriété permet donc d'encoder une information binaire, selon l'état de la résistance. L'écriture d'une information dans la cellule se fait en modifiant l'orientation de ce champs et selon la méthode utilisée, cela crée différentes catégories de solutions : la STT-MRAM (*Spin Torque Transfer*, la *Toggle*-MRAM,la SOT-MRAM et la TAS-MRAM. Un récapitulatif des cellules publiée est proposé avec le tableau 6.2.

Certaines des cellules proposées sont en tout point supérieures à la SRAM, et donc un simple remplacement de technologie dans ce cas reste la meilleure solution. Par exemple, Oboril et al[114] explorent différentes combinaisons de technologies entre la SRAM et la SOT-MRAM. L'étude effectuée suggère que pour les caches supérieurs à 128kO, les paramètres de mémoires de latence, de surface, de consommation statique ou dynamique, sont à l'avantage de la SOT-MRAM comparée à une mémoire SRAM et donc un simple remplacement de technologie dans ce cas est la meilleure stratégie à adopter. Pour les caches proches processeurs de taille inférieure à ce seuil, la consommation dynamique d'une mémoire SOT-MRAM reste cependant supérieure. Mais ce facteur est nettement contrebalancé par une consommation statique d'un ordre de grandeur inférieur à la SRAM. La solution optimale en termes de consommation reste donc encore un remplacement direct de la technologie des caches proches processeurs que ce soit pour le cache d'instruction ou le cache de données.

Pour la STT-RAM, certains paramètres restent moins bons que la SRAM, et il devient alors intéressant de combiner les différentes technologies afin de tirer parti du meilleur des deux. Au niveau architectural, cela aboutit à des solutions originales mais qui dépendent largement de la cellule STT-RAM considérée. Ainsi, par exemple, dans les solutions proposées par Jadidi et al. [115] et Wang et al. [116], les bancs de caches STT-RAM ont un cout nettement plus important pour l'écri-

ture que ce soit pour la latence ou la consommation dynamique. Ils utilisent donc conjointement des bancs de cache SRAM et STT-RAM et cherchent à rediriger les écritures vers les bancs SRAM. Pour cela, Jadidi et al. [115] utilisent des saturation de compteurs en écriture sur les bancs STT-RAM alors que Wang et al. [116] modifient la machine d'état du protocole de cohérence pour déterminer le placement du bloc sur un banc à la technologie adaptée. Ces solutions dépendent donc directement des paramètres de cellules considéré et par exemple, une autre cellule STT-RAM proposé par Komalan et al. [117] présente une asymétrie inverse entre cout en lecture et cout en écriture. Il devient donc plus favorable d'exécuter les écritures sur des bancs de STT-RAM que sur des bancs SRAM, la solution proposée est donc diamétralement opposée à celle des travaux précédents.

La technologie STT-MRAM étant encore à l'état de recherche, les paramètres ne convergent donc pas encore parfaitement. Une revue de l'état de l'art[118] sur les technologie NVM a compilé plus de 300 publications sur les technologies NVM notamment la STT-RAM, publiées entre 2000 et 2014, ces résultats peuvent présenter une variabilité d'un ordre de magnitude que ce soit pour la latence en lecture, la latence en écriture ou pour la surface. Cependant, plusieurs outils comme le simulateur NVSim [119] ou NVmain [120] ont été créés pour permettre d'étudier les possibilités de telles technologies au niveau architecturale de la même façon que CACTI [35] et DRAMsim [121] le proposent déjà pour des technologies standards. Ces simulateurs se basent sur les rapport de l'ITRS et sur le simulateur Mastar [98] pour les paramètres technologiques. Ces outils libres permettent ainsi de travailler sur des modèles de cellules réutilisables et permettre ainsi des points de comparaison entre les différentes solutions. Nous proposons dans la suite de cette section d'étudier l'impact d'un remplacement de technologie vers une technologie STT-MRAM pour le cache RO à l'aide de NVSim [119].

6.3.2 Simulations et résultats

Le design du cache RO a été simulé avec NVMSim et le tableau 6.3 compare les paramètres de cellules, à design identique, avec une technologie SRAM standard (modélisation effectuée avec CACTI). Les paramètres montrent une asymétrie entre lecture et écriture que ce soit en consommation ou en performance pour la technologie NVM, asymétrie quasi inexistante pour la technologie SRAM. Du fait que le cache RO reste de petite taille, sa consommation est principalement dominée par sa consommation dynamique et bénéficie donc peu de la réduction importante de la puissance statique apportée par le changement de technologie. Le cout important en écriture sera surtout contrebalancé par un cout en lecture plus faible de 20% par rapport à une technologie standard.

Les gains en consommation sont donc améliorés de 4% en moyenne par rapport à une solution avec une technologie standard alors que les légers gains observés précédemment sur l'AMAT de 6% en moyenne sont réduis pour passer à 2%. Le remplacement direct est bénéfique en consommation sur toutes les applications sauf *wodcam*. Ces derniers sont logiquement corrélés à la réutilisation des données

TABLE 6.3 – Paramètres de cellules avec une technologie SRAM et une technologie STT-RAM

	SRAM	STT-RAM
Energie Lecture	0,051nJ	0,037nJ
Energie Ecriture	0,049nJ	0,729nJ
Latence Lecture	0,232ns	1,587ns
Latence Ecriture	0,176ns	10,261ns
Puissance Statique	121,259mW	31,095mW
Surface	$0{,}315mm^2$	$0{,}167mm^2$

dans le cache RO et donc à son taux de défauts. Empiriquement, plus le taux de défauts est faible, plus les gains sont importants. Cette observation intuitive peut être analysée plus finement à l'aide d'une adaptation de la modélisation de la consommation dynamique du cache tel que proposé par Zhang et al. [122] suggère que les blocs de caches doivent être réutilisés dans le cache RO un certain nombre de fois afin de compenser le cout en écriture plus élevé du cache RO.

$$ConsoDyn_{SRAM} = nbAccesRO * CoutAcces_{SRAM} + nbDefauts * CoutDefaut_{SRAM}$$
$$CoutDefaut_{SRAM} = CoutLectureL2 + CoutAcces_{SRAM} + ...$$

$$ConsoDyn_{NVM} = nbAccesRO * CoutLecture_{NVM} + nbDefauts * CoutDefaut_{NVM}$$
$$CoutDefaut_{NVM} = CoutLectureL2 + CoutEcriture_{NVM} + ...$$

A partir de ce modèle, on peut donc représenter l'évolution de la consommation dynamique du cache RO en fonction du nombre d'accès à un bloc sous la forme de droites pour les deux technologies, ceci dû au fait que le cache RO ne subit pas d'écritures de la part des processeurs. Cette consommation dynamique augmente plus vite en fonction du nombre de réutilisation pour une technologie SRAM du fait d'un cout en lecture plus important mais le cout initial pour apporter le bloc dans le cache est moins important du fait du cout en écriture plus faible. On peut déterminer à gros grain le nombre moyen d'accès d'un bloc à travers le cache RO $reuse_{lim}$ à partir duquel le remplacement de technologie vers une technologie NVM soit rentable en terme de consommation, il s'agit du point d'intersection des deux droites. On obtient pour le calcul de $reuse_{lim}$:
$reuse_{lim} = (CoutEcriture_{NVM} - CoutAcces_{SRAM})/(CoutAcces_{SRAM} - CoutLecture_{NVM})$
A partir des paramètres des deux cellules du tableau 6.3, $reuse_{lim}$ vaut 48,6 ce qui correspond à un taux de défauts de cache de 2,1%. Ce résultat signifie qu'à partir d'un taux de défauts de cache inférieur à ce seuil, le gain sur le cout en lecture dépasse la perte sur le cout en écriture et le changement de technologie devient alors bénéfique en termes de consommation. Ce modèle néglige la consommation statique du cache RO mais qui dans nos simulations reste peu importante par rapport à son énergie dynamique et l'on peut vérifier que seule *wodcam* ne bénéficie

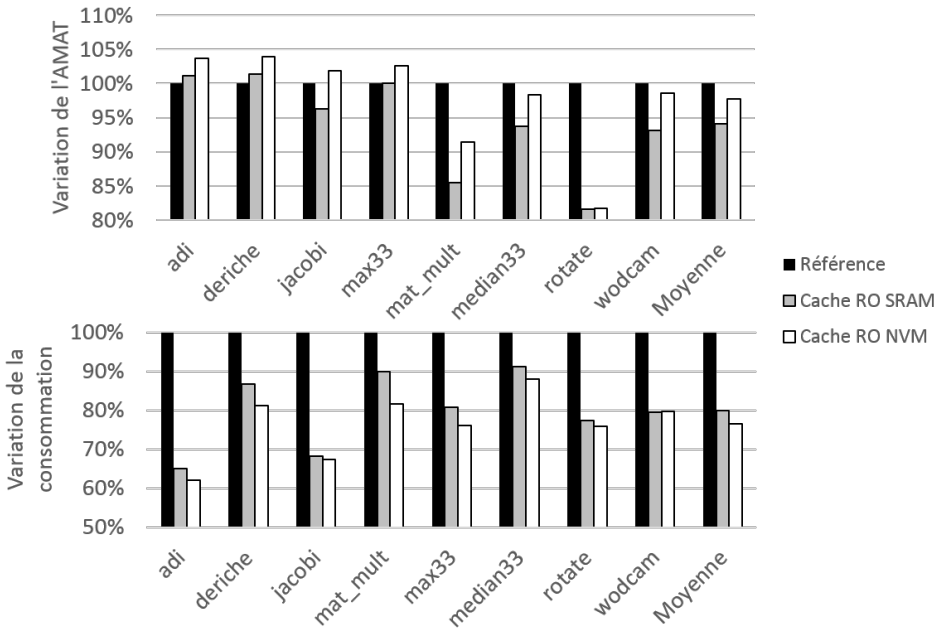


FIGURE 6.6 – Variation de la consommation et de la performance de la hiérarchie mémoire avec un cache RO STT-MRAM

pas en terme de consommation du changement de technologie car elle est la seule application dont le taux de défauts du cache RO dépasse les 2%.

Cependant, l'impact en terme de performance reste important également à cause de la latence en écriture dans le cache qui elle, n'est pas compensée. La dégradation est donc inévitable. La latence d'accès au tableau de données du cache NVM est donc fixée à 11 cycles car nos simulations s'effectuent une fréquence d'horloge à 1GHz dans le cache. Les gains observés sur l'AMAT dans le chapitre précédent qui étaient de 6% passe à moins de 2% en moyenne sur l'ensemble des applications. Le compromis performance/énergie proposé par une telle solution reste correct et permet d'améliorer les gains en consommation de 3,8%. Le remplacement direct de technologie vers une technologie STT-MRAM peut donc être envisagée sans optimisation particulière pour le cache RO et peut ainsi représenter un cas d'utilisation de ce type de technologie, là ou pour le cache d'instruction ou le cache de données l'impact de l'écriture est plus important et ne permet pas un simple remplacement.

6.3.3 Discussion

Dans la littérature, les travaux vont généralement au-delà d'un simple remplacement de la technologie et proposent des solutions à base d'hybridation entre plusieurs technologies pour réduire l'impact négatif des performances des écri-

TABLE 6.4 – Distribution des distances de réutilisation des accès faits au cache RO

Distance de réutilisation	[1-31]	[32-255]	[255-1024]	[1024-]
adi	98,0%	0,4%	0,1%	1,5%
deriche	98,5%	0,5%	0%	0,9%
jacobi	91,9%	4,5%	3,3%	0,2%
median33	97,4%	0,4%	1,3%	0,0%
rotate	96,1%	1,2%	1,3%	0,0%
mat_malt	96,1%	0,3%	3,5%	0,1%
wodcam	93,3%	0,1%	0,1%	6,5%

tures. Cette proposition a du sens pour un cache de données classique ou l'on peut distinguer des comportements distincts en termes de lecture et d'écriture qui peuvent être exploités à travers des bancs de technologies différentes. Cette proposition a également du sens pour un cache d'instruction si une différence en terme de réutilisation existe qui peut être potentiellement exploitée comme cela est fait par exemple avec Komalan et al.[123] qui, pour le cache d'instruction, stocke les blocs dans un petit buffer de technologie SRAM (une extension du MHSR) et attend qu'un bloc soit accédé plusieurs fois avant de le stocker effectivement sur un banc de cache STT-MRAM. Cela permet d'éviter de stocker sur des bancs NVM des blocs peu réutilisés. Cependant, les données passant par le cache RO, possèdent un comportement homogène en termes de réutilisation. C'est ce que montre la distribution des distances de réutilisation des accès du cache RO pour chaque application montré dans le tableau 6.3.3. On peut voir que la majorité des accès ont des distances de réutilisation très faible, et une telle solution ne permettrait donc pas d'avantager dans le cache RO des données qui seraient plus réutilisées par rapport à d'autres car une telle différence n'est peu ou pas présente à part pour *wodcam*.

6.4 Conclusion

A partir d'une proposition d'architecture effectuée dans le chapitre précédent, il a été exposé ici quelques optimisations propres au cache RO qui permettent d'améliorer les résultats obtenus dans le chapitre précédent. Les gains obtenus dans ce chapitre restent cependant peu importants relativement aux gains obtenus sur la proposition architecturale principale. Cela s'explique par le fait que les optimisations proposées dans ce chapitre n'impactent que le cache RO qui compte pour une faible part dans les performances et la consommation du système. Cela permet cependant d'étudier des techniques d'optimisation originales spécifiques au cache RO.

Chapitre 7

Conclusions et Perspectives

Sommaire

Les architectures modernes sont confrontées à des problématiques d'efficacité énergétique de la hiérarchie pour des architectures proposant un nombre croissant de coeurs que ce soit pour le domaine de l'embarqué ou du calcul haute performance et ces problématiques poussent à des améliorations continues de l'utilisation des caches et de la réduction de leurs consommations. Dans ce cadre, parmi plusieurs facteurs de consommation, nous avons relevé 3 facteurs importants que sont la localité des données, le design des caches associés et le protocole de cohérence. Adopter une gestion différenciée des données dans la hiérarchie mémoire permet de renforcer la localité des données déjà présente et de proposer des designs spécifiques à chaque type de données. De telle idées ont déjà été utilisées et exploitées pour différents types de données mais peu de travaux adressent directement la gestion spécifique des données en lecture seule. Une étude des données en lecture a permis d'abord une justification de l'approche choisie et les résultats de l'exploration architecturale montrent des gains significatifs sur la consommation de la hiérarchie mémoire avec des applications de traitement d'image.

7.1 Synthèse des Travaux

Une synthèse de la thèse et des différentes contributions :

— A partir d'une définition souple d'une donnée en lecture seule, une analyse qualitative et quantitative du comportement des données en lecture seule a été faite sur deux suites d'applications. Elle montre un taux de détection significatif dans les applications ainsi qu'une différence de localité importante par rapport aux autres données. Cette étude a fait l'objet d'une publication au workshop EWiLi'14 [124]

— Une chaine de compilation qui permet une détection des données en lecture seule sur un chemin de données particulier de façon automatique. L'évaluation de cette chaine montre un taux de détection suffisant de ces données pour rendre possible une optimisation architecturale transparente pour l'utilisateur.

— Une exploration architecturale sur la gestion des données en lecture seule au sein d'une hiérarchie mémoire sur architecture multi-cœurs. Cette exploration montre qu'une solution appropriée pour les données en lecture seule permet d'obtenir des gains en consommation significatif tout en conservant des performances stables. Une première version de la solution proposant entre un codesign architecture compilation a d'abord été publiée dans la conférence PDP'16 et une étude plus complète avec quelques optimisation a été publiée à la conférence SBAC-PAD 2016.

7.2 Perspectives

7.2.1 A Court Terme

Une gestion hétérogène de la mémoire

Un des premiers axes d'amélioration des travaux existants est l'extension d'une telle solution à d'autres domaines applicatifs que le traitement d'image. Bien que l'on puisse utiliser une méthodologie d'étude identique, la solution finale pourra alors être différente car elle dépend largement des caractéristique en terme de localité et de partage, des données en lecture seule des applications. Pour continuer à explorer le potentiel d'une telle solution, plusieurs axes de recherche peuvent être envisagés à court terme :

Le modèle utilisé à la compilation pour décrire les zones de lecture seule ne cherche pas, pour le moment, à décrire les motifs d'accès à l'intérieur de cette zone. Dans le cadre d'applications de traitement d'image, les boucles de calcul appartiennent souvent à la classe des SCoP [1]. Dans cette classe de boucle, on peut envisager d'analyser les motifs d'accès aux zones en lecture seule pour évaluer l'intérêt en termes de localité de placer les zones de données en lecture seule dans un cache séparé. Pour le moment, toutes les zones détectées comme étant en lecture seule sont placées sur le chemin de données alternatif, mais l'on peut imaginer des situations dans lesquelles cette séparation n'a pas un impact bénéfique sur le système. Des heuristiques de compilation peuvent alors être proposées pour éviter ce type de situation.

Certains principes exploités par notre solution ne sont pas spécifique aux données en lecture seule et l'on peut se demander si une telle solution pourrait convenir à d'autres types de données. Par exemple, nous avons vu que notre solution bénéficie entre autre de la réutilisation entre plusieurs threads au sein du cache au niveau L1 et l'on peut se demander l'intérêt d'une telle approche pour des données qui seraient exclusivement partagées. La difficulté d'une telle approche serait alors la classification des données à la compilation car il est plus difficile à la compilation

1. textitStatic Control Program : il s'agit d'un programme qui possède des boucles dont les bornes et les fonctions d'accès aux tableaux sont des fonctions affines des itérateurs de boucles

d'étudier la façon dont sont partagées les données que de rechercher des zones de lectures. Cependant, une telle approche pourrait s'appuyer sur les informations directement renseignées par l'utilisateur à travers le modèle de programmation parallèle utilisé comme par exemple les attributs *shared* ou *private* dans les directives OpenMP.

Dans ces travaux, nous avons considéré uniquement des mémoires caches pour effectuer la gestion spécifiques des données en lecture seule. Du fait d'une maitrise à la compilation des données placées sur ce chemin de données, une utilisation automatique à travers une mémoire scratchpad peut être envisagée. Cela permettrait d'améliorer l'efficacité énergétique du chemin de données en exploitant un cout par accès plus faible et une surface plus faible à taille de mémoire équivalente. Afin de résoudre le problème de cohérence que présenterait l'utilisation de cache et de mémoire scratchpad au niveau L1, on pourra rapprocher nos travaux à ceux d'Alvarez et al.[125] qui sont confrontés à des problématiques similaire.

7.2.2 A Long Terme

La tendance à l'hétérogénéité de traitement dans les architectures actuelle est une tendance importante, on l'on trouve ce type d'approche que ce soit à l'intérieur des unités de calculs avec l'utilisation d'extensions appropriés ou éalement au niveau de la hiérarchie mémoire. Un exemple important, dans le domaine du calcul haute performance est la nouvelle génération du Xeon phi, les Knights Landing qui succèdent aux Knights Corner, qui inclut une mémoire de 16 Go de technologie MCDRAM (*Multi-Channel DRAM*) utilisable conjointement avec la mémoire DRAM. La principale différence entre les deux mémoires est leurs bandes passantes qui est de 90GB/S pour la DRAM alors qu'elle est de 400GB/s pour la MCDRAM. On trouve plusieurs modes d'organisation entre ces deux mémoires : un mode cache ou la MCDRAM se comporte comme un cache L4, un mode scratchpad ou les deux mémoires possèdent des espaces d'adressages différents et un mode proposant un mix entre les deux premiers. Une telle dualité à ce niveau de mémoire pose des questions du même ordre que celles posées dans cette thèse, sur la répartition des données à effectuer entre les deux mémoires, cette dernière pouvant alors donc accueillir des données critiques en terme de bande passante vers la mémoire.

Du côté des technologies non volatiles, après être restées longtemps dans le domaine de la recherche, des circuits commercialisés ont commencé à voir le jour à partir du début des années 2010. Par exemple, la technologie de mémoire 3D Xpoint d'Intel développé conjointement avec Micron et annoncée en Juillet 2015 se base sur de la technologie ReRAM. Cette technologie est moins chère que la DRAM mais reste plus cher qu'une technologie NAND. De la même façon, elle est plus rapide qu'une technologie NAND mais plus lente qu'une technologie DRAM. Parfois vu comme une alternative à la technologie NAND, elle pourrait

aussi se positionner comme une mémoire intermédiaire entre la mémoire NAND et la mémoire principale DRAM ou, comme dans le cas de la MCDRAM, on peut imaginer qu'elle soit utilisée en parallèle d'une technologie DRAM classique. Une bonne utilisation d'une telle mémoire suppose d'y placer des données qui sont majoritairement lues et on peut alors effectuer une transposition de nos travaux qui se situent au niveau cache, ou le même type d'analyse à la compilation peut être pertinent pour le placement des données dans une telle mémoire. Cela pourrait permettre également de tirer parti d'une propriété que nous n'avons pas eu l'occasion d'exploiter dans ces travaux à savoir a la propriété de région continues de mémoire.

Liste des Publications

Articles Scientifiques

Read Only Data Specific Management for an Energy Efficient Memory System
Vaumourin G., Barthou D., Guerre A., Dombek T.
En soumission à SBAC-PAD, the 28th International Symposium on Computer Architecture and High Performance Computing

Specific Read-only Data Management for Memory System Optimization
Vaumourin G., Barthou D., Guerre A., Dombek T.
PDP'16 , The 24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing

Co-simulating complex energy harvesting WSN applications : an in-tunnel wind powered monitoring example
Quang V., Didioui A., Vaumourin G., Bernier C., Broekaert F., Fritsch A.
IJSNet, International Journal of Sensor Networks

Specific read only data management for memory hierarchy optimization
Vaumourin G., Barthou D., Guerre A., Dombek T.
EWiLi'14, The 4th Embedded Operating Systems Workshop

Posters

Reducing Memory System Energy Consumption by Read-only Data Specific Management
Vaumourin G., Barthou D., Guerre A., Dombek T.
Colloque du GdR SoC SiP 2016

Improving Caches Energy Consumption by Specific Read-only Data Management
Vaumourin G., Barthou D., Guerre A., Dombek T.
Ecole d'été ACACES, 11th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems

Exploitation des données en lecture seule dans les hiérarchies mémoires
Vaumourin G., Barthou D., Guerre A., Dombek T.
Colloque du GdR SoC SiP 2014

Bibliographie

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition : A Quantitative Approach*, 5th ed. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2011.

[2] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 211–222, Oct. 2002. [Online]. Available : http://doi.acm.org/10.1145/635506.605420

[3] S. Rodriguez and B. Jacob, "Energy/power breakdown of pipelined nanometer caches (90nm/65nm/45nm/32nm)," in *Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, ser. ISLPED '06. New York, NY, USA : ACM, 2006, pp. 25–30. [Online]. Available : http://doi.acm.org/10.1145/1165573.1165581

[4] H.-H. S. Lee and G. S. Tyson, "Region-based caching : An energy-delay efficient memory architecture for embedded processors," in *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '00. New York, NY, USA : ACM, 2000, pp. 120–127. [Online]. Available : http://doi.acm.org/10.1145/354880.354898

[5] B. Cuesta, A. Ros, M. E. Gomez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by avoiding the tracking of noncoherent memory blocks," *IEEE Trans. Comput.*, vol. 62, no. 3, pp. 482–495, Mar. 2013. [Online]. Available : http://dx.doi.org/10.1109/TC.2011.241

[6] Y. Xie, "Cost/architecture/application implications for 3d stacking technology," in *2017, ASP-DAC*. IEEE, 2017.

[7] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 5, pp. 33–35, Sept 2006.

[8] R. H. Dennard, F. H. Gaensslen, H. nien Yu, V. L. Rideout, E. Bassous, Andre, and R. Leblanc, "Design of ion-implanted mosfets with very small physical dimensions," *IEEE J. Solid-State Circuits*, p. 256, 1974.

[9] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee : A many-core x86 architecture for visual computing," in *ACM SIGGRAPH 2008 Papers*, ser. SIGGRAPH '08. New York, NY, USA : ACM, 2008, pp. 18 :1–18 :15. [Online]. Available : http://doi.acm.org/10.1145/1399504.1360617

[10] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown *et al.*, "Tile64-processor : A 64-core soc with mesh interconnect," in *2008 IEEE International Solid-State Circuits Conference-Digest of Technical Papers*. IEEE, 2008, pp. 88–598.

[11] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the amd opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, March 2010.

[12] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "Power5 system microarchitecture," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 505–521, Jul. 2005.

[13] S. Inc., "The s6000 family of processors," http://www.stretchinc.com/files/soArcMtectureOverview.pdf,, accessed : 2016-06-30.

[14] P. Cumming, "The ti omap^{TM} platform approach to soc," in *Winning the SOC Revolution*. Springer, 2003, pp. 97–118.

[15] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 1st ed. Morgan & Claypool Publishers, 2011.

[16] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the ieee futurebus," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86. Los Alamitos, CA, USA : IEEE Computer Society Press, 1986, pp. 414–423. [Online]. Available : http://dl.acm.org/citation.cfm?id=17407.17404

[17] A. Charlesworth, "Starfire : Extending the smp envelope," *IEEE Micro*, vol. 18, no. 1, pp. 39–49, Jan. 1998.

[18] J. Laudon and D. Lenoski, "The sgi origin : A ccnuma highly scalable server," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 241–251, May 1997.

[19] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA : ACM, 2012, pp. 241–252. [Online]. Available : http://doi.acm.org/10.1145/2370816.2370853

[20] S. Kaxiras and A. Ros, "Efficient, snoopless, system-on-chip coherence," in *SOCC*, 2012.

[21] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras, "The effects of granularity and adaptivity on private/shared classification for coherence," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 3, pp. 26 :1–26 :21, Aug. 2015. [Online]. Available : http://doi.acm.org/10.1145/2790301

[22] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory : Design alternative for cache on-chip memory in embedded systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, ser. CODES '02, 2002.

[23] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10 : An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005. [Online]. Available : http://doi.acm.org/10.1145/1103845.1094852

[24] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC : Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[25] J. Andrews and N. Baker, "Xbox 360 system architecture," *IEEE Micro*, vol. 26, no. 2, pp. 25–37, March 2006.

[26] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, Jul. 2005. [Online]. Available : http://dl.acm.org/citation.cfm?id=1148882.1148891

[27] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing memory systems for chip multiprocessors," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 358–368, Jun. 2007. [Online]. Available : http://doi.acm.org/10.1145/1273440.1250707

[28] X. Wu, J. Li, L. Zhang, E. Speight, and R. Rajamony, "Hybrid cache architecture with disparate memory technologies," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009. [Online]. Available : http://doi.acm.org/10.1145/1555754.1555761

[29] L. Jiang, B. Zhao, Y. Zhang, and J. Yang, "Constructing large and fast multi-level cell stt-mram based cache for embedded processors," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA : ACM, 2012, pp. 907–912. [Online]. Available : http://doi.acm.org/10.1145/2228360.2228521

[30] A. Cohen, M. Duranton, S. Yehia, B. De Sutter, K. De Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, N. Navarro, A. Ramirez, O. Temam, and M. Valero, *The HiPEAC Vision*, M. Duranton, Ed. HiPEAC network of excellence, 2015. [Online]. Available : https://hal.inria.fr/inria-00551078

[31] S. S., "Low power design techniques for microprocessors," in *International Solid State Circuit Conference*, February 2001.

[32] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stehpany, and S. C. Thierauf, "A 160-mhz, 32-b, 0.5-w cmos risc microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 11, pp. 1703–1714, Nov 1996.

[33] A. Sodani and C. Processor, "Race to exascale : Opportunities and challenges," 2011.

[34] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench : A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01, 2001. [Online]. Available : http://dx.doi.org/10.1109/WWC.2001.15

[35] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Architecting efficient interconnects for large caches with cacti 6.0," *Micro, IEEE*, vol. 28, no. 1, pp. 69–79, Jan 2008.

[36] P. Feautrier, "Some efficient solutions to the affine scheduling problem : I. one-dimensional time," *Int. J. Parallel Program.*, vol. 21, no. 5, pp. 313–348, Oct. 1992.

[37] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," *In Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, 2011.

[38] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan, "A model for fusion and code motion in an automatic parallelizing compiler," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA : ACM, 2010, pp. 343–352.

[39] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin, "Productivity via automatic code generation for pgas platforms with the r-stream compiler," 2009.

[40] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '78. New York, NY, USA : ACM, 1978, pp. 84–96. [Online]. Available : http://doi.acm.org/10.1145/512760.512770

[41] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *SIGPLAN Not.*, vol. 26, no. 4, pp. 63–74, Apr. 1991. [Online]. Available : http://doi.acm.org/10.1145/106973.106981

[42] F. Bodin, W. Jalby, D. Windheiser, and C. Eisenbeis, "A quantitative algorithm for data locality optimization," in *In Code Generation-Concepts, Tools, Techniques*. Springer Verlag, 1992, pp. 119–145.

[43] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," *SIGPLAN Not.*, vol. 30, no. 6, pp. 279–290, Jun. 1995. [Online]. Available : http://doi.acm.org/10.1145/223428.207162

[44] K. Esseghir, "Improving data locality for caches," Ph.D. dissertation, Rice University, 1993.

[45] G. Rivera and C.-W. Tseng, "Data transformations for eliminating conflict misses," *SIGPLAN Not.*, vol. 33, no. 5, pp. 38–49, May 1998. [Online]. Available : http://doi.acm.org/10.1145/277652.277661

[46] D. H. Bailey, "Unfavorable strides in cache memory systems (rnr technical report rnr-92-015)," *Sci. Program.*, vol. 4, no. 2, pp. 53–58, Apr. 1995. [Online]. Available : http://dx.doi.org/10.1155/1995/937016

[47] N. Ahmed, N. Mateev, and K. Pingali, "Synthesizing transformations for locality enhancement of imperfectly-nested loop nests," *Int. J. Parallel Program.*, vol. 29, no. 5, pp. 493–544, Oct. 2001.

[48] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," *SIGPLAN Not.*, vol. 34, no. 5, pp. 215–228, May 1999. [Online]. Available : http://doi.acm.org/10.1145/301631.301668

[49] G. Rivera and C.-W. Tseng, "Locality optimizations for multi-level caches," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC '99. New York, NY, USA : ACM, 1999. [Online]. Available : http://doi.acm.org/10.1145/331532.331534

[50] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA : IEEE Computer Society Press, 2012, pp. 40 :1–40 :11. [Online]. Available : http://dl.acm.org/citation.cfm?id=2388996.2389051

[51] T. Grosser, S. Verdoolaege, A. Cohen, and P. Sadayappan, "The relation between diamond tiling and hexagonal tiling," *Parallel Processing Letters*, vol. 24, no. 03, p. 1441002, 2014.

[52] B. Bao and C. Ding, "Defensive loop tiling for shared cache," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. Washington, DC, USA : IEEE Computer Society, 2013, pp. 1–11. [Online]. Available : http://dx.doi.org/10.1109/CGO.2013.6495008

[53] Y. Zhang, M. Kandemir, and T. Yemliha, "Studying inter-core data reuse in multicores," in *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '11. New York, NY, USA : ACM, 2011, pp. 25–36. [Online]. Available : http://doi.acm.org/10.1145/1993744.1993748

[54] Z. Sustran, S. Stojanovic, G. Rakocevic, V. M. Milutinovic, and M. Valero, "A survey of dual data cache systems," in *Industrial Technology (ICIT), 2012 IEEE International Conference on*, March 2012, pp. 450–456.

[55] A. González, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *Proceedings of the 9th International Conference on Supercomputing*, ser. ICS '95. New York, NY, USA : ACM, 1995, pp. 338–347. [Online]. Available : http://doi.acm.org/10.1145/224538.224622

[56] K.-W. Lee, J.-S. Lee, G.-H. Park, J.-H. Lee, T.-D. Han, S.-D. Kim, Y.-C. Kim, S.-W. Jung, and K.-Y. Lee, "The cache memory system for calmrisc32," in *ASICs, 2000. AP-ASIC 2000. Proceedings of the Second IEEE Asia Pacific Conference on*, 2000, pp. 323–326.

[57] I. Corporations, "Intel strongarm sa-1110 microprocessor develper's manual," 2000.

[58] O. Adamo, A. Naz, K. Janjusic, Tommy ans Krishna, and C.-P. Chung, "Smaller split l-1 data caches for multi-core processing systems," *ISPAN, International Symposium on Pervasive Systems, Algorithms, and Networks*, vol. 30, no. 6, Jun. 2009.

[59] M. J. Geiger, S. A. Mckee, and G. S. Tyson, "Beyond basic region caching : Specializing cache structures for high performance and energy conservation," in

In Proc. 1st International Conference on High Performance Embedded Architectures and Compilers, 2005, pp. 70–75.

[60] M. J. Geiger, S. A. McKee, and G. S. Tyson, "Drowsy region-based caches : minimizing both dynamic and static power dissipation," in *Proceedings of the 2nd conference on Computing frontiers*. ACM, 2005, pp. 378–384.

[61] L. E. Olson, Y. Eckert, S. Manne, and M. D. Hill, "Revisiting stack caches for energy efficiency," 2014.

[62] S. c. Kang, C. Nicopoulos, H. Lee, and J. Kim, "A high-performance and energy-efficient virtually tagged stack cache architecture for multi-core environments," in *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, Sept 2011, pp. 58–67.

[63] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace snooping : Filtering snoops with operating system support," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA : ACM, 2010, pp. 111–122. [Online]. Available : http://doi.acm.org/10.1145/1854273.1854292

[64] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA : ACM, 2011, pp. 93–104. [Online]. Available : http://doi.acm.org/10.1145/2000064.2000076

[65] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones, "Compiler-assisted data distribution for chip multiprocessors," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA : ACM, 2010, pp. 501–512. [Online]. Available : http://doi.acm.org/10.1145/1854273.1854335

[66] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca : Near-optimal block placement and replication in distributed caches," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 184–195, Jun. 2009. [Online]. Available : http://doi.acm.org/10.1145/1555815.1555779

[67] Y. Li, R. Melhem, and A. K. Jones, "Practically private : Enabling high performance cmps through compiler-assisted data classification," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA : ACM, 2012, pp. 231–240. [Online]. Available : http://doi.acm.org/10.1145/2370816.2370852

[68] H. Hossain, S. Dwarkadas, and M. C. Huang, "Pops : Coherence protocol optimization for both private and shared data," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, Oct 2011, pp. 45–55.

[69] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "Swel : Hardware cache coherence protocols to map shared data onto shared caches,"

in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 465–476.

[70] A. Ros and A. Jimborean, "A hybrid static-dynamic classification for dual-consistency cache coherence," *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[71] A. Esteve, A. Ros, A. Robles, M. E. Gómez, and J. Duato, "Tokentlb : A token-based page classification approach," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA : ACM, 2016, pp. 26 :1–26 :13. [Online]. Available : http://doi.acm.org/10.1145/2925426.2926280

[72] A. Ros, M. Davari, and S. Kaxiras, "Hierarchical private/shared classification : the key to simple and efficient coherence for clustered cache hierarchies," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 186–197.

[73] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin : Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA : ACM, 2005, pp. 190–200. [Online]. Available : http://doi.acm.org/10.1145/1065010.1065034

[74] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970. [Online]. Available : http://dx.doi.org/10.1147/sj.92.0078

[75] B. T. Bennett and V. J. Kruskal, "Lru stack processing," *IBM J. Res. Dev.*, vol. 19, no. 4, pp. 353–357, Jul. 1975. [Online]. Available : http://dx.doi.org/10.1147/rd.194.0353

[76] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite : Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA : ACM, 2008, pp. 72–81. [Online]. Available : http://doi.acm.org/10.1145/1454115.1454128

[77] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA : ACM, 2008, pp. 101–113. [Online]. Available : http://doi.acm.org/10.1145/1375581.1375595

[78] W. Landi and B. G. Ryder, "Pointer-induced aliasing : A problem classification," in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '91. New York, NY, USA : ACM, 1991, pp. 93–103. [Online]. Available : http://doi.acm.org/10.1145/99583.99599

[79] B. Creusillet and F. Irigoin, "Interprocedural array region analyses," in *In Proceedings of the 8th International Workshop on Languages and Compilers for parallel Computing*. Springer-Verlag, 1995, pp. 46–60.

[80] H. Sharangpani, "Intel Itanium processor microarchitecture overview," instHP, Tech. Rep., 1999, presented at Microprocessor Forum, October 6–9, 1999.

[81] "Ia-64 application developer's architecture guide," 1999.

[82] K. Beyls and E. H. D'Hollander, "Generating cache hints for improved program efficiency," *J. Syst. Archit.*, vol. 51, no. 4, pp. 223–250, Apr. 2005. [Online]. Available : http://dx.doi.org/10.1016/j.sysarc.2004.09.004

[83] X. Gu and C. Ding, "On the theory and potential of lru-mru collaborative cache management," *SIGPLAN Not.*, vol. 46, no. 11, pp. 43–54, Jun. 2011. [Online]. Available : http://doi.acm.org/10.1145/2076022.1993485

[84] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems, "Using the compiler to improve cache replacement decisions," in *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '02. Washington, DC, USA : IEEE Computer Society, 2002, pp. 199–. [Online]. Available : http://dl.acm.org/citation.cfm?id=645989.674328

[85] A. Mukkara, N. Beckmann, and D. Sanchez, "Whirlpool : Improving dynamic cache management with static data classification," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA : ACM, 2016, pp. 113–127. [Online]. Available : http://doi.acm.org/10.1145/2872362.2872363

[86] J. Merrill, "Generic and gimple : A new tree representation for entire functions," in *in Proc. GCC Developers Summit, 2003*, 2003, pp. 171–180.

[87] V. Lefebvre and P. Feautrier, "Automatic storage management for parallel programs," *Parallel Comput.*, vol. 24, no. 3-4, pp. 649–671, May 1998. [Online]. Available : http://dx.doi.org/10.1016/S0167-8191(98)00029-5

[88] B. W. Kernighan, *The C Programming Language*, 2nd ed., D. M. Ritchie, Ed. Prentice Hall Professional Technical Reference, 1988.

[89] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2000.

[90] D. J. Pearce, P. H. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis of c," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, Nov. 2007. [Online]. Available : http://doi.acm.org/10.1145/1290520.1290524

[91] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, "Is reuse distance applicable to data locality analysis on chip multiprocessors ?" in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, ser. CC'10/ETAPS'10. Berlin, Heidelberg : Springer-Verlag, 2010, pp. 264–282.

[92] P. Glaskwsky, "Nvida's fermi : The first complete gpu computing architecture," *White Paper*, 2009.

[93] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, and R. Sen, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011. [Online]. Available : http://doi.acm.org/10.1145/2024716.2024718

[94] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005. [Online]. Available : http://doi.acm.org/10.1145/1105734.1105747

[95] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The m5 simulator : Modeling networked systems," *IEEE Micro*, vol. 26, pp. 52–60, 2006.

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

[97] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat : An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 469–480.

[98] S. I. Association, "Model for assessment of cmos technologies and roadmaps (mastar)," 2007. [Online]. Available : http://www.itrs.net/models.html

[99] K. Samira, A. Alaa, and W. Chris, "Improving cache performance by exploiting read-write disparity," in *In processings on High Performance Computer Architecture (HPCA)*, 2014.

[100] S. Kaxiras and G. Keramidas, "Sarc coherence : Scaling directory cache coherence in performance and power," *IEEE Micro*, vol. 30, no. 5, pp. 54–65, Sep. 2010. [Online]. Available : http://dx.doi.org/10.1109/MM.2010.82

[101] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA : ACM, 2007, pp. 381–391. [Online]. Available : http://doi.acm.org/10.1145/1250662.1250709

[102] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 60–71, Jun. 2010. [Online]. Available : http://doi.acm.org/10.1145/1816038.1815971

[103] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[104] R. Natarajan and M. Chaudhuri, "Characterizing multi-threaded applications for designing sharing-aware last-level cache replacement policies," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 1–10.

[105] Y. Chen, W. Li, C. Kim, and Z. Tang, "Efficient shared cache management through sharing-aware replacement and streaming-aware insertion policy," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–11.

[106] B. Panda and S. Balachandran, "Csharp : Coherence and sharing aware cache replacement policies for parallel applications," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*. IEEE, 2012, pp. 252–259.

[107] I. L. Prejbeanu, S. Bandiera, J. Alvarez-Hérault, R. C. Sousa, B. Dieny, and J.-P. Nozières, "Thermally assisted mrams : ultimate scalability and logic functionalities," *Journal of Physics D : Applied Physics*, vol. 46, no. 7, p. 074002, 2013. [Online]. Available : http://stacks.iop.org/0022-3727/46/i=7/a=074002

[108] T. W. Andre, J. J. Nahas, C. K. Subramanian, B. J. Garni, H. S. Lin, A. Omair, and W. L. Martino, "A 4-mb 0.18- mu ;m 1t1mtj toggle mram with balanced three input sensing scheme and locally mirrored unidirectional write drivers," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 1, pp. 301–309, Jan 2005.

[109] R. Bishnoi, M. Ebrahimi, F. Oboril, and M. B. Tahoori, "Architectural aspects in design and analysis of sot-based memories," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2014, pp. 700–707.

[110] X. Fong and K. Roy, "Complimentary polarizers stt-mram (cpstt) for on-chip caches," *IEEE Electron Device Letters*, vol. 34, no. 2, pp. 232–234, Feb 2013.

[111] A. V. Khvalkovskiy, D. Apalkov, S. Watts, R. Chepulskii, R. S. Beach, A. Ong, X. Tang, A. Driskill-Smith, W. H. Butler, P. B. Visscher, D. Lottis, E. Chen, V. Nikitin, and M. Krounbi, "Basic principles of STT-MRAM cell operation in memory arrays," *Journal of Physics D Applied Physics*, vol. 46, no. 7, p. 074001, Feb. 2013.

[112] W. Kang, Z. Wang, Y. Zhang, J.-O. Klein, W. Lv, and W. Zhao, "Spintronic logic design methodology based on spin hall effect–driven magnetic tunnel junctions," *Journal of Physics D : Applied Physics*, vol. 49, no. 6, p. 065008, 2016. [Online]. Available : http://stacks.iop.org/0022-3727/49/i=6/a=065008

[113] I. M. Miron, G. Gaudin, S. Auffret, B. Rodmacq, A. Schuhl, J. Vogel, S. Pizzini, and P. Gambardella, "Current-driven spin torque induced by the Rashba effect in a ferromagnetic metal layer," *Nature Materials*, vol. 9, p. 230, Jan. 2010. [Online]. Available : https://hal.archives-ouvertes.fr/hal-00459160

[114] F. Oboril, R. Bishnoi, M. Ebrahimi, and M. B. Tahoori, "Evaluation of hybrid memory technologies using sot-mram for on-chip cache hierarchy,"

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 34, no. 3, pp. 367–380, 2015.

[115] A. Jadidi, M. Arjomand, and H. Sarbazi-Azad, "High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement," in *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, Aug 2011, pp. 79–84.

[116] J. Wang, Y. Tim, W. F. Wong, Z. L. Ong, Z. Sun, and H. H. Li, "A coherent hybrid sram and stt-ram l1 cache architecture for shared memory multicores," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2014, pp. 610–615.

[117] M. P. Komalan, C. Tenllado, J. I. G. Pérez, F. T. Fernández, and F. Catthoor, "System level exploration of a stt-mram based level 1 data-cache," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 1311–1316.

[118] K. Suzuki and S. Swanson, "A survey of trends in non-volatile memory technologies : 2000-2014," in *2015 IEEE International Memory Workshop (IMW)*, May 2015, pp. 1–4.

[119] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim : A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, July 2012.

[120] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0 : A user-friendly memory simulator to model (non-)volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, July 2015.

[121] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "Dramsim : a memory system simulator," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 100–107, 2005.

[122] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache architecture for embedded systems," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, June 2003, pp. 136–146.

[123] M. Komalan, J. I. G. Pérez, C. Tenllado, P. Raghavan, M. Hartmann, and F. Catthoor, "Feasibility exploration of nvm based i-cache through mshr enhancements," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14. 3001 Leuven, Belgium, Belgium : European Design and Automation Association, 2014, pp. 21 :1–21 :6. [Online]. Available : http://dl.acm.org/citation.cfm?id=2616606.2616632

[124] G. Vaumourin, T. Dombek, A. Guerre, and D. Barthou, "Specific read only data management for memory hierarchy optimization," in *Proceedings on the 4th Embedded Operating Systems Workshop*, 2014.

[125] L. Alvarez, L. Vilanova, M. Moreto, M. Casas, M. Gonzàlez, X. Martorell, N. Navarro, E. Ayguadé, and M. Valero, "Coherence protocol for transparent

management of scratchpad memories in shared memory manycore architectures," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA : ACM, 2015, pp. 720–732. [Online]. Available : http://doi.acm.org/10.1145/2749469.2750411