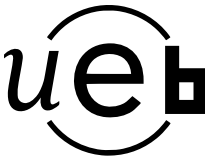
Modèles et protocoles de cohérence de données, décision et optimisation à la compilation pour des architectures massivement parallèles

Safae Dahmani

ANNÉE 2015



THÈSE / UNIVERSITÉ DE BRETAGNE SUD
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD

Mention : Maths/STIC

École doctorale SICMA

présentée par

Safae DAHMANI

préparée à l'unité de recherche LaBSTICC – UMR6285
Université de Bretagne Sud Faculté des Sciences et Sciences de l'Ingénieur

---

Modèles et protocoles de cohérence de données, décision et optimisation à la compilation pour des architectures massivement parallèles.

Thèse soutenue à Saclay
le 14 décembre 2015

devant le jury composé de :

Henri-Pierre Charles
Directeur de recherche au CEA Grenoble) / *Président*

Raymond Namyst
Professeur à l'Université de Bordeaux / *Rapporteur*

Gilles Sassatelli
Directeur de recherche CNRS, Montpellier / *Rapporteur*

Gabriel Antoniu
Directeur de recherche à l'INRIA, Rennes / *Examinateur*

Guy Gogniat
Professeur à l'Université de Bretagne Sud / *Directeur de thèse*

Loïc Cudennec
Ingénieur chercheur au CEA Saclay / *Encadrant de thèse*

Table des matières

Introduction

La révolution technologique de ce siècle doit en grande partie son effervescence à l'informatique. Du téléphone intelligent à la voiture autonome en passant par la console de jeux, l'informatique est rendue de plus en plus indispensable dans le quotidien de chacun. Si cette science évolue aussi rapidement dans le marché grand public, elle en fait de même pour des domaines plus critiques tels que la prévision météorologique, la simulation de phénomènes physiques ou encore la manipulation et le traitement de grandes quantités de données.

Les exigences de ces domaines applicatifs ne cessent d'augmenter ce qui nécessite une très grande puissance de calcul et une capacité de stockage importante. La *loi de Moore*, résultat d'une extrapolation empirique, prévoit une multiplication par deux de la puissance de calcul tous les 18 mois. Elle a été vérifiée pendant plusieurs années grâce à la croissance de la technologie des semi-conducteurs qui a permis de doubler le nombre de transistors sur une même surface de *silicium* tous les 24 mois au plus tard. Une des métriques utilisées pour la vérification de *la loi de Moore* est l'augmentation de la fréquence d'horloge qui caractérise la vitesse de traitement des systèmes. Cependant, cette évolution est rapidement limitée, en particulier pour les systèmes autonomes, par la consommation d'énergie et la dissipation thermique qui en découle.

Depuis 2004 la fréquence des processeurs ne peut plus suivre la courbe donnée par la *loi de Moore*. Il est donc question de chercher des solutions à basse consommation énergétique tout en garantissant les performances de calcul. Les multicœurs sont des systèmes parallèles sur puce qui permettent d'agréger la puissance de calcul de plusieurs cœurs afin d'atteindre de hautes performances. Depuis plus d'une dizaine d'années, les systèmes sur puce suivent une tendance dérivée des multicœurs qui consiste à implémenter plusieurs centaines voire des milliers de cœurs simples à basse fréquence sur une même puce. Ces systèmes sont appelés *manycœurs* et ils promettent un bon compromis entre la performance du calcul parallèle et le coût énergétique.

Parmi les manycœurs disponibles actuellement sur le marché, on peut citer la puce *MPPA* à 288 cœurs de *Kalray*, la famille de processeurs *GX* de *Tilera* qui atteint 72 cœurs, la puce *Xeon Phi* d'*Intel* avec 61 cœurs et la puce *Epiphany* à 72 cœurs d'*Adapteva*. Les réseaux de communication sur les manycœurs sont optimisés pour faciliter les échanges entre les nombreux cœurs de la puce. Les connexions courtes et rapides entre les éléments d'un manycœurs forment ce qui est appelé plus communément un *NoC (en anglais Network on Chip)*.

L'utilisation de la mémoire dans ce genre de systèmes est basée sur une approche

hiérarchique répartie où chaque niveau de mémoire représente un coût et une capacité différents. Cette approche est largement utilisée dans des systèmes à plus large échelle tels que les grilles de calculateurs.

En dépit des capacités physiques que peut offrir une architecture manycœurs, l'exploitation de la puissance de calcul disponible n'est pas toujours triviale. Du point de vue de l'utilisateur, la programmation des systèmes hautement parallèles peut être très complexe. Elle nécessite une gestion efficace de tous les aspects concurrentiels du calcul parallèle, notamment les conflits d'accès à la mémoire. Nous nous intéressons dans ces travaux particulièrement aux problèmes de la cohérence des données dans les architectures massivement parallèles. Pour des raisons de passage à l'échelle, chacun des cœurs du système a au moins un niveau de cache privé garantissant un accès physique direct aux copies locales des données.

Les accès aux données par plusieurs cœurs en parallèle mènent souvent à des problèmes de cohérence et de conflit d'accès. Ces problèmes sont gérés soit par le matériel, solution complexe et qui ne passe pas à l'échelle, ou bien par l'utilisateur, qui est amené à prendre des décisions sur les points de synchronisation et sur les accès atomiques dans son programme, ce qui n'est pas toujours facile à faire. Une autre solution, dite à *mémoire virtuellement partagée*, est de créer un espace d'adressage logique et global, partagé par toutes les mémoires réparties de l'architecture. Cette vision unifiée de l'espace de stockage sur une puce permet d'abord de réduire la charge de l'utilisateur puisque les accès à la mémoire sont gérés de manière transparente par le système. Elle permet également une flexibilité concernant les choix des mécanismes de gestion de la cohérence.

À notre connaissance il n'existe pas encore de plate-forme basée sur le paradigme de mémoire virtuellement partagée pour les systèmes manycœurs qui permet une gestion automatique et adaptative de la cohérence des données.

Objectifs de la thèse

L'objectif de cette thèse est d'étudier les modèles et protocoles de cohérence de données pour les architectures massivement parallèles. Deux principaux axes ont été définis dans le cadre de la thèse :

L'étude des protocoles de gestion de la cohérence pour les manycœurs : la gestion de la cohérence s'intéresse non seulement à la gestion des mises à jour des données mais aussi à la disponibilité des données qui relève du problème de la gestion du stockage sur la puce.

Le choix des protocoles : De notre étude des protocoles de cohérence et de gestion des données nous avons conclu qu'il n'existe pas un protocole unique qui soit performant pour l'ensemble des contextes applicatifs et des contextes d'exécution. Nous nous intéressons donc dans ce travail à la conception d'une plate-forme multi-protocolaire pour le choix et la configuration des protocoles de cohérence à la compilation.

La question de la cohérence étudiée dans le cadre de cette thèse prend en compte à la fois les contraintes liées au comportement de l'application et à la plate-forme d'exécution, ainsi que les objectifs de performances définis par l'utilisateur selon son domaine d'application. La plate-forme proposée prend les décisions sur le choix et la configuration des protocoles de cohérence pendant la compilation. Elle permet d'obtenir un meilleur compromis de performance selon les besoins d'un contexte particulier. Nous détaillons dans ce qui suit les différentes contributions proposées dans cette thèse pour répondre à cette question.

Contributions de la thèse :

Les contributions de cette thèse sont divisées en deux grandes parties :

Protocoles de glissement de données : il s'agit d'une approche à caches coopératifs permettant de gérer efficacement l'espace de stockage global au niveau de la puce. Les accès à la mémoire extérieure étant très coûteux, le glissement de données autorise la migration des données entre les caches des cœurs afin d'éviter leur éjection hors puce. Une extension de cette approche basée sur le modèle masse-ressort est proposée afin de gérer automatiquement les déplacements des données. Le modèle permet de décider de la distance et de la trajectoire de tous les mouvements d'une donnée.

Plate-forme de compilation pour la configuration des protocoles : la plate-forme de compilation proposée permet de prendre des décisions sur le choix des protocoles de cohérence pendant la compilation. Cette prise de décision prend en compte le comportement de l'application et la cible d'exécution. À chaque application est associée une combinaison d'instances de protocoles qui permettent de gérer les accès mémoire de l'application.

Analyse statique du code : cette étape permet de caractériser le comportement de l'application en se basant sur les schémas d'accès aux données partagées. L'analyse de ce comportement permet d'éliminer certains choix de protocoles de cohérence et d'en favoriser d'autres. Chaque protocole de cohérence est préalablement caractérisé en fonction de ses performances et de son comportement ce qui permet d'associer un type de protocole à un schéma de comportement des applications. Cette première phase de décision aboutit à des choix préliminaires de protocoles principalement en fonction du comportement de l'application.

Configuration des protocoles de cohérence : cette phase de la plate-forme de compilation consiste à raffiner le choix des protocoles en fonction des contraintes posées par l'architecture d'exécution. Elle permet d'instancier les paramètres des protocoles choisis pour mieux les adapter aux objectifs de performances souhaités. Chaque protocole de cohérence peut être configuré de plusieurs manières selon l'espace de définition de ses paramètres. L'espace des solutions exploré dans cette phase augmente exponentiellement avec le nombre de solutions par protocole et

la taille de la trace mémoire étudiée. Afin de surmonter la complexité d'un tel processus de décision, nous proposons une approche d'optimisation basée sur un algorithme génétique itératif. Le choix de cette approche a l'avantage de réduire le temps de décision en garantissant une bonne qualité de solution. Chaque solution parcourue par l'algorithme génétique est évaluée et classée selon son degré de satisfaction des objectifs de performance. Le mécanisme de configuration proposé permet de chercher un compromis entre plusieurs objectifs de performances. L'évaluation des solutions parcourues est basée sur des modèles haut niveau qui génèrent différentes métriques de performances. Le paragraphe suivant décrit les modèles d'évaluation proposés dans le cadre de cette thèse.

Évaluation des performances des protocoles de cohérence : l'évaluation est une étape nécessaire au processus de décision. Elle consiste à estimer le surcoût associé au protocole de cohérence adopté. Plusieurs solutions sont évaluées lors du processus de configuration ce qui rend difficile l'utilisation d'une modélisation de bas niveau. Nous proposons deux modèles d'évaluation. Un premier modèle analytique qui mesure les performances des caches (ex. le nombre de défauts de cache) et les statistiques des communications au niveau du *NoC*. Ce modèle ne donne pas d'informations temporelles de performances. Un deuxième modèle est proposé qui, associé à un simulateur *TLM* de *NoC*, offre la possibilité d'estimer les temps d'accès à chaque donnée, les pénalités de communications et la contention dans le réseau.

La suite de ce manuscrit est organisée comme suit : Le chapitre 1 présente un état de l'Art général de la thèse portant sur les architectures réparties sur puce, leurs modèles de programmations ainsi que les mécanismes de gestion de la cohérence associés. Le chapitre 2 présente les différentes techniques de gestion des données dans les architectures réparties, il décrit également deux propositions de protocoles de glissement de données. Le chapitre 3 décrit les résultats d'évaluation des performances des protocoles proposés par rapport à des protocoles de l'état de l'art. Le chapitre 4 est dédié à une présentation générale de la plate-forme de compilation proposée, avec une focalisation sur la brique d'analyse statique de l'application pour l'aide à la décision dans le choix des protocoles. Le chapitre 5 décrit les modèles temporels d'évaluation proposés dans le cadre cette thèse qui permettent d'évaluer les performances du système obtenues avec les différents protocoles de cohérence étudiés. Le chapitre 6 décrit le processus de configuration proposé basé sur une méthode de recherche génétique qui garantit d'obtenir une meilleure qualité de solution dans des temps d'exécution raisonnables et drastiquement réduits par rapport à une méthode de recherche exhaustive. La conclusion décrit les différents résultats de la thèse ainsi que les perspectives envisagées pour la continuité de ces travaux.

Les travaux présentés précédemment ont fait l'objet de publications et de communications dans des conférences nationales et internationales.

Liste des publications

Conférences internationales avec acte

[DCG13] Safae Dahmani, Loïc Cudennec and Guy Gogniat. **Introducing a data sliding mechanism for cooperative caching in manycore architectures.** In *18th International Workshop on High-Level Parallel Programming Models and Supportive Environments, in conjonction with IPDPS*, Boston, Massachusetts, USA, May 2013.

[DCLG14] Safae Dahmani, Loïc Cudennec, Stéphane Louise and Guy Gogniat. **Using the Spring Physical Model to Extend a Cooperative Caching Protocol for Many-Core Processors.** In *8th International Symposium on Embedded Multicore/Manycore SoCs (MCSoc)*, Aizu-Wakamatsu, Fukushima-Ken,Japan, September 2014.

[CDCGS15] Hamza Chaker, Safae Dahmani, Loïc Cudennec, Guy Gogniat and Martha Johanna Sepùlveda. **Cycle-based Model to Evaluate Consistency Protocols within a Multi-protocol Compilation Toolchain**. In *Proceedings of the International Workshop on Code Optimisation for Multi and Many-cores (COSMIC 2015), in conjunction with CGO*, San Francisco, California, USA, February 2015.

[MDCGS16] Cédric Maignan, Safae Dahmani, Loïc Cudennec, Guy Gogniat and Martha Johanna Sepùlveda. **Network Contention-aware Method to Evaluate Data Consistency Protocols within a Compilation Toolchain** . soumis à Date 2016.

Communications sans acte et présentation orale

[DC12] Safae Dahmani. **Exploitation de motifs d'accès mémoire et amélioration de la coopération des caches dans les architectures manycœurs.** In *Journée Logiciels Embarqués et Architectures Matérielles du GDR SoC-SiP*, Paris, France, November 2012.

[DAHMANI13-1] Safae Dahmani. **Consistency Protocol Decision at Compile-time for Multi-protocol Distributed Shared Memory Systems.** In *Journée*

Logiciels Embarqués et Architectures Matérielles du GDR SoC-SiP, Paris, France, June 2013.

[DAHMANI13-2] Safae Dahmani. **Adaptive cooperative caching for many-cores systems.** In *Proceedings of the 9th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES, HIPEAC)*, Fuji, Italy, July 2013.

[DCG14-2] Safae Dahmani, Loïc Cudennec, Guy Gogniat. **Consistency Protocol Decision at Compilate Time for Multi-protocol Distributed Shared Memory Systems.** *Conférence en Parallélisme, Architecture et Système (COMPAS)*, Neuchatel, Suisse, 2014.

[DCCG15] Safae Dahmani, Sergiu Carpov, Loïc Cudennec, Guy Gogniat. **Aide à la décision pour le choix et le paramétrage de protocoles de cohérence des données.** *Conférence de la Société Française de Recherche Opérationnelle et Aide à la Décision (ROADEF).*, Marseille, France, Février 2015.

Chapitre 1

Contexte général de la thèse

L'évolution technologique des systèmes parallèles nous offrent aujourd'hui de plus en plus de puissance de calcul. Cependant, les performances en terme de mémoire et de consommation énergétique n'évoluent pas à la même vitesse. Afin de profiter au mieux des performances de calcul de ces systèmes, en dépit de ces limitations, il est important de pourvoir les programmer de manière efficace et simple. De nombreux travaux ont donné lie à différents modèles de programmation avec chacun des avantages et des inconvénients. Nous nous intéressons dans ces travaux au modèle de programmation à mémoire virtuellement partagée qui permet de gérer les accès mémoire de manière transparente et uniforme. La gestion des accès concurrents à la mémoire dans ce modèle de programmation repose sur des mécanismes et protocoles de cohérence des données. Nous nous arrêtons dans le cadre de cette thèse sur les problèmes liés à la gestion de la cohérence des données pour des architectures massivement parallèles telles que les systèmes sur puce de type manycœurs. Ce chapitre introduit le sujet avec une étude de l'état de l'art des systèmes répartis sur puce, des modèles de programmations et des techniques de gestion de la cohérence pour différents types de systèmes répartis.

La suite de ce chapitre est organisée comme suit : la section 1.1 présente l'historique de l'évolution des systèmes intégrés sur puce en se focalisant sur les technologies des réseaux de communication et les technologies mémoire, la section 1.2 présente une discussion sur les modèles de programmation existants et la section 1.3 porte sur les modèles et mécanismes de gestion de la cohérence des donnés.

1.1 Des multicœurs vers les manycœurs

Les systèmes intégrés sur puce connaissent une très forte croissance, si ce n'est la plus rapide dans le marché des ordinateurs. Ils couvrent une large partie des applications grands publics (ex. électroménager, consoles de jeux, téléphonies, automobiles). Ils sont également présents dans des domaines plus pointus tels que l'accélération des calculs scientifiques (ex. prévisions météorologiques, simulations physiques, analyse financière) et pour le contôle de certains systèmes complexes (ex. industrie avionique et sous-marine).

Comparés aux ordinateurs de bureau et aux systèmes de serveurs, les système embarqués offrent un plus large éventail de puissances de calculs et de consommations énergétiques. Le marché des systèmes embarqués propose une offre qui va des petits processeurs 8-bit et 16-bit qui coûtent moins d'un euro et du microprocesseur *32-bit* capable de traiter *500 MIPS* qui coûte quelques dizaines d'euros aux systèmes d'une grande puissance de calcul capables d'exécuter des billions d'instructions par seconde et dont le coût est de l'ordre de centaines (*Tilera Gx* [Ram11a]) voire milliers (*Intel XeonPhi 7100* [Chr14], *Kalray MPPA-256* [dDAB$^+$13]) d'euros.

Le marché des systèmes sur puce est aujourd'hui mené par trois tendances technologiques qui évoluent à différentes vitesses : technologies des circuits intégrés 1.1.1, technologies des réseaux de connexion 1.1.2 et finalement les technologies mémoire 1.1.3

1.1.1 Évolution des circuits intégrés vers les manycœurs

Le microprocesseur *Intel 4004* est l'un des tous premiers fabriqués par *Intel* en 1971. Il s'agit d'un calculateur intégrant 2300 transistors *MOS* et cadencé à une fréquence ne dépassant pas les *740 kHz*. Le développement qu'a connu le marché des processeurs par la suite, doit son essor à l'augmentation de la densité des transistors intégrés et à la diminution de la surface de gravure.

Gordon Moore (l'un des fondateurs d'Intel) a constaté que la complexité des semiconducteurs doublait tous les deux ans à coût constant depuis leur apparition en 1959. Il prévoyait la continuité de cette croissance exponentielle sur les années à venir. Cette extrapolation empirique est connue depuis 1965 sous le nom de la *loi de Moore*.

En 1975, quatre ans plus tard, la *loi de Moore* a fait l'objet d'une réévaluation. Elle postule que le nombre de transistors sur une puce en silicium va doubler tous les ans. Bien qu'il ne s'agit pas d'une loi physique, la *loi de Moore* a été vérifiée jusqu'en 2001, la densité des transistors doublait tous les 18 mois. En conséquence, les ordinateurs sont devenus moins coûteux, moins encombrant et surtout plus puissants.

Plusieurs versions inspirées de la *loi de Moore* ont été diffusées. La version sur la fréquence d'horloge qui semblait suivre la même évolution n'est plus vérifiée depuis 2004. Il est actuellement de plus en plus difficile d'augmenter la fréquence pour des raisons liées à la dissipation thermique et à la consommation énergétique.

Si récemment la densité des transistors augmente chaque année de 35%, en quadruplant à peu près tous les 4 ans, l'évolution au niveau de la taille de la puce reste moins prédictible mais plus lente (de 10% à 20% par an). En conséquences, le nombre de transistors sur puce augmente de 40% à 55% par an (i.e. doublement tous les 18 à 24 mois) [HP12]. Cette tendance est touchée sur le marché par des processeurs parallèles, appelés multicœurs, regroupant plusieurs unités de calcul simples et à basse fréquence, Cette approche permet de mieux gérer la dissipation thermique, directement liée à la fréquence d'horloge, tout en garantissant une grande puissance de calcul.

Face à la demande du marché, cette tendance continue à attirer les fondeurs en leur permettant d'améliorer les performances des systèmes sans augmenter leur prix. Une nouvelle génération de systèmes sur puce massivement parallèles voit le jour : les manycœurs [Cor11]. Les architectures manycœurs promettent une grande capacité de

Métrique	Bus	Anneau	Maille 2D	Tore 2D	Hypercube	Arbre	Connexion complète
Distance maximale (Nb de pas)	1	32	14	8	6	11	1
Coût (Nombre de liens)	1	128	176	192	256	384	2080

TABLE 1.1 – Topologies de réseau

calcul à basse consommation énergétique grâce à l'intégration de plusieurs dizaines (voir plusieurs centaines) de cœurs simples sur la même puce. Ce type d'architectures repose sur une organisation mémoire (cache hiérarchique) et une structure de communication (réseau sur puce) optimisées pour accélérer les traitements parallèles sur la puce.

Les deux sections suivantes décrivent les tendances technologiques au niveaux des réseaux de communication et de la mémoire.

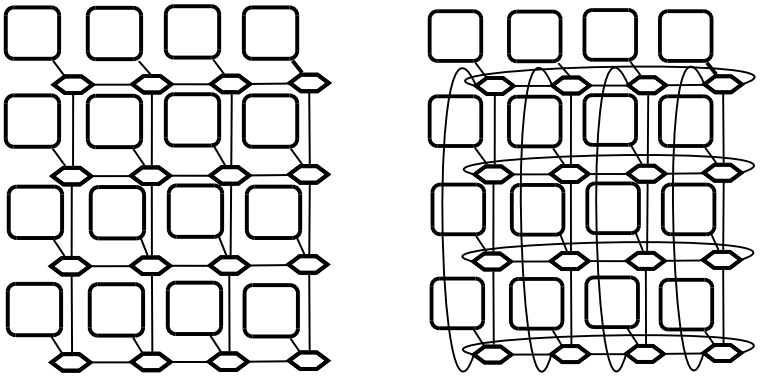
1.1.2 Technologies des réseaux de communication sur puce

Dans les systèmes sur puce à un grand nombre d'unités de calcul, il est important d'avoir une meilleure communication entre elles. La conception d'un système de communication sur puce est contrainte par la surface de la puce et le coût de développement ce qui limite ses performances. Le passage à l'échelle dans ce genre de réseaux, objectif majeur des systèmes sur puce, dépend fortement des performances en latence et en bande passante. Ce dernier est caractérisé par la structure de connexion entre les cœurs (i.e. topologie) et par le système de communication associé (ex. routage, arbitrage).

Les réseaux sur puce, dits *NoC* (correspondant à *Network-on-Chip*), est un type particulier de réseaux dédié à la connexion entre les unités de calcul et les périphériques du systèmes (ex. registres, mémoires caches) au sein d'une microarchitecture. Ils sont conçus pour connecter efficacement plusieurs dizaines à plusieurs centaines de nœuds à une distance de l'ordre de quelques centimètres. Parmi les *NoCs* existants sur le marché actuellement, nous retrouvons le *NoC* d'*Intel Teraflops* [VHR+07] reliant 80 cœurs simples ; celui d'*Intel Single-Chip Cloud Computer(SCCC)* [DBMO11] regroupant 48 cœurs de l'architecture IA-32 ; et le réseau de la gamme *Tilera TILE-Gx* [Ram11b] connectant jusqu'à 200 cœurs (2013). Ces *NoC* atteignent des bandes passantes de 256 GBps pour les puces *d'Intel* et 200 GBps pour le processeur *Tilera Tile-Gx*.

Une topologie de *NoC* permet d'établir un lien de communication entre n'importe quels deux points distants du système, soit de façon directe (connexion physique directe) ou à travers des points intermédiaires (point-à-point). Elle est constituée de l'ensemble des liens physiques, des commutateurs et des nœuds d'une architecture. De très nombreuses topologies plus ou moins complexes ont été proposées pour les différents types de systèmes. La complexité d'une topologie est dûe essentiellement au nombre de liens physique qu'elle engendre, ce qui reflète son coût, et à la distance maximale (en nombre de pas réseau) entre deux éléments distants dans le système. Le tableau 1.1 présente une comparaison entre un ensemble de topologies sur la base de leur coût et de leur performance (distance maximale) pour une architecture à 64 cœurs.

Le choix d'une topologie peut avoir un grand impact sur les performances d'un système surtout lorsque le nombre d'éléments connectés devient important et la latence est critique (comme dans le cas des manycœurs).



(a) (b)

FIGURE 1.1 – Exemples de topologies de réseaux sur puce (1.1a) Maille, (1.1a) Tore.

Contrairement aux réseaux de connexion à grande échelle (i.e. LAN, WAN), les topologies *NoC* convergent vers des formes régulières et structurées (ex. maille(figure 1.1a), tore(figure 1.1b)) afin de faciliter le routage, l'acheminement des paquets ainsi que le passage à l'échelle (faciliter de dupliquer un motif simple).

Des travaux récents se sont intéressés à la conception de circuits intégrés tridimensionnels [BAB+06] dans l'optique de contourner les contraintes liées à la surface de gravure de la puce. Ce type d'architectures intègre des liaisons verticales fines pour véhiculer les données, ce qui permet de palier au souci de ralentissement des communications horizontales dans les puces denses (ie. nombre important d'intermédiaires entre deux nœuds distants). Une structure tridimensionnelle permet également de surmonter le problème de dissipation thermique en offrant une meilleure répartition des éléments du système.

1.1.3 Technologies mémoire

À l'instar des réseaux de connexion, les technologies mémoires ont connues une importante évolution au fils du temps. Différentes technologies ont été développées afin de répondre aux besoins de l'utilisateur et de suivre la croissance du niveau de parallélisme. Les choix liés à la mémoire occupent de plus en plus d'importance dans le cycle de conception des systèmes parallèles tels que les manycœurs.

Les trois principales familles de technologies mémoire, citées ci-dessous, représentent chacune des performances et des coûts différents :

Technologie mémoire DRAM : La capacité de la mémoire *DRAM* par puce a démontré récemment une évolution de 25% à 40% par an, soit un doublement tous les deux à trois ans. Notons que le taux de croissance des mémoires *DRAM* a diminué durant les 10 dernières années 1.2. Cette limitation est dûe à la difficulté de fabrication

Année	Taux de croissance par an	Impact sur la capacité des mémoires *DRAM*
1990	60%	×4 tous les 3 ans
1996	60%	×4 tous les 3 ans
2003	40 à 60%	×4 tous les 3 à 4 ans
2007	40%	×2 tous les 2 ans
2011	25 à 40%	×2 tous les 2 à 3 ans

TABLE 1.2 – Variation du taux de croissance de la technologie *DRAM* au fil des années. À partir de 2011 cette croissance est ralentie. Actuellement, il devient plus difficile de prévoir une croissance sur les 5 à 7 ans à venir.

de plus petites cellules *DRAM*. Plusieurs alternatives à la technologie *DRAM* ont été proposées [Kim12].

Technologie mémoire FLASH : La capacité des mémoires *FLASH* a augmenté récemment de 50% à 60% par an en doublant moyennement tous les 2 ans. Comparée à la technologie *DRAM* les mémoires *FLASH* sont 15% à 20% fois moins chères.

Technologie mémoire à disque magnétique : Le stockage à disque magnétique a connu une évolution de densité de 30% à 100% par an entre 1990 à 1996. Cette évolution a été limitée depuis 2004 à 40%. Le prix par bit de la technologique mémoire magnétique est 15% à 25% moins cher que celui de la mémoire *FLASH*. Étant donné le ralentissement de croissance des mémoire *DRAM*, les mémoires à disque sont actuellement 300 à 500 fois moins chères que les *DRAMs*. La technologie à disque magnétique est plus utilisée à l'échelle des serveurs et des entrepôts de données.

Face à des limitations technologiques et afin de mieux profiter des compromis coût-performance des différentes technologies mémoire, une nouvelle solution est proposée ; il s'agit de la mémoire *hiérarchique*. Cette solution permet également de mieux profiter du principe de *localité* basé sur l'idée que les données ne sont pas accédées de façon uniforme durant toute l'exécution. Deux types de *localité* sont distingués : *localité spatiale* et *localité temporelle* :

Définition 1 (Localité spatiale)**.** La *localité spatiale* signifie que les blocs de données prochainement utilisés sont probablement situés à proximité des données en cours d'utilisation.

Définition 2 (Localité temporelle)**.** La *localité temporelle* signifie qu'un bloc de données récemment accédé sera plus probablement réutilisé dans un futur proche.

D'un autre coté la différence entre le coût et les performances (vitesse d'accès, consommation d'énergie) entre les technologies présentées plus haut emmène naturellement vers une structure hiérarchique à plusieurs niveaux de stockage. La figure 1.2 présente la structure hiérarchique typique dans un téléphone mobile. Les quatre niveaux de mémoire représentent différentes performances en terme de vitesse d'accès et de capacité de stockage.

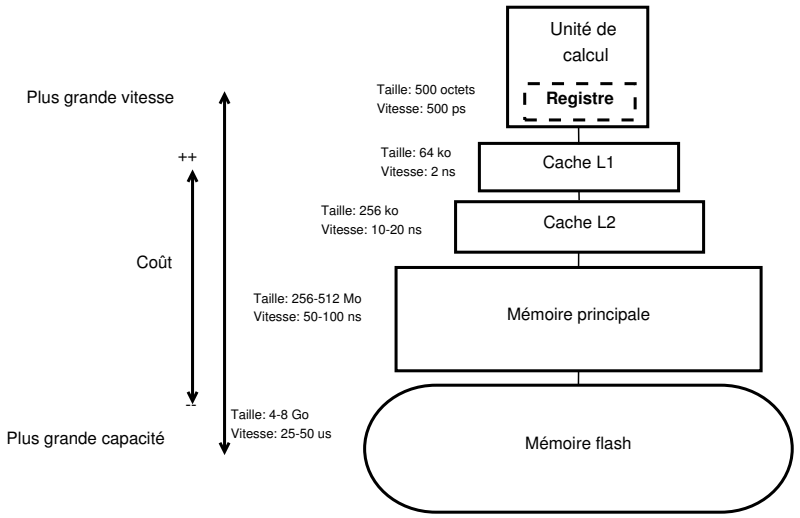


FIGURE 1.2 – Structure typique d'une mémoire hiérarchique présente dans un téléphone mobile. Les mémoires dans les couches inférieures sont plus lentes mais plus grandes et moins coûteuses.

L'importance de l'organisation en plusieurs couches doit son essor à l'augmentation de l'écart de performance entre les processeurs et la mémoire. Elle a permis aux processeurs haut de gamme récents d'avoir une plus grande bande passante (Intel i7, 409,6 Go/s) et ainsi d'accélérer leur vitesse de calcul. L'amélioration de la bande passante est atteinte grâce à l'utilisation de plusieurs niveaux de caches distincts (L_1 et L_2) en plus de la séparation entre cache d'instructions et cache de données dans le premier niveau de cache. En revanche, la bande passante de la mémoire principale *DRAM* est drastiquement plus faible (25 Go/s) ce qui reflète la limitation technologique à ce niveau de la hiérarchie.

Dans la conception des hiérarchies mémoire, les architectes se concentrent sur l'optimisation du temps d'accès moyen déterminé par le temps d'accès au cache, le taux de défauts de cache et la pénalité d'un défaut de cache (Temps d'accès à la mémoire principale).

La consommation d'énergie des accès aux caches représente entre 25% et 50% de la consommation totale d'un système ce qui n'est négligeable. La consommation d'énergie devient aux cotés du temps d'accès une considération majeure pour les architectures modernes.

1.1.4 Discussion

Les technologies des microprocesseurs, des mémoires, des réseaux progressent à de multiples vitesses. Les principales métriques de cette différence d'évolution sont la *bande passante* (ou le débit) et la *latence*. La *bande passante* correspond à la quantité de traitement réalisée dans un temps limité (ex. Mégabytes/s pour les transferts disques). La *latence* est définie comme le temps écoulé entre le début et la fin d'un événement

(ex. millisecondes pour un accès au disque). Les performances des microprocesseurs et des réseaux ont vu une grande augmentation : 10000 à 25000 fois pour la *bande passante* et 30 à 80 fois au niveau de la latence. En dépit de l'augmentation de la capacité des mémoires et des disques, leur performance en *bande passante* a augmenté de 300 à 1200 fois tandis que la latence n'a pas dépassé 6 à 8 fois. Ce déséquilibre dans l'évolution des performances des technologies mémoires et disques constitue un obstacle (en anglais connue sous l'appellation *Memory Wall*) pour les performances globales que peut atteindre un système.

Améliorer l'organisation de la mémoire (organisation de la *DRAM* en banc) ou ajouter des niveaux hiérarchies avec des mémoire caches plus larges n'est plus suffisant pour rejoindre la croissance exponentielle de la puissance computationnelle.

Une manière de surmonter cette difficulté technologique est d'avoir un modèle de programmation performant permettant de profiter au mieux des capacités de calcul d'un système en réduisant l'effet d'un tel écart avec la mémoire.

La section suivante présente les modèles de programmation les plus utilisés dans la programmation parallèle en expliquant les avantages et les inconvénients de chaque modèle.

1.2 Programmation des systèmes parallèles sur puce

Selon le modèle de machine *Von Neumann* (contrairement au modèle de *Harvard*, qui sépare mémoire de données et celle du programme), une instruction peut définir une opération arithmétique, une adresse d'une donnée à lire ou à écrire dans la mémoire ou bien l'adresse de la prochaine instruction à exécuter.

S'il est possible d'écrire un programme à partir de ce modèle en utilisant le langage machine, il reste plus difficile pour des programmes complexes par-ce-qu'il faut garder la trace de plusieurs millions d'adresses mémoires et organiser l'exécution de plusieurs milliers d'instructions. La technique de conception modulaire a été introduite afin de faciliter la programmation des applications complexes en les découpant en plusieurs modules simples. Les modules sont organisés en termes de niveaux d'abstractions haut niveau tels que les structures de données, les boucles itératives et les procédures. Des abstractions comme les procédures permettent une exploitation plus facile du concept de modularité en autorisant la manipulation des objets sans s'intéresser à leur structure interne. La programmation parallèle a augmenté la complexité de l'approche modulaire avec le grand nombre d'instructions à exécuter en parallèle et les millions d'interactions à gérer entre les processeurs.

Ce chapitre explique les différentes techniques de la programmation parallèle guidées par les exigences de l'utilisateur comme la transparence du type d'architecture et du réseau de communication, la facilité d'utilisation, la portabilité en plus des performances à l'exécution.

Granularité	Objet	Parallélisé par
Très fine	Instruction	Processeur
Fine	Boucle/Bloc d'instructions	Compilateur
Moyenne	Fonction	Programmeur
Large	Processus lourd	Programmeur

TABLE 1.3 – Niveau de parallélisme en fonction de la taille de granularité de code

1.2.1 Les niveaux de parallélisme

Le parallélisme apparaît à différent niveaux matériels et logiciels : signal, circuit, module et au niveau système. Au plus bas niveau, les signaux circulent en parallèle sur les chemins de données. À un niveau un peu plus haut appelé plus communément *Parallélisme d'instructions* où plusieurs unités fonctionnelles opèrent parallèlement. Par exemple, un processeur comme le *Pentium Pro* a la capacité de traiter 3 instructions simultanément.

Plusieurs systèmes chevauchent les activités des processeurs et des interfaces d'entrée/sortie, en autorisant par exemple l'accès au disque d'un utilisateur pendant le traitement d'une instruction d'un autre utilisateur. D'autres systèmes utilisent la technique d'entrelacement des accès mémoire où les blocs mémoire sont accédés en parallèle afin d'accélérer le temps d'accès. Un autre niveau de parallélisme est celui où plusieurs processeurs ou un réseau d'ordinateurs connectés (i.e. *grappes*) fonctionnent comme une seule machine. Les deux premiers niveaux de parallélisme (niveau signal et niveau circuit) sont réalisés au niveau matériel. Les deux autres techniques sont réalisées implicitement ou explicitement en se basant sur des techniques logicielles. Les niveaux de parallélisme peuvent aussi se baser sur les bouts de code parallélisables. Le tableau 1.3 explique les différents types de granularité de code. L'objectif de toutes les formes de parallélisme basées sur cette granularité est d'améliorer les performances en cachant la latence liée aux opérations coûteuses telles que les accès en mémoire. A chaque fois qu'une opération lente se produit, une autre opération doit être prête à s'exécuter. Les deux formes de parallélisme à grain fin dans une application sont gérées respectivement par le matériel ou par le compilateur. Les deux autres granularité sont prises en charges par le programmeur.

Deux principales familles de parallélisme sont distinguées : (1) La parallélisation *implicite* ; une approche utilisée par les langages de programmation parallèle et les compilateurs de parallélisation. L'utilisateur ne doit spécifier ni contrôler l'ordonnancement des calculs ni le placement des données. (2) La parallélisation *explicite* ; où l'utilisateur est responsable de la parallélisation de son programme. Il doit définir la décomposition des tâches, la structure de communication entre les différentes tâches.

La section suivante décrit des principaux modèles de programmation issues des différentes techniques de parallélisme discutées dans cette section.

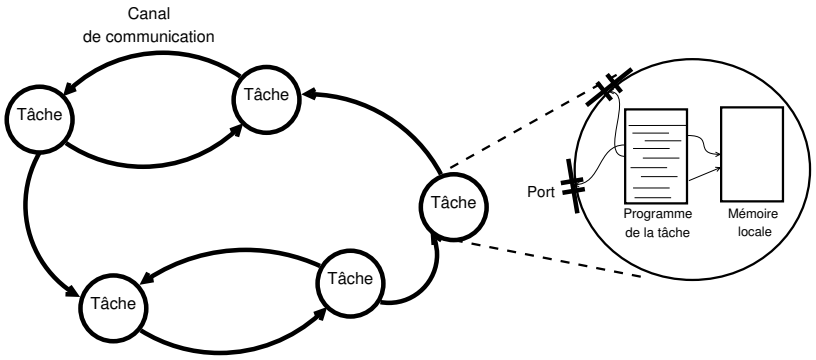


FIGURE 1.3 – Un exemple simplifié de programme parallèle. La figure montre un ensemble de tâches qui communiquent entre elles via des canaux de communication. Une tâche englobe un programme séquentiel et sa mémoire locale et définit un ensemble de ports pour échanger avec son environnement.

1.2.2 Classification des modèles de programmation

Un modèle de programmation constitue le pont entre le modèle intuitif de l'application établi par le développeur et l'implémentation de cette application sur une architecture particulière. Il consiste en un ensemble d'algorithmes permettant de mettre en place le parallélisme selon la plate-forme matérielle cible. Les trois principaux modèles de programmation parallèles sont : le modèle à mémoire partagée, le modèle à mémoire répartie virtuellement partagée et le modèle à mémoire répartie (par passage de message).

Modèles par passage de message

Le modèle de programmation par passage de message permet d'écrire des programmes parallèles pour des systèmes à mémoire répartie. Il est constitué de bibliothèques fournissant les routines nécessaires à l'initialisation et la configuration de l'environnement de passage de messages aussi bien que les routines de transmission et de réception des paquets de données. Actuellement, les deux systèmes de passage de messages les plus connus sont *PVM (Parallel Virtual Machine)* développé par le *OAK Ridge National Laboratory* [Gei94] et le *MPI(Message Passing Interface)* défini par le *MPI Forum* [GLDS96].

La figure 1.3 décrit le schéma de communication entre un semble de tâches réalisant un calcul. Dans le modèle par passage de message les tâches (Définition 3) partagent des canaux de communication (Définition 4). Le transmetteur envoie son message en utilisant une routine d'envoi, ensuite une routine de réception est utilisée au niveau de la destination. Les tâches ne partagent que les canaux et pas les données. Il est donc pas possible d'accéder simultanément à une même donnée par différentes tâches. Les problèmes de synchronisation des accès aux ressources n'est pas abordé dans cette approche.

Définition 3 (Tâche)**.** Une tâche correspond à un programme séquentiel et sa mémoire locale (une machine Von Neumann virtuelle) en plus des ports d'entrée et de sorties constituant son interface de communication avec son environnement.

Définition 4 (Canal de communication)**.** Un canal de communication est le mécanisme qui assure le lien de communication entre les tâches d'un programme. A chaque canal de communication est associé un ensemble d'adresses correspondant à des emplacements mémoire différents. Les accès à ces derniers sont gérés à l'aide d'un système d'accès offrant des routines de lecture et d'écriture que les tâches utilisent pour recevoir ou transmettre des données.

Il existe plusieurs implémentations de *MPI* pour les différents systèmes tels que les clusters et les multiprocesseurs à mémoire répartie. Il est de ce fait plus facile de porter les applications sur de nouvelles plate-formes sans avoir à les réécrire de nouveau. Du point de vue de l'utilisateur, les objectifs de portabilité et de transparence de l'architecture et du réseau de connexion sont donc satisfaits grâce à *MPI*.

Cependant, l'inconvénient de ce modèle basé sur le passage de message est que l'utilisateur est toujours responsable de la majorité des tâches de parallélisation. Une grande partie de l'effort de programmation est réservé à la gestion de la communication, de la synchronisation entre processus, de la distribution des données, placement des processus et les entrées/sorties des structures de données. Si le programmeur n'a pas de support particulier sur ces tâche, il sera donc difficile pour lui de bien exploiter le parallélisme.

Modèle orienté parallélisme de données (Flot de données)

Le modèle orienté parallélisme de données permet d'exploiter le concurrence dérivée de l'application d'une même opération sur de multiples données (ou éléments d'une structure de données). Un programme est donc constitué d'une séquence d'opérations. La granularité la plus fine d'un programme correspond à une seule opération. Les compilateurs exigent que l'utilisateur fournisse des informations sur le placement des données. Ces choix permettent de déterminer la localité ce qui impacte les performances d'une telle approche. Par exemple pour effectuer $A = B + C$ il est possible de ne pas avoir besoin de communications si toutes les données sont stockées localement (meilleur cas), mais il est possible aussi de produire une communication pour chaque accès à une donnée dans le cas inverse.

Cependant, il n'est pas nécessaire de spécifier explicitement toutes les communications ; elles sont déduites automatiquement par le compilateur du schéma de décomposition défini par le programmeur.

La faiblesse du modèle orienté parallélisme de données est qu'il n'est pas adapté à tous les types d'applications ce qui l'a empêché d'être plus généralisé.

La méta-programmation par modèle

Une alternative à l'utilisation du modèle par passage de messages est d'offrir un ensemble d'abstractions haut niveau basée sur des modèles. Un modèle correspond à

une instanciation d'un paradigme de programmation spécifique (définition 5) avec les primitives de contrôle et de communication liée à l'application.

Un grand effort de programmation est dépensé dans la définition des routines liées au paradigme et pas à l'application. L'existence d'un modèle efficace peut réduire drastiquement le temps de développement et ainsi simplifier le travail du programmeur. L'utilisation de cette technique a démontré des résultats intéressants [GVL10] en permettant à l'utilisateur de concentrer son effort sur la spécification d'une interface adaptée au paradigme. Le point fort de la programmation par modèle est qu'elle est générique. Par conséquent, elle peut s'implémenter en dessous d'un autre modèle de programmation tel que le Modèle par passage de messages.

Définition 5 (Paradigme de programmation)**.** Un paradigme de programmation est un ensemble d'algorithmes encapsulant les informations sur les données utiles et les schémas de communication.

Modèle à mémoire partagée

Contrairement au modèle par passage de messages (mémoire répartie) ce modèle est inspiré des architectures à mémoire partagée. Tous les processeurs accèdent à une mémoire physiquement partagée où chaque donnée a une adresse unique. Les paradigmes basés sur cette approche les plus connus sont *POSIX (PThreads)* [NBF96] et *OpenMP* [Ope97, DE98].

POSIX (PThreads) : c'est le standard de la programmation multithreads qui est venu pour unifier une interface de programmation utilisée par tous les acteurs industriels. L'interface *PThreads* définit les routines de gestion des tâches (create, detach, join), d'exclusion mutuelle (create, destroy, lock, unlock), les variables de condition et de synchronisation (verrous de lecture/écriture, barrières). Ces outils mis à disposition de l'utilisateur lui donne beaucoup de contrôle sur le fonctionnement des tâches. *PThreads* est donc une interface bas niveau où le programmeur met plus d'effort à gérer les tâches qu'à gérer la conception logique haut niveau de l'application.

OpenMP : En 1997 *OpenMP* [DE98] a été proposé pour unifier les outils de programmation des systèmes à mémoire partagé. Ce modèle offre des extensions de langages comme Fortran, C, C++ . Il consiste à ajouter des annotations dans le code séquentiel pour indiquer comment le travail sera réparti entre les tâches d'exécution. Grâce à son efficacité pour ce genre de système *OpenMP* a été ensuite introduit pour les systèmes répartis [BME07]. Contrairement au modèle *PThreads*, le modèle *OpenMP* offre une approche plus abstraite. Il permet également une bonne expressivité au dépend de d'un degré plus élevé de complexité.

Modèle à mémoire répartie virtuellement partagée

Un autre modèle de programmation parallèle est celui basé sur le concept de mémoire virtuellement partagée (i.e. *Distributed Shared Memory (DSM)*). Il implémente une

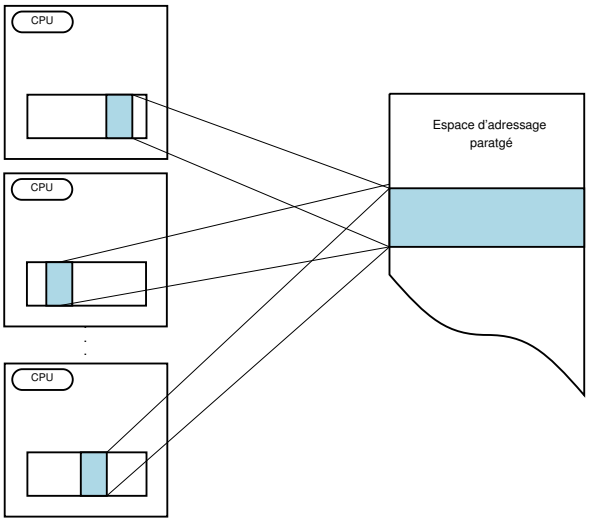


FIGURE 1.4 – Une mémoire répartie virtuellement partagé. Plusieurs copies d'une même données peuvent être chargées dans les mémoires locales associées aux différents processeurs. Chaque donnée a une référence unique dans la mémoire virtuelle partagée. Les processeurs ne voient que cette référence unique pour chaque accès à une données.

mémoire virtuelle partagée dans un environnement à mémoire répartie. De nombreux systèmes à mémoire partagée ont été proposés dans la littérature : Dash [LGL+90], Ivy [HGC02], Linda [NL91], Memnet [DSF88], Munin [BCZ90], Shiva [KCDZ94a].

Les accès aux données s'effectuent à travers un espace d'adressage commun donnant l'illusion aux processeurs d'une mémoire physique partagée [NL91]. Cet espace d'adressage est accessible grâce à des opérations d'écritures et de lecture ordinaires. Contrairement au modèle par passage de messages, il n'y a pas besoin de connaître l'emplacement d'une donnée pour pourvoir la charger, le système s'occupe de localiser la donner et la charger dans la mémoire du demandeur. Les données peuvent être dupliquées dans les mémoires réparties afin de favoriser la parallélisation des calculs 1.4. Or, cette facilité de manipulation des données nécessite une bonne gestion des synchronisations des accès et de la cohérence des données par le système.

1.2.3 Discussion

Les modèles de programmation décrits dans cette section peuvent être classés selon deux approches. Dans la première approche, le programmeur est en charge de structurer et de gérer les tâches de communication et de contrôle en s'appuyant sur quelques primitives fournies par le modèle. La deuxième approche propose plus de support au programmeur en terme de contrôle et de communication. Le rôle du programmeur se réduit à définir les routines liées à l'application pour la partie calcul et génération de données. Le modèle à mémoire virtuellement partagée permet de faciliter la programmation des systèmes massivement parallèles en donnant l'illusion au programmeur d'avoir une seule mémoire physique. Les performances de ce modèle dépendent fortement de

sa capacité à gérer les partage des données entre un grand nombre de cœurs. Plusieurs techniques de gestion de la cohérence des données ont été proposées dans l'objectif de garantir la consistance des calculs parallèles.

Nous traitons dans ce qui suit l'état de l'art des modèles et protocoles de cohérence pour les systèmes sur puce.

1.3 Modèles et protocoles de cohérence pour les systèmes sur puce

Dans un modèle de programmation à mémoire virtuellement partagée, la gestion des données est à la charge du système. Le programmeur ne doit pas s'occuper explicitement des synchronisations entre les accès concurrents aux données. Dans un programme parallèle, une données peut être accédée par une ou plusieurs tâches simultanément. Ceci peut conduire à un problème d'incohérence entre les différentes copies d'une même donnée. Un modèle de cohérence doit donc garantir une gestion efficace de la cohérence des données afin de réduire la dégradation des performances que peut engendrer le partage des données dans ce genre de systèmes.

Ce chapitre est consacré à la problématique de la cohérence des données. La section 1.3.1 définit la problématique de cohérence des données, la section 1.3.2 présente les différents modèles de cohérence existants, ensuite nous traitons les protocoles et mécanismes de gestion de la cohérence issus des différents modèles dans la section 1.3.3. Finalement, la section 1.3.4 est une discussion des limitations que peut présenter le problème de la cohérence pour les architectures manycœurs.

1.3.1 Définition du problème de la cohérence des données

Les systèmes à mémoire répartie sont caractérisés par plusieurs niveaux de stockage (Mémoire caches, Mémoire principale). La gestion efficace de ces différents niveaux de stockage permet d'améliorer les performances du système.

Les mémoires caches exploitent la notion de *localité* en maintenant automatiquement des copies des données récemment utilisées. Les prochains accès à ces données sont effectués directement dans les mémoires caches et non pas dans la mémoire principale. De multiples copies d'une même donnée peuvent êtres présentes sur les différents caches. Les cœurs en possession d'un copie d'une donnée peuvent à tout moment modifier sa valeur localement ce qui génère le problème d'incohérence. La figure 1.5 illustre un exemple de scénario d'incohérence où deux différentes valeurs sont associées à la même donnée a dans les caches des cœurs 1 et cœur 2 après modification de sa valeur par le premier cœur.

Dans ce genre de système, les modèle de gestion de cache est responsable de transmettre les mises à jours des données vers la mémoire principale. Un système de cache est donc dit *cohérent* si pour chaque accès en lecture à une donnée la valeur retournée correspond à la dernière modification apportée à cette donnée. La figure 1.6 illustre un scénario de gestion de la cohérence entre les deux cœurs 1 et 2 correspondant au cas

étudié précédemment.

Il existe plusieurs approches pour résoudre le problème de la cohérence des données. Ces approches dépendent de plusieurs critères.

La suite de ce chapitre présente l'état de l'art des modèles de gestion de la cohérence pour les systèmes répartis à différentes échelles.

1.3.2 Modèles de cohérence des données

Le problème de la cohérence des données a été traité à plusieurs reprises, pour différentes architectures et différents types de systèmes. Pour des programmes séquentiels, s'exécutant sur des machines de type Von Neumann, le modèle d'exécution est naturellement clair. Les instructions sont exécutées dans le même ordre défini par le programmeur ou le compilateur.

Cependant, avec l'utilisation des systèmes parallèles à mémoire répartie, notamment les manycœurs, les opérations sur les données se font potentiellement dans différents ordres.

Les incohérences dûes au accès concurrentiels aux données peuvent être problématiques dans certains cas. Par contre, il peut arriver qu'il n'y ait pas besoin d'une cohérence stricte. Par exemple, dans le cas d'une application qui surveillerait la charge des processeurs d'un réseau d'ordinateurs afin de choisir sur quelle machine exécuter une tâche, les données qui représentent la charge des processeurs évoluent sans arrêt : il n'est pas nécessaire d'avoir à tout moment accès à la dernière valeur de la charge .

Répondre à la question de la cohérence des données revient à définir un ensemble de techniques pour le stockage des données sur les mémoires réparties et pour la gestion des accès parallèles à ces données.

La cohérence des données telle que présente dans la littérature peut être classée en plusieurs niveaux hiérarchiques (figure 1.7) :

Cohérence stricte : c'est la forme la plus intuitive de la cohérence. Elle correspond à garantir au programmeur que chaque lecture d'une donnée correspond bien à la version la plus fraîchement modifiée dans tous les caches du système.

Cohérence séquentielle : Elle consiste à préserver l'ordre global des accès mémoire en lecture. Cet ordre correspond à l'ordre du programme. Le résultat final d'une exécution est similaire au résultat d'une exécution séquentielle du programme.

Cohérence faible : le programmeur est impliqué dans la gestion de la cohérence en utilisant des opérateurs de synchronisation séquentiellement cohérent.

Cohérence par nœud : L'ordre des écritures effectuées par un même nœud est préservé tandis que les écritures des différents nœuds peuvent être vues autrement.

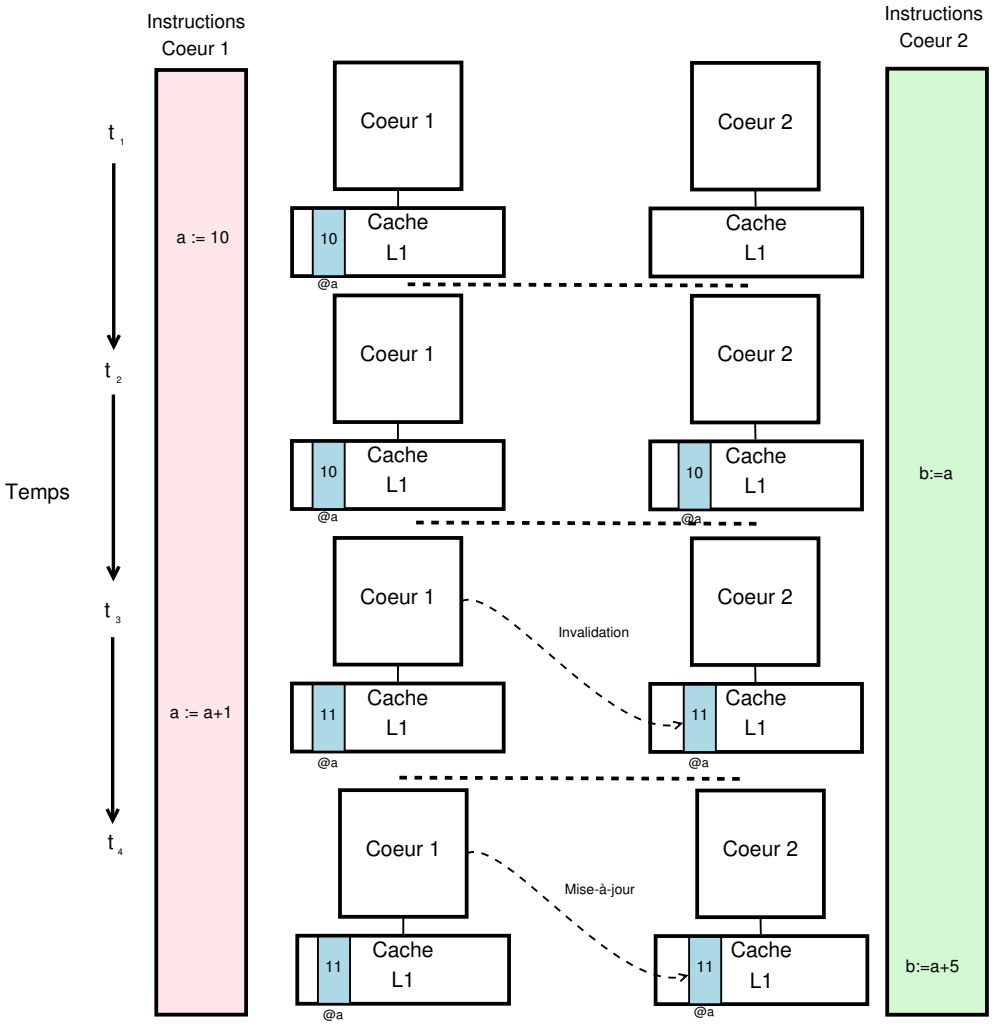


FIGURE 1.5 – Illustration d'un scénario d'accès multiples à une même donnée par les deux cœurs 1 et 2 dans un système à caches répartis. L'incohérence de la donnée A est dûe à son accessibilité possible par les deux différents cœurs.

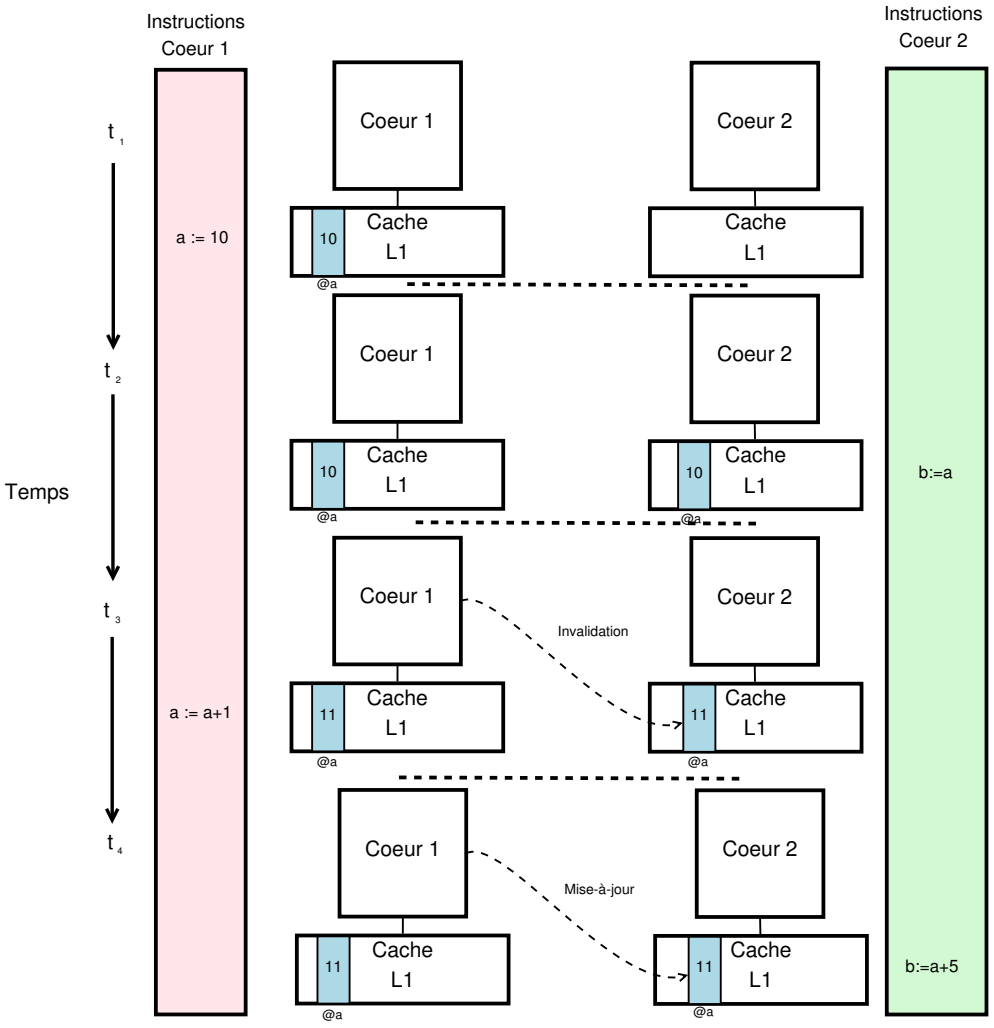


FIGURE 1.6 – Illustration d'un scénario simplifié de gestion de la cohérence de données dans un système à cache cohérent. Les traits en pointillé représentent les opérations générées par le mécanisme de la cohérence des données. Avant que le cœur 1 modifie la copie de a dans son cache local, un message d'invalidation est envoyé au cœur 2 (et à tout autre cœur possédant une copie de cette donnée). La valeur mise à jour de a est ensuite transmise aux cœurs concernés.

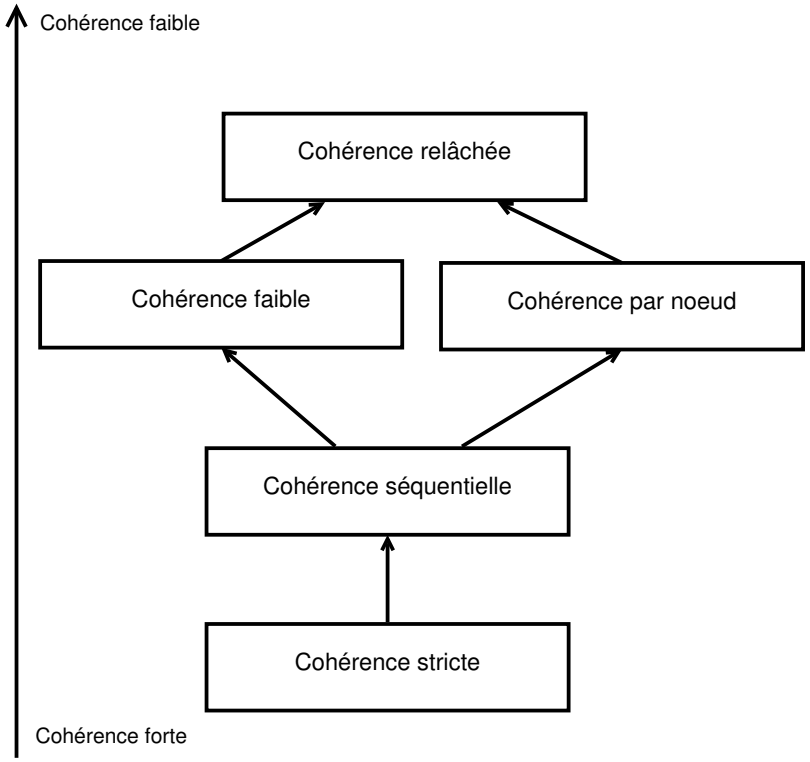


FIGURE 1.7 – Représentation hiérarchique des différents modèles de cohérence de données du modèle le plus strict (plus bas) au modèle le plus relâché (plus haut).

Cohérence relâchée La cohérence relâchée considère deux types d'opérateurs de synchronisation : *acquire, release*. Chaque opérateur est considéré cohérent par nœud.

L'efficacité d'un modèle de cohérence est primordiale dans l'écriture d'un programme dans un modèle à mémoire virtuellement partagée.

La synchronisation basée sur des verrous est sujette à un certain nombre de dysfonctionnement dûs au phénomènes suivants :

- Interblocage et inversion de priorité des tâches (une tâche secondaire bloque une autre plus prioritaire).
- Vulnérabilité à l'arrêt brusque d'un tâche avant de libérer le verrou.
- Attente active en cas de forte concurrence d'accès à une ressources partagée (verrou). Chacun des tâches concurrents essaie de récupérer le verrou ce qui cause un blocage infini.
- Blocage des threads tant qu'ils ont la main sur un verrou, ce qui dégrade les performances en cas de défaut d'accès à la mémoire.

Afin d'éviter les problèmes précédents liés à l'utilisation des verrous, le concept de la *mémoire transactionnelle* a été proposé par (Herlihy et Moss 1993) [HM93]. La *mémoire transactionnelle* est une abstraction permettant de libérer le programmeur de la gestion de la concurrence. Il est chargé de définir uniquement les séquences d'instructions ayant besoin d'accès atomiques aux données partagées. Cette technique repose sur des notions d'atomicité, de cohérence et d'isolation inspirées du domaine des bases de données. Elle impose à l'exécution des critères de concurrence qui permettent de décider d'effectuer une transaction ou bien l'arrêter et la recommencer plus tard. La possibilité d'interrompre une tâche élimine les interblocages potentiels tout en garantissant l'exécution concurrente des tâches non conflictuelles.

La *mémoire transactionnelle* peut être implémentée entièrement en logiciel ou avec un support matériel. Une implémentation matérielle a été proposée dans les travaux de Herlihy et Moss [HM93]. Elle est basée sur extension des protocoles de cohérence et fournit le support pour un langage transactionnel permettant d'exprimer les opérations de synchronisation (sous forme de transactions). Pour sérialiser les accès concurrents aux données, cette approche matérielle utilise la technique d'exclusion mutuelle entre les sections critiques (i.e. sections atomiques). Des approches logicielles ont été proposées afin d'améliorer l'implémentation des *mémoires transactionnelles* [HLMSI03]. Elles sont basées sur des algorithmes de synchronisation non bloquante, tel que l'algorithme *STM (pour Software Transactional Memory)* [ST97], ce qui permet des accès concurrents et asynchrones aux données partagées tout en garantissant leur cohérence.

La *mémoire transactionnelle* nécessite une connaissance préalable de l'ensemble des ressources partagées de toutes les transactions et aucun nouvel accès non défini ni pris en compte. Herelihy et al. ont proposé plus tard de nouveaux algorithmes [HLM06, HF03] pour la gestion dynamique des ressources partagées afin de répondre aux besoins des applications utilisant des allocations dynamiques des données.

Dans un modèle de programmation à mémoire virtuellement partagée, les performances de calcul dépendent fortement de la capacité du système à gérer le partage des données. Il est donc très important d'avoir, lors de la conception d'un système à mémoire virtuellement partagée, un choix efficace du modèle de cohérence et des protocoles qui en relève. La section suivante présente les différentes implémentations d'un modèle de cohérence.

1.3.3 Implémentations : Protocoles et mécanismes de gestion de la cohérence

Tous les systèmes répartis à mémoire virtuellement partagée assure une forme de cohérence des données. Une manière intuitive de gérer cette cohérence est de sérialiser toutes les requêtes d'accès aux données partagée ce qui correspond à de la cohérence forte. Cette forme de cohérence va à l'encontre du principal avantage des systèmes à mémoire réparti reposant sur le parallélisme des tâches.

D'autre part, l'autorisation de duplication des données dans les différentes mémoires complique la gestion de la cohérence.

Il existe deux grandes familles de protocoles de cohérence pour les systèmes à mémoire répartie :

Protocoles *Write-Through* [Jha93] : cette technique consiste à propager la valeur modifiée après toute opération d'écriture. Tous les nœuds ont donc en permanence des données à jour. L'inconvénient de cette méthode en revanche est que les propagations des mises à jour doivent être totalement ordonnées pour respecter l'ordre des émissions de tels messages (figure 1.8).

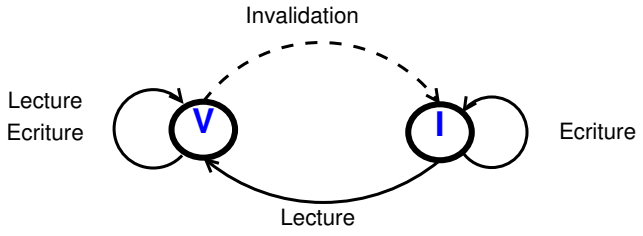


FIGURE 1.8 – Les deux états du protocole *Write-Through*. Tout accès en écriture à une donnée nécessite l'invalidation des autres copies disponibles.

Protocole *Write-Back* [LLG+90a] : Cette approche permet plusieurs accès en lecture seule simultanés sur une donnée, mais un seul en écriture est possible. Lorsqu'un nœud veut modifier une donnée, il doit envoyer un message aux autres nœuds pour invalider cette donnée et attendre leurs retours. Les mises à jour sont propagées uniquement quand un processus accède à une donnée invalide, ce qui évite les transmissions superflues. De plus le coût est réduit car les propagations n'ont pas besoin d'être totalement

ordonnées. Le protocole à état appelé *MESI (Modified, Exclusive, Shared, Invalid)*, ainsi que ses différentes variantes [SBL04], est le plus commun de cette famille (figure 1.9).

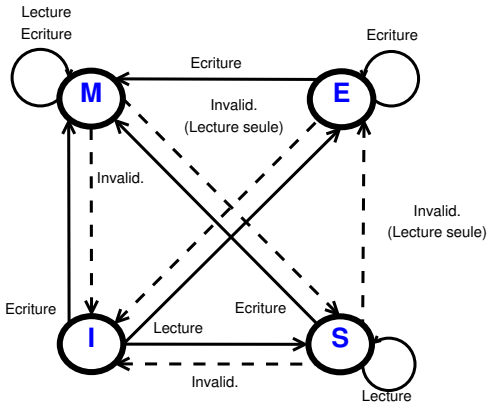


FIGURE 1.9 – Les 4 états du protocole *MESI*. L'état *Modified* indique que la donnée a été localement modifiée. L'état *Exclusive* correspond à un accès exclusif de la donnée ce qui signifie que toutes les autres copies sont invalides. L'état *Shared* signifie que la donnée est partagée entre plusieurs nœuds et toutes les copies sont cohérentes. L'état *Invalid* se produit lorsque un nœud demande un accès exclusif à la donnée pour une modifier sa valeur ainsi toutes les autres copies sont invalidées.

1.3.4 Discussion

La gestion de la cohérence dans les systèmes multicœurs et manycœurs nécessite une prise en compte de nouvelles contraintes liées à l'organisation de la mémoire dans ce type d'architecture.

Avec l'augmentation du coût d'accès à la mémoire principale, la hiérarchie mémoire associée à chaque cœur peut être composée de un ou plusieurs niveaux de caches. Une façon d'obtenir de meilleures performances est de maximiser l'utilisation de la mémoire sur puce afin de réduire les accès coûteux à la mémoire extérieure. Une gestion efficace de la capacité mémoire sur la puce nécessite la limitation de la réplication des données entre les caches.

Les caches peuvent être organisés de différentes manières en fonction du nombre de niveaux de cache et de la structure des différents niveaux (cache privé/ partagé). L'organisation de la mémoire sur puce a un impact important sur la conception des mécanismes de cohérence entre les caches. Dans une organisation telle que décrite dans la figure 1.10 la cohérence doit être garantie entre les caches $L1$ (privés) et aussi entre les caches $L2$ (partagés).

Une autre manière de maximiser la capacité de cache sur la puce est de récupérer les données des caches voisins en impactant le protocole de cohérence.

Afin de répondre à la question de passage à l'échelle de la mémoire sur puce, les systèmes sur puce tels que les manycœurs utilisent de nombreuses techniques de pla-

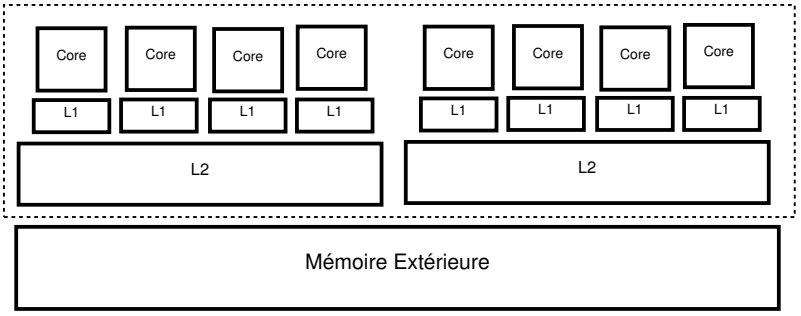


FIGURE 1.10

cement, de migration et de réplication des données. Il devient plus courant d'intégrer l'ensemble des ces mécanismes dans le protocoles de gestion de la cohérence.

Dans la suite de ce travail nous nous intéressons à la gestion de la cohérence pour les manycœurs. Un état de l'art plus développé est présenté dans les chapitres suivants.

Chapitre 2

Étude des mécanismes de gestion des données pour les systèmes répartis

La gestion de la cohérence des données dans les architectures massivement parallèles est un problème clé pour l'optimisation des performances de ces systèmes. Afin de réduire le temps d'accès aux données et la contention dans la mémoire principale, les processeurs à hautes performances intègrent plusieurs niveaux de caches. Chaque cœur a un premier niveau de cache privé $L1$ composé d'un cache d'instructions et d'un cache de données. En plus d'un deuxième niveau de cache $L2$, certaines architectures incluent un troisième niveau $L3$. Dans la suite de ces travaux le dernier niveau de cache est noté LLC (*Low Level Cache*). Ces différents niveaux de cache sont organisés de plusieurs manières selon des choix de conception qui peuvent varier d'une architecture à une autre.

Nous étudions dans ce chapitre d'abords les techniques de gestion des données existantes dans différents types d'architectures. Ensuite, nous présentons deux contributions proposant de nouveaux protocoles coopératifs de gestion des données pour des architectures manycœurs.

La suite de ce chapitre est organisée comme suit : la section 2.1 décrit l'approche à cache privé et et celle à cache partagé, la section 2.2 présente l'approche à cache coopératif pour différents types de systèmes, la section 2.3 est focalisée sur les techniques coopératives pour les systèmes manycœurs, la section 2.4 présente une première contribution proposant le protocole de glissement de données, un protocole coopératif autorisant la migration des données vers les voisinages coopératifs. et finalement la section 2.5 décrit la deuxième contribution qui est une extension du protocole de glissement de données adaptant dynamiquement le rayon de migration des données à la répartition de la charge sur la puce.

2.1 Les approches à cache privé et à cache partagé

Les performances des systèmes répartis sont fortement liées à leur capacité à gérer le stockage des données dans leurs mémoires. De ce fait, l'organisation des différents niveaux de cache est d'une grande importance dans la conception de ces systèmes. Le coût et la capacité des mémoires caches sont les principales contraintes des choix effectués par le concepteur.

Un exemple d'organisation courante dans les systèmes sur puce est l'approche à cache partagé. L'organisation à cache partagé consiste à créer un niveau de cache LLC plus large et partagé entre un ensemble de cœurs (figure 2.1a). Une alternative à cette approche est l'organisation en caches privés où chaque cœur est doté d'un niveau de cache LLC privé avec un accès local direct et rapide (figure 2.1b).

2.1.1 Cache partagé

La structure à cache partagé est constituée, en plus du niveau de cache $L1$ privé, d'un cache LLC partagé par plusieurs cœurs. Le cache LLC partagé est caractérisé par une grande capacité de stockage ce qui lui permet de répondre aux besoins des cœurs lui étant associés. Dans cette organisation, le cache $L1$ privé s'occupe de filtrer les requêtes d'accès au niveau de chaque cœur afin de réduire la charge en accès à la zone partagée. En effet, lorsqu'un défaut de cache se produit sur le cache $L1$, une requête est envoyée par le cœur vers le cache partagé LLC. Le contrôleur du cache LLC récupère la requête et recherche la donnée dans son cache.

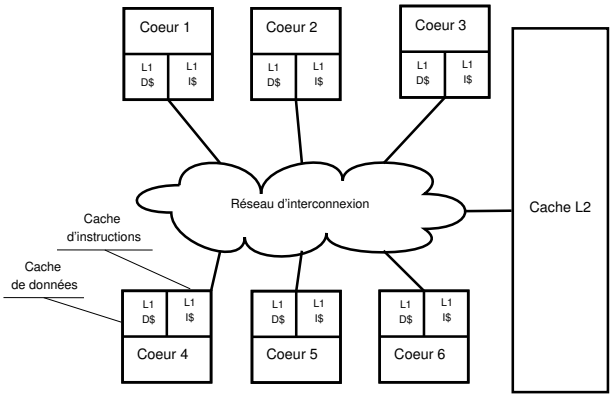
Il n'y a pas de duplication de données dans le cache LLC partagé. Cependant, une donnée peut avoir plusieurs copies dans les caches $L1$ privés de plusieurs cœurs. Ceci nécessite la gestion de la cohérence des données entre les niveaux de cache $L1$ et le cache LLC partagé.

Parmi les techniques de gestion des mises à jour des données les plus utilisées pour des systèmes à petite échelle (i.e. moins de 10 cœurs), nous citons :

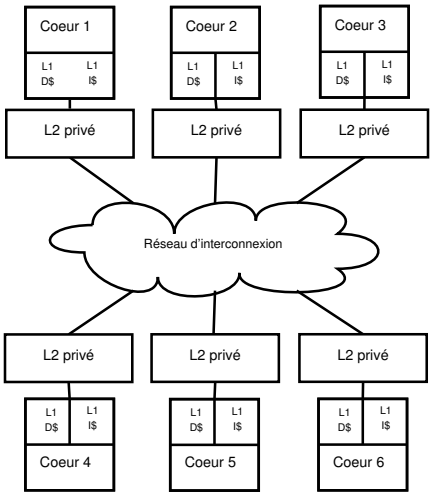
- Écriture immédiate (*write-through*) : La modification apportée à la copie d'une donnée dans le cache $L1$ est immédiatement rapportée à la copie principale de la donnée dans le cache LLC.
- Écriture différée (*write-back*) : La mise à jour de la donnée dans le cache LLC n'est effective que lorsque la copie de la donnée est invalidée ou remplacée.
- Écrire et invalider : Avant toute écriture toutes les autres copies d'une données sont invalidées.
- Écrire et mettre à jour : toutes les copies d'une donnée sont mises à jour à chaque écriture.

L'utilisation de ces techniques pour des systèmes à plus grande échelle (i.e. plusieurs dizaines de cœurs) est limitée par le trafic et la dissipation thermique qui en résultent.

Avec l'introduction des techniques de communication directe entre le cache LLC et les différents cœurs, de nouveaux mécanismes de cohérence sont proposés. Une des



(a) Organisation à cache partagé



(b) Organisation à cache privé

FIGURE 2.1 – Deux différentes organisations physiques des caches sur une puce. La figure 2.1a représente 6 cœurs avec chacun un niveau de cache $L1$ et d'un niveau de cache $L2$ partagé entre les différents cœurs. La figure 2.1b représente la même architecture avec une organisation à caches $L2$ privés répartis entre les 6 cœurs. Dans les deux configurations le niveau de cache $L1$ est toujours privé et il est composé d'un cache d'instructions et d'un cache de données.

techniques les plus utilisées est la gestion de la cohérence par répertoires (*Directory-based*). Dans cette approche chaque donnée est associée à un répertoire qui s'occupe de garder l'information sur le degré de fraîcheur de la donnée en notant si un autre cœur en a une copie valide. Des requêtes de lecture ou d'invalidation peuvent être envoyées afin d'assurer la cohérence entre le cache LLC et ses caches $L1$ associés.

Le modèle à cache LLC partagé présente de nombreux avantages. Tout d'abord, l'espace de cache disponible peut être alloué dynamiquement entre plusieurs cœurs pour une utilisation plus efficace de la zone du cache partagée. De plus, une seule copie de la donnée est maintenue dans le cache $L2$ même si elle est sollicitée par plusieurs cœurs ce qui permet ainsi d'optimiser l'utilisation de l'espace cache et d'augmenter ses performances (plus de succès de cache) [BJM11, LSL+09].

Toutefois, l'interférence entre les différentes charges de travail des cœurs peut dégrader les performances en engendrant plus de défauts de cache. La gestion de la cohérence ajoute également un surcoût pouvant pénaliser les performances du système. De plus, de nombreux travaux ont démontré que le temps d'accès aux données dans le cache LLC partagé est très long en moyenne [DS07, ZIUN08].

2.1.2 Cache privé

Dans certaines architectures le cache partagé est remplacé par un ensemble de caches *LLCs* privés. Dans cette configuration chaque cœur a au moins deux niveaux de cache privé : un niveau $L1$ (données/instructions) et un niveau LLC. Si la donnée n'est pas disponible dans le cache du niveau $L1$, une requête est envoyée vers le cache LLC. Les caches *LLCs* sont indépendants et ne sont accessibles que par leur cœur local, ce qui permet un accès plus rapide aux données. Il n'y a pas pas de gestion de cohérence complexe entre les différents niveaux de caches ni de concurrence d'accès, ce qui augmente les performances d'accès à la mémoire.

L'un des inconvénients liés à ce modèle est que les données accédées par plusieurs cœurs sont dupliquées sur les différents *LLCs* privés de ces cœurs en dégradant ainsi le taux d'utilisation de la mémoire globale sur la puce. Un deuxième problème lié à l'efficacité de gestion de l'espace cache global est dû à l'allocation statique du cache LLC. En effet, en cas de différentes charges de travail un cœur peut ne pas utiliser la totalité de son cache LLC, tandis qu'un autre cœur est en besoin de plus d'espace de stockage que ce qui est disponible sur son cache LLC privé. L'utilisation d'une approche à cache LLC partagé permet la gestion dynamique de la mémoire partagée entre les différents cœurs ce qui offre une meilleure utilisation de l'espace mémoire.

2.1.3 Comparaison entre cache privé et partagé

Le tableau 2.1 résume les différences entre les deux configurations des niveaux de cache. Les deux modèles d'organisation de cache LLC privé et partagé représentent chacun des aspects positifs et négatifs. La section suivante 2.2 présente une technique qui permet de combiner le modèle à cache privé et celui à cache partagé afin d'augmenter l'efficacité de stockage sur la puce en s'appuyant sur les points forts des deux approches

Cache partagé	Cache privé
Pas de copies multiples des données (Capacité effective plus grande)	Données partagées répliquées (Capacité effective plus petite)
Allocation dynamique des données entre fils d'exécution et cœurs	Caches alloués statiquement aux différents cœurs (Capacité effective plus petite)
Surcoût pour la gestion de la cohérence des données partagées	Gestion de cohérence entre les caches $L1$ et les caches LLC plus simple
Des interférences entre les fils d'exécution dégradeent les performances	Pas d'interférences entre les fils d'exécution (Meilleures performances)
Plus grande latence d'accès au LLC partagé	Plus petite latence d'accès au niveau LLC privé
Grande contention pour accéder aux données stockées dans le cache LLC partagé	Pas de contention pour accéder au cache LLC privé)

TABLE 2.1 – Table de comparaison entre les structures à cache LLC partagé et privé

précédentes.

2.2 Approches à caches coopératifs pour les systèmes répartis

La section précédente (section 2.1) a permis de discuter des avantages et des inconvénients des caches partagés et privés.

Une solution permettant le passage à l'échelle tout en garantissant de bonnes performances de mémoire est l'approche hybride, regroupant les deux modèles précédents. Nous étudions dans cette section l'approche hybride à caches coopératifs. Elle permet de combiner l'avantage de réduire la latence d'accès aux données grâce à une organisation privée des mémoires caches et l'avantage de diminuer le nombre de défauts de cache en autorisant l'échange des données partagées entre les caches privés distants comme dans une approche à cache partagé.

Nous présentons les différents travaux basés sur le concept des caches coopératifs dans les différents systèmes répartis tels que les réseaux connectés (ex : LAN, web) (section 2.2.1), les réseaux sans fils (ex : réseaux mobiles, réseaux de capteurs)(2.2.2), et les multiprocesseurs (§ 2.2.3).

2.2.1 L'approche coopérative pour les systèmes de fichiers

L'exploration de la piste de la mémoire coopérative pour des systèmes de fichiers est motivée par deux tendances technologiques : La première est liée la divergence entre l'évolution de la puissance de calcul des processeurs et la vitesse d'accès en mémoire disque. Il est actuellement 10 à 20 fois plus rapide d'effectuer un accès à une mémoire distante que d'accéder au disque d'un serveur distant [DWAP94, VGRN08]. L'augmentation de cette divergence rend plus important de réduire les accès aux disques. La deuxième tendance technologique est liée à l'amélioration des performances des réseaux de connexion (ex. Ethernet Gigabit, InfiniBand, Myrinet), ce qui permet désormais de déplacer les données entre les différents clients à une latence beaucoup plus faible comparée au réseau *Ethernet standard*.

La hiérarchie mémoire des systèmes de fichiers est composée de trois niveaux : (1) la mémoire du client, (2) le cache associé au serveur,(3) le disque du serveur.

Une solution intuitive d'amélioration des performances mémoire du système est d'augmenter la fraction de la mémoire cache du système allouée au serveur. Cette solution est coûteuse et ne passe pas à l'échelle avec l'augmentation du nombre des clients connectés au serveur.

L'approche coopérative peut être vue comme l'introduction d'une nouvelle couche dans la hiérarchie mémoire des systèmes de fichiers. Elle s'appuie sur la coordination entre les différentes mémoires clients en permettant les accès entre les caches des clients distants. Une donnée peut donc se trouver dans : le cache local, le cache serveur, le disque serveur ou bien le cache d'un client distant.

Une telle approche représente l'avantage d'améliorer la disponibilité des données dans le système (grâce à des caches clients plus grands) et leur accessibilité (grâce à la qualité du réseau de communication entre les nœuds).

Il existe dans la littérature de nombreux algorithmes de coopération entres les mémoires réparties des clients dans les systèmes de fichiers. Ces algorithmes varient en fonction de leurs implémentations du concept de coopération dans les différents contextes de fonctionnement.

Nous décrivons dans ce qui suit, de manière non exhaustive, les différents algorithmes étudiés dans les travaux de Dahlin et al. [DWAP94] et dans des travaux postérieurs [CGL96].

Coopération directe : Un client à cache saturé peut stocker ses données supplémentaires dans le cache d'une autre machine non active. Le client actif peut accéder à ses données jusqu'au réveil de la machine distantes, qui va ensuite les éjecter. Cet algorithme propose des mécanismes efficaces permettant de repérer les clients non actifs. Il ne nécessite aucune modification au niveau du serveur, ce qui fait sa simplicité et son faible coût d'implémentation. Le principal point faible de cette approche est qu'il n'y a pas de coopération entre les clients actifs ce qui limite les performances du système.

Coopération par transfert glouton : L'ensemble des caches clients constitue un espace de stockage uni où chaque client gère son cache local de façon gloutonne sans

prendre en compte le contenu des autres caches. Dans le cas d'un défaut de cache local, une requête d'accès est envoyée au serveur. Si ce dernier ne peut pas satisfaire cette requête à partir de son cache, il la transfère au client ayant la donnée. Le client en question se charge par la suite d'envoyer la donnée vers le demandeur. Dans le cas où aucun client ne possède une copie de la donnée, la requête est satisfaite de manière classique à partir du disque serveur. Cette technique permet à chaque client de gérer son cache selon ses propres besoins tout en profitant des autres caches clients. Le manque de coordination directe entre les différents caches clients peut causer une pollution de la mémoire système à cause de la duplication des données.

Coopération centralisée : Le cache client est statiquement partitionné en deux zones, une zone privée gérée localement par le client et une zone globale contrôlée par le serveur. Ce dernier utilise un algorithme de remplacement global pour une gestion centralisée des fractions de mémoire mises à sa disposition. L'accueil d'une nouvelle donnée en provenance du disque, nécessite le transfert d'une donnée cible du cache serveur vers un cache client. Le choix de ce dernier est basé sur une recherche globale de la donnée la moins récemment utilisée *LRU (Least Recently used)*. L'avantage d'une gestion centralisée de la mémoire coopérative est l'augmentation du nombre de succès de cache au niveau de la mémoire globale. Une dégradation des performances du système peut être causée par la limitation de la taille des fractions locales des caches clients et par l'augmentation de la charge du serveur dûe à la centralisation de la coordination entre les clients.

Coopération par transfert multiple (*N-Chance Forwarding*) : Il s'agit d'une gestion dynamique de la répartition de la mémoire entre le cache client local et le cache coopératif en fonction de la charge mémoire des différents clients. Cet algorithme modifie *l'algorithme par transfert glouton* avec la prise en compte des contenus des caches clients afin d'éviter la duplication des données. Le principe de l'algorithme de *transfert multiple* est de déplacer les copies uniques de données *N-fois* entre les noeuds clients. Ceci permet d'avoir un meilleur compromis d'allocation mémoire entre les données locales et les données conservées dans le cache global. Afin de réduire la charge des nœuds actifs et d'y libérer plus d'espace de stockage, les données transférées vont plus vers les nœuds inactifs. D'autres techniques sont utilisées pour réduire le trafic réseau et les communications avec le serveur, notamment l'utilisation de la valeur du paramètre de déplacement N pour déduire s'il s'agit d'une copie unique de la donnée. La technique de coopération par *transfert multiple* permet d'avoir de meilleures performances grâce à la gestion dynamique du stockage des données entre le cache privé et le cache coopératif global. Il permet également d'augmenter l'efficacité de stockage à travers la prise en compte de l'état de chaque donnée en favorisant le stockage des copies uniques dans les caches distants. La charge du réseau dûe aux multiples déplacements des données peut malheureusement limiter les performances de cette technique.

Coopération distribuée par hachage : La mémoire globale centralisée est divisée

en plusieurs parties. Chacune de ces parties est gérée par un client différent. Ce dernier est défini selon une fonction de hachage qui permet au serveur de choisir la destination des blocs de données qu'il souhaite déplacer de son cache. Un client peut envoyer directement les requêtes d'accès aux données sans passer par le serveur. Cet algorithme diffère de l'algorithme de *coopération centralisée* par sa capacité à réduire la charge réseau.

Remplacement par pondération : Comme pour l'algorithme par *transfert multiple*, la politique de remplacement par pondération vise à conserver les données les plus fréquemment utilisées tout en favorisant les moins dupliquées sur la puce. Ainsi, il associe un poids à chaque donnée en prenant en compte le nombre de copies de cette donnée, sa fréquence d'accès aussi bien que l'espace qu'elle occupe dans le cache. Il s'agit d'une version pondérée de l'algorithme de remplacement des données les moins récemment utilisées *LRU*. Le principal avantage de cet algorithme est sa capacité à équilibrer entre la conservation des données dupliquées les plus utilisées et les copies uniques les moins fréquemment utilisées dans l'optique d'éviter les accès au disque serveur.

Plusieurs travaux ont étudié l'efficacité des techniques de coopération dans différents scénarios applicatifs et différentes configurations du système. Les travaux de Leff et al. [LYW91, LWY93, LYW93] ont exploré plusieurs formes de caches coopératifs pour les systèmes de fichiers de type client/serveur.

Leff et al. ont étudié deux techniques d'implémentation de la coopération entre les caches : une première technique basée sur une vision globale permettant à chaque client de déterminer directement l'emplacement d'une donnée et une deuxième technique basée sur une vision locale où le client doit interroger des clients intermédiaires. Cette étude a démontré que lorsque la décision d'accès au cache d'un client distant est basée sur une vision globale du système de meilleures performances sont obtenues pour des systèmes de petite taille (< 10 nœuds). Cependant, le coût d'implémentation d'une approche globale reste élevé ce qui limite le passage à l'échelle.

Le principe de coopération entre des caches distants a également été étudié dans les travaux de Francklin et al. [FCL92] pour des systèmes de base de données. Ils ont étudié le mécanisme de gestion centralisée de la mémoire globale du système. Ces travaux ont mené à une amélioration des performances à travers la réduction du taux de duplication des objets dans la mémoire globale et la diminution de la charge du serveur.

Les travaux de thèse de Blaze [Bla93] étudient une nouvelle hiérarchie de caches dynamique basée sur le transfert des données fréquemment utilisées entre les caches des clients. Les résultats de ces travaux ont démontré une bonne réduction de la charge du serveur par rapport au modèle de hiérarchie de cache statique proposé par Muntz and Honeyman [MH91]. Le modèle de coopération défini dans les travaux de Muntz et Honeyman consiste à désigner des clients comme serveurs intermédiaires. Cette technique revient à définir un niveau de stockage supplémentaire afin de réduire les accès aux serveurs principaux.

L'étude des algorithmes et techniques de coopération pour des systèmes de fichiers a démontré des performances variables en fonction de la charge de travail et de la configu-

ration du système. La majorité des études effectuées a conclu sur l'efficacité d'utilisation d'une telle approche permettant d'avoir un bon compromis entre la répartition de la mémoire globale entre les clients distants et le coût de gestion lui étant associé.

Nous traitons dans la section suivante l'approche de coopération dans l'environnement des réseaux sans fils avec une nouvelle contrainte à gérer qui est la contrainte de volatilité.

2.2.2 L'approche coopérative pour les réseaux sans fils

L'évolution des communications pair-à-pair a permis d'introduire l'approche coopérative pour des systèmes sans fil tels que les réseaux mobiles ad-hoc et les réseaux de capteurs. L'approche coopérative initialement conçue pour des systèmes statiques et connectés physiquement, a dû être adaptée pour les réseaux sans fil en développant de nouvelles techniques de partage et de coordination qui prennent en compte la mobilité des nœuds ainsi que la non fiabilité des connexions sans fils.

L'émergence des technologies de communication Pair-à-Pair a permis de développer des réseaux mobiles ad-hoc (*en anglais : Mobile Adhoc Network (MANET)).* Ils sont constitués de serveurs stationnaires qui fournissent les informations aux clients mobiles existants dans leur espace de services. Dans les réseaux MANET les clients mobiles sont capables de partager les informations entre eux sans passer par les serveurs stationnaires. Dans les systèmes pair-à-pair ce paradigme de partage constitue un modèle de coopération entre les caches des différents clients mobiles. Nous trouvons dans la littérature différentes techniques de coopération adaptée aux contraintes de ce type de systèmes telles que l'asymétrie des communications et l'instabilité de la topologie du réseau dues à la mobilité des clients.

Cho et al. ont proposé une technique de coopération de voisinage direct [COK+03] afin d'améliorer l'efficacité des caches des nœuds mobiles. Leur technique est basée sur la création de zones de coopération autour de chaque nœud. Les nœuds séparés d'une distance d'un pas constituent un voisinage coopératif. Pendant leur temps d'inactivité, les nœuds mettent leur espace cache libre à la disposition de leurs voisins. Le voisin ayant le plus grand temps d'inactivité est le plus favorable pour recevoir la donnée éjectée. Le choix du meilleur voisin est effectué à partir des informations échangées à chaque communication entre voisins. Le cœur demandeur, n'ayant pas toujours l'information sur l'état courant de son voisinage, doit négocier l'accueil de sa donnée avec le voisin sélectionné. Ce dernier peut rejeter la requête de stockage en cas de saturation de son cache. Si tous les voisins rejettent la requête de stockage, la donnée est perdue.

Afin de réduire la consommation de la bande passante et la latence d'accès aux données, les travaux de Yin et al. ont proposé 3 modèles de coopération [YC06] :

- CacheData : les nœuds intermédiaires stockent localement des copies de données afin de répondre aux futures requêtes d'accès à ces données.
- CachePath : les nœuds mobiles gardent en cache les chemins d'accès aux données afin de rediriger les requêtes d'accès aux données vers les nœuds les plus proches possédant une copie de la donnée demandée.

- HybridCache : un modèle hybride prenant avantage des deux précédentes techniques. Avec ce modèle, un nœud peut choisir entre garder une copie de la donnée ou de son chemin d'accès. Le choix est fait selon certains critères définis par le modèle tels que la taille de la donnée ou son temps de vie (TTL).

Ces travaux ont permis de réduire drastiquement le temps de réponse aux requêtes d'accès et le nombre de messages générés par les différents nœuds mobiles pour assurer le transfert d'une donnée.

Le partage d'information dans les réseaux mobiles est limité par la bande passante, la mobilité des utilisateurs mais aussi par la batterie des nœuds connectés. Différents modèles de coopération basés sur la formation de groupes coopératifs au sein du réseau ont été proposé pour défier ces limitations. Chacun des modèles propose une technique de création des groupes en prenant en compte certaines propriétés physiques du réseau.

Le modèle *PReCinCT* [SJKD05] proposé par Shen et al. consiste à diviser le réseau en plusieurs régions de proximité géographique. Chaque région est responsable d'un ensemble de données. L'association entre groupes de données et régions est calculée par une fonction de hachage. Les requêtes d'accès aux données sont directement envoyées vers les régions appropriées. Un nœud n'appartenant pas à cette région se limite à transférer la requête au nœud suivant. Aussi, afin de réduire le coût de mobilité des nœuds, les clés de données sont cédées au reste des nœuds résidents dans la région à chaque fois que l'un d'eux la quitte. Les performances de ce modèle, notamment en terme de consommation énergétique dépendent de certains paramètres tels que la taille et le nombre des régions définies dans le réseau.

Un autre modèle basé sur la coopération par groupe de nœuds a été proposé par Chow et al. Ces travaux [CLC07] ont considéré à la fois le voisinage géographique des données et leur voisinage fonctionnel. Dans le modèle *COCA (Cooperative Caching)*, le voisinage est maintenu par une diffusion périodique de messages de connexion. Un nœud diffuse une requête d'accès dans son voisinage. Le premier nœud qui répond favorablement à sa requête est défini comme nœud cible. Au prochain accès, il sera d'abord consulté par un message direct (Point-à-Point) avant la diffusion de la requête à tout le voisinage dans le cas d'une réponse défavorable. Cette technique permet de développer une connaissance de son voisinage afin d'éviter les diffusions de messages groupés à chaque accès.

Le modèle *GroCOCA* est une extension du modèle précédent qui prend en compte les schémas de mobilité et les affinités d'accès aux données. La mobilité est modélisée par la distance moyenne entre les nœuds. Le nœud serveur définit un vecteur compteur pour chaque nœud afin de maintenir une visibilité sur les accès aux données par les différents nœuds. Ces informations permettent à chaque nœud de gérer son cache local en respectant non seulement ses besoins en mémoire mais aussi celui des membres de son groupe.

Le domaine des réseaux de capteurs sans fils connaît un grand essor du fait de son large spectre d'applications (ex : bâtiments intelligents, suivi de cible, contrôle d'environnement, internet des objets). La majorité de ces applications nécessite l'optimisation des communications entre les capteurs pour garantir une bonne qualité de service tout

en réduisant le coût énergétique. Pour répondre à ces questions, le concept de coopération entre les caches des capteurs a été introduit grâce à son efficacité prouvée pour les systèmes mobiles.

Les travaux de Dimokas et al. ont exploré l'approche à cache coopératif [DKM08, DKTM11] en exploitant des nœuds de capteurs ayant des positions stratégiques dans le réseau. Ces nœuds sont chargés de transférer des paquets et de prendre des décisions de communication à bas coût temporel et énergétique grâce à leur proximité physique avec les autres nœuds. Le point faible de cette approche est que les performances du système dépendent beaucoup de la qualité des choix de ces nœuds médiateurs et de leur fiabilité.

Le principe de coopération a montré ses performances dans des environnements volatiles et instables tels que les réseaux mobiles ou sans fils. Nous étudions dans la section suivante les performances d'une telle approche dans un domaine plus stable et plus exigent en terme de performances.

2.2.3 L'approche coopérative pour les multicœurs

Les problèmes de gestion des données rencontrés dans les domaines précédents sont constatés dans le domaine des systèmes multicœurs sur puce. L'espace de stockage global étant limité par la surface de la puce, il est important que chaque cœur utilise de manière efficace tous les niveaux de cache auxquels il a accès. La latence d'accès aux données varie en fonction de leurs positions dans la puce. Il est donc nécessaire de prendre cela en compte lors du stockage des données.

Le principe de coopération entre les caches consiste à créer un espace de stockage logiquement partagé à travers l'agrégation de plusieurs caches privés. Chacun des cœurs de la puce, faisant partie d'une zone de coopération, a un accès étendu aux caches des autres cœurs de cette même zone. Les données stockées dans les différents caches sont accessibles via cet espace d'adressage commun.

Le mécanisme de coopération pour les systèmes multicœurs sur puce a été proposé dans les travaux de Chang et Sohi [CS07]. Il permet d'adapter dynamiquement l'utilisation des ressources par les différents cœurs en fonction de leur besoin en mémoire. La figure 2.2 représente une zone coopérative créée à partir des caches privés de 4 cœurs voisins.

Le mécanisme de cache coopératif est mis en œuvre à l'aide d'unités matérielles spécialisées dans le contrôle d'allocation des données dans les caches. La gestion des données stockées dans la zone coopérative est effectuée par une unité de contrôle de cohérence de cache. Dans le modèle de cache coopératif proposé par Chang et Sohi [CS06], la gestion de la cohérence est centralisée. Elle est assurée par un répertoire commun *CCE (Centralized Cooperative Engine)* qui comporte la duplication de toutes les informations de cohérence des caches $L1$ et $L2$ (figure 2.3). En cas de défaut d'accès au cache local, le *CCE* redirige la requête vers le cœur en possession de la donnée et cette dernière est ensuite transférée de cache en cache jusqu'au cœur demandeur.

Lors de l'éviction d'une donnée du cache $L2$ local, le CCE transfère la donnée sur un autre cache privé de la zone partagée. Afin de mieux utiliser l'espace de stockage

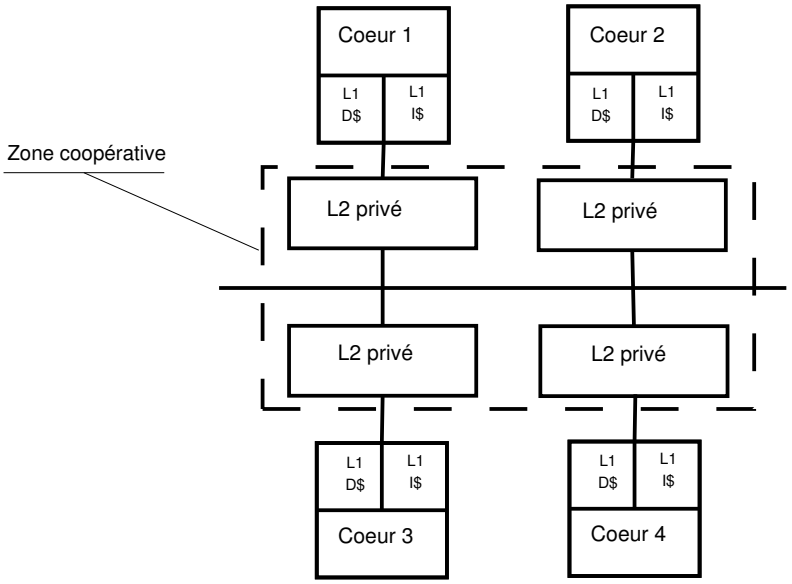


FIGURE 2.2 – Zone coopérative entre : Un espace de coopération entre quatre caches physiquement répartis de cœurs **voisins**. La zone coopérative (en pointillé) est définie comme l'agrégation virtuelle des caches répartis.

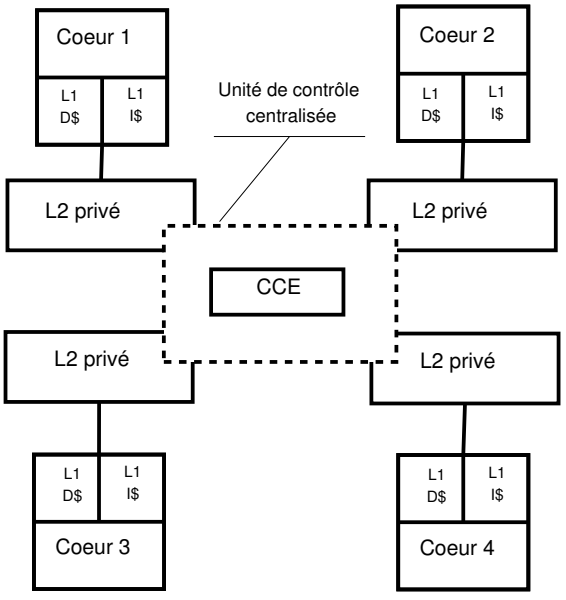


FIGURE 2.3 – Structure de cache coopératif à contrôle **centralisé**. L'ensemble des caches répartis est géré par une seule unité de contrôle.

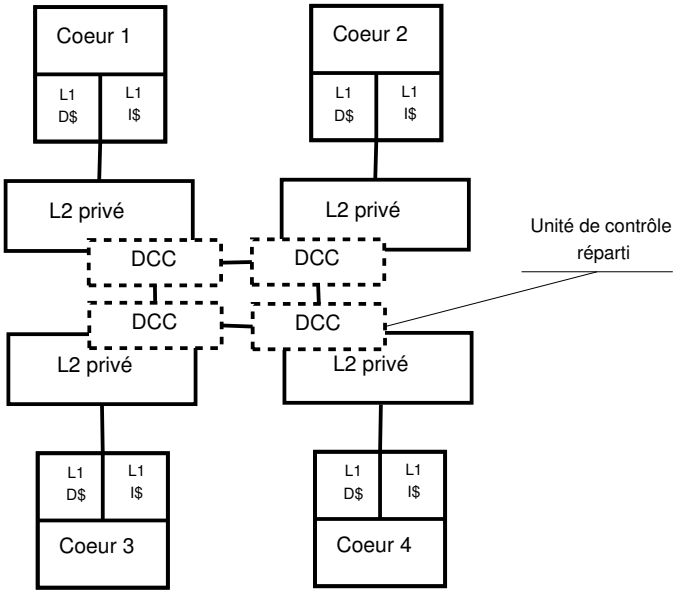


FIGURE 2.4 – Structure de cache coopératif à contrôle réparti. A chaque cache est associé une unité de contrôle permettant de gérer les données dans le cache.

commun, le contrôleur de cohérence utilise une politique de remplacement dite du *N-Chance Forwarding* [HGC10b], qui permet d'éviter qu'un bloc ne circule infiniment sur le réseau. La technique du *N-Chance Forwarding* consiste à associer un compteur à chaque bloc de donné. Tout bloc est donc autorisé à migrer N fois avant d'être éjecté hors puce. Ce compteur est initialisé à chaque fois que la donnée est rechargée sur la puce.

Cependant, l'approche des caches coopératifs à répertoire centralisé représente plusieurs limitations. La principale limitation rencontrée est le passage à l'échelle du répertoire centralisé. La deuxième limitation est liée à la consommation d'énergie au niveau de l'unité de contrôle centralisé dont la charge en nombre de vérifications par requête augmente avec le nombre de cœurs. Une nouvelle approche est proposée pour répondre à ces limitations. Il s'agit d'une stratégie de gestion de cohérence de données répartie appelée *DCC (Distributed Coopérative Caching)* [HGC08].

Le mécanisme du DCC est conçu principalement pour résoudre le problème de passage à l'échelle de la configuration centralisée. Le principe de l'approche répartie est de diviser l'unité de contrôle CCE en plusieurs unités réparties DCC, dont chacune s'occupe d'une plage d'adresses. Les DCC comportent des informations sur la distribution des blocs sur les différents cœurs (figure 2.4).

La section 2.3 suivante présente une nouvelle technique de coopération des caches pour des architectures manycœurs afin d'assurer un meilleur passage à l'échelle de la mémoire sur puce pour ce type d'architectures.

2.3 L'approche coopérative pour les manycœurs

L'utilisation optimale des performances computationnelles d'un système sur puce à plusieurs dizaines, centaines ou milliers de cœurs dépend entre autres du coût d'accès aux données dans sa mémoire. les systèmes manycœurs sont caractérisés par une hiérarchie mémoire constituée de 2 à 3 niveaux de caches par cœur en plus de la mémoire principale extérieure. Pour des limitations technologiques expliquées précédemment, les accès à la mémoire extérieure sont 10 à 15 fois plus chers que les accès au niveaux de caches disponibles sur la puce. Toutefois, une manière de réduire cette pénalité est de minimiser l'éjection des données hors-puce. Ceci revient à maximiser l'utilisation de l'espace mémoire global sur la puce.

Nous étudions dans cette section l'introduction des mécanismes de glissement pour les systèmes sur puce.

2.3.1 Description du mécanisme de cache élastique

Nous définissons dans cette section le mécanisme de *cache élastique*. Il définit une hiérarchie mémoire répartie et dynamique qui s'adapte de façon autonome au comportement de l'application. L'approche de cache élastique *(Elastic Cooperative Caching* développé dans les travaux de Herrero et al. [HGC10a] est une forme de coopération entre les caches répartis dans un système manycœurs.

Elle consiste à diviser logiquement les caches *LLCs* en deux zones : *Privée* et *Partagée* afin d'optimiser l'utilisation des caches. La zone *privée* stocke toutes les données éjectées du cache du niveau plus haut (ex. cache $L1$). Tandis que la zone *Partagée* accueille les données provenant du voisinage. Le partitionnement du cache LLC est adapté dynamiquement aux comportements des applications. Ceci permet d'avoir un espace cache totalement *privé* si toutes les applications ont les mêmes besoins en mémoire et un plus grand espace *partagé* si seulement certaines applications profitent des zones *partagées*.

Le mécanisme de cache élastique permet également de gérer la duplication des données dans les deux zones. Les données partagées sont dupliquées dans les zones *privées* où elles sont sollicitées. Cependant, les zones *partagées* ne contiennent que des copies uniques des données.

Cette approche est caractérisée par un comportement élastique, autonome et adaptatif du cache LLC. Elle assure également le passage à l'échelle via son aspect réparti. Le mécanisme de cache élastique tend à équilibrer l'utilisation de l'espace de stockage commun, en dépit de la différence en besoin mémoire des différents cœurs. Outre tous les mécanismes proposés au préalable, concernant le partage des ressources mémoire d'une structure à cache hiérarchique, l'aspect élastique des caches reste le seul qui considère le comportement dynamique des différentes applications. Pour concevoir une hiérarchie mémoire qui passe à l'échelle, le protocole du cache élastique utilise une structure répartie pour la gestion de la cohérence *DCE*. L'allocation des blocs migrants se limite au voisinage proche. Une autre spécificité du mécanisme élastique par rapport au mécanisme réparti de base est que l'on ne peut pas déplacer un bloc de donnée plus d'une

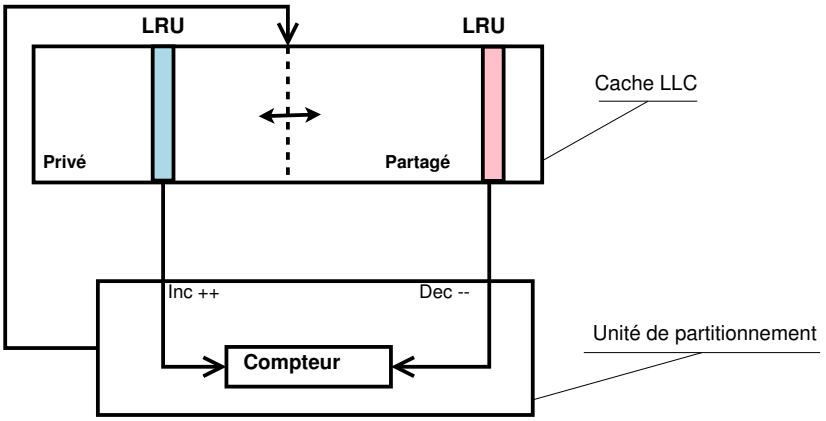


FIGURE 2.5 – L'unité de partitionnement de cache est constituée d'un compteur qui s'incrémente pour chaque accès au LRU privé et se décrémente pour chaque accès au LRU partagé. Une mise à jour cyclique est effectuée pour ajuster le partitionnement du cache aux valeurs des compteurs.

fois (*1-Chance Forwarding*). La structure des caches élastiques introduit deux unités de contrôle :

Unité de partitionnement : c'est la partie de contrôle qui assure le comportement adaptatif du partitionnement de cache LLC en deux zones (*privée/partagée*). Elle utilise un compteur d'accès qui s'incrémente à chaque succès d'accès au bloc LRU de la zone privée, et se décrémente pour chaque accès au bloc LRU de la zone partagée (figure 2.5). La mise à jour du partitionnement se fait de manière cyclique. Tous les N cycles processeur, le partitionnement du cache est modifié, en fonction de la valeur courante du compteur d'accès. Si la valeur du compteur est supérieure à un seuil maximal, le cache de la zone privée est élargi. Si cette valeur est inférieure à un seuil minimal, c'est le cache de la zone partagée qui est élargie. Pour toutes les valeurs moyennes le partitionnement ne change pas.

Unité d'allocation de blocs : l'allocation des blocs est un mécanisme qui permet de distribuer de manière efficace les blocs sur les cœurs destinataires. Les informations concernant le partitionnement du cache d'un cœur sont diffusées à tous les autres cœurs voisins, à chaque mise à jour. Ces informations sont ainsi utilisées pour décider du cœur destinataire de la prochaine éviction. L'unité d'allocation des blocs utilise un arbitre à *tourniquet* (*Round Robin*) avec un vecteur de bits que l'on met à jour avec les informations de partitionnement de chaque cache. Lorsqu'on déplace un bloc de la partition privée d'un cache local, l'arbitre choisit comme destinataire le cœur du bloc partagé suivant. De cette manière la distribution de la migration des blocs dans la zone de coopération se fait de façon équilibrée entre les cœurs d'un voisinage. La figure 2.6 décrit le fonctionnement de l'unité d'allocation. Le changement du partitionnement du cache d'un cœur nécessite la transmission d'un message de mise à jour à l'unité

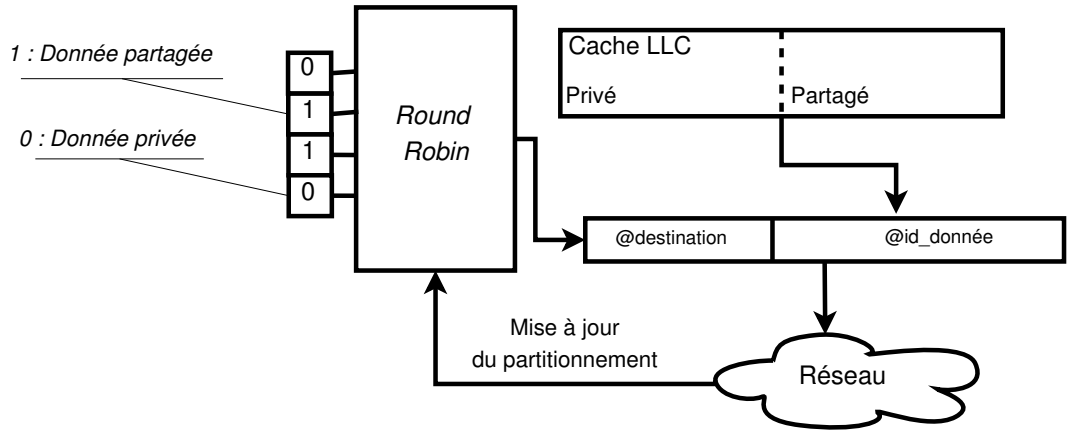


FIGURE 2.6 – Structure de l'unité d'allocation de blocs. Pour chaque mise à jour de partitionnement du cache LLC un message de mise à jour est envoyé à l'unité d'allocation des données pour être prise en compte dans le prochain choix de destinataire.

d'allocation qui prend en compte la nouvelle configuration dans ses futurs choix de cache destinataire des blocs de données déplacés.

Si une zone du cache est fortement sollicitée, il faudra attendre la prochaine mise à jour du partitionnement pour ajuster la taille de la zone sollicitée aux besoins du cœur demandeur.

La section suivante présente les différentes limitations du mécanisme de cache élastique notamment dans le contexte d'une concentration de la charge mémoire au niveau de la puce.

2.3.2 Discussion : limitations du modèle coopératif élastique

Le protocole de partitionnement de cache élastique est basé sur un seul compteur prenant en compte les accès réussis aux deux zones *privée* et *partagée* de la mémoire.

Lorsque la charge d'un cœur augmente alors que la charge de ses voisins baisse, l'espace *privé* de sa mémoire locale augmente aux dépens de la zone *partagée*. Cette dernière est inversement augmentée dans le cas d'une baisse de charge au niveau du cache local et d'une hausse de la charge mémoire dans le voisinage.

En revanche, si les deux zones sont fortement stressées, le protocole de partitionnement basé sur la valeur du compteur d'accès sera instable. Deux scénarios de fonctionnement sont possibles :

- Partitionnement statique : c'est le cas d'une succession d'accès alternés entre zone *privée* et *partagée*. Par conséquent, le cache va garder son partitionnement courant ce qui cause une dégradation des performances du système.

- Partitionnement instable : le compteur va osciller entre les deux seuils définis par l'unité de partitionnement. Cette oscillation va engendrer un dysfonctionnement dans la mise à jour des partitions du cache : éjection des données, grand nombre

de messages de mise à jour sur le réseau. L'unité de partitionnement du cache est donc pénalisée qui cause une forte dégradation des performances du système.

Un autre aspect concerne le choix du voisin coopératif. L'approche à cache élastique utilise l'approche *tourniquet* (*Round Robin*) pour définir le cache destinataire de la donnée déplacée du cache local. Cette stratégie vise à équilibrer la distribution des données autour d'un cœur mais ne tient pas compte de la charge des voisins destinataires. Si le voisin sélectionné est lui même stressé, le stockage des données dans son cache *partagé* se fait au dépend de ses données privées. Une connaissance de l'état de cœur destinataire permet de mieux adapter le processus de coopération à la répartition de la charge mémoire dans un voisinage.

Pour terminer, le mécanisme à cache élastique est plus performant dans le cas d'un déséquilibre de charge mémoire entre le cœur et ses voisins coopératifs. Mais il l'est beaucoup moins dans le cas d'une forte concurrence mémoire entre un cœur et son voisinage. Afin de mieux gérer la coopération entre les caches voisins, il faut donc prendre en compte les besoins mémoire du cœur local (*cœur demandeur*) aussi bien que le niveau de charge des cœurs voisins (*cœurs sollicités*). Le respect de ces contraintes permet de mieux équilibrer la coopération sans pénaliser les performances globales des caches.

Nous présentons dans la section suivante la technique de glissement de données proposée dans le cadre de cette thèse. Elle consiste à adapter l'utilisation de la mémoire cache sur la puce aux besoins mémoire des cœurs coopératifs.

2.4 Contribution : le mécanisme de glissement de données pour les manycœurs

Nous proposons dans cette section un mécanisme de coopération basé sur la technique de *glissement de données*. Cette approche garantit une meilleure adaptation à la répartition de la charge mémoire sur la puce tout en autorisant la coopération entre les différents caches.

Nous définissons d'abord l'approche proposée de *glissement de données*. Ensuite nous expliquons les détails d'implémentation dans un environnement manycœurs et à la fin nous discutons les performances d'une telle approche dans les différentes situations de voisinages stressés.

2.4.1 Description de l'approche coopérative par glissement de données

Le mécanisme de glissement de données proposé permet de gérer efficacement le stockage des données à travers une meilleure utilisation de l'espace mémoire disponible sur la puce. La technique proposée repose sur le principe de coopération entre les caches des voisins proches.

Afin de réduire la charge dans les zones stressées de la puce, le protocole de glissement autorise chaque cœur de solliciter le cache appartenant à un cœur coopératif de son

voisinage. À partir d'un niveau de saturation élevé du cache local, chaque cœur peut glisser ses données fortement sollicitées chez l'un de ses voisins les moins stressés.

À la réception d'une donnée déplacée le cœur accueillant vérifie la disponibilité de stockage dans son cache local. Deux cas de figure se présentent :

- Le cœur d'accueil dispose d'un espace libre et dans ce cas ses données locales ne bougent pas
- Le cache d'accueil est saturé, il doit donc migrer une donnée locale chez un voisin proche afin d'accueillir la nouvelle donnée reçues. Ce comportement est l'origine de l'appellation du protocole de glissement de données.

Le mouvement des données s'effectue de cœur en cœur tout en gardant chacune des données à une distance d'un pas de son cœur propriétaire. La donnée ne peut effectuer qu'un seul déplacement vers les voisins directs de son cœur propriétaire. Les données sont propagées dans la puce de manière à réduire la concentration de la charge dans les zones stressées. La distance de migration des données, étant fixée à 1, garantit la proximité d'accès entre les données déplacées et leur cœur propriétaire.

Le protocole de glissement de données a d'une part, l'avantage d'éviter l'éjection des données hors-puce lorsqu'il n'y a plus d'espace de stockage dans les caches privés des cœurs stressés. Ceci revient à réduire le nombre de défauts de cache et donc à améliorer les performances du système. D'autre part, comme les données sont stockées dans le voisinage direct de chaque cœur, le temps d'accès aux données glissées reste inférieur au temps d'accès à la mémoire extérieure.

2.4.2 Implémentation du protocole de glissement de données

Le mécanisme de glissement de données décrit dans cette section, autorise à chaque cœur de stocker ses propres données dans un cache voisin au lieu de les éjecter à l'extérieur de la puce. Différentes implémentations de cette approche existent dans la littérature (section 2.2) où le dernier niveau de cache de chacun des cœurs coopératifs est partagé en deux zones : privée et partagée. La zone privée ne peut accueillir que des données privées, et celle partagée est mise à la disposition de l'espace coopératif. Le partitionnement du cache local entre les deux types de données nécessite une gestion dynamique de la frontière entre les deux zones afin de prendre en compte la différence de charge entre les cœurs coopératifs.

La suite de cette section présente les détails d'implémentation du protocoles de glissement de données.

2.4.2.1 Description de l'algorithme de glissement de donnée

Le protocole de glissement de données, introduit dans cette section, propose une solution adaptative sans partitionnement de cache. Le cache d'un cœur est d'abord chargé par ses données privées. Lors de la sollicitation d'un cœur voisin, les données locales sont remplacées par les données provenant du voisinage. Une donnée peut être

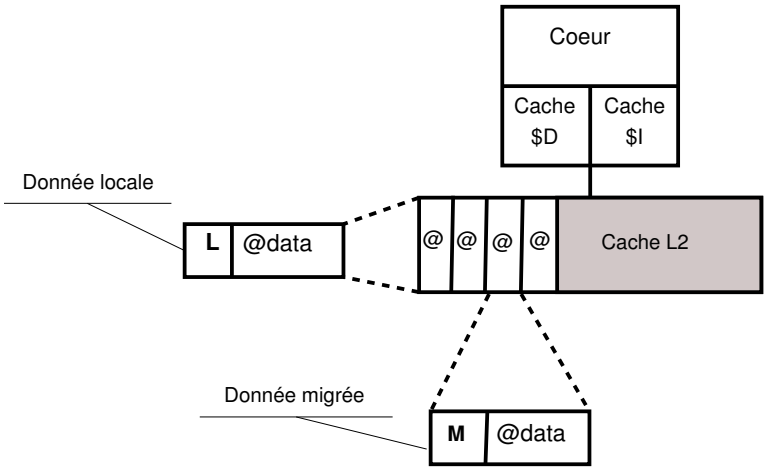


FIGURE 2.7 – Étiquetage des blocs de données dans le cache *LLC* d'un cœur. Dans le cache, chaque donnée est étiquetée selon si elle appartient au cache local (L) ou à un cœur voisin (M).

soit privée (stockée dans le cache de son propre cœur), soit hébergée (stockée dans un cache voisin). La figure 2.7 présente la structure d'un cache privé.

Dans l'approche proposée les zones coopératives sont créées à partir des voisinages proches. Un voisinage est défini comme un ensemble de cœurs adjacents connectés directement sur le réseau sur puce. Le périmètre du voisinage d'un cœur dépend de la topologie réseau de l'architecture. Pour simplifier le modèle, nous considérons une grille *2D* où chaque cœur forme un voisinage avec ses 4 voisins adjacents (figure 2.8).

Le mécanisme de glissement est destiné à la gestion des points de stress dans la puce. Un point de stress correspond à une zone de la puce subissant une forte sollicitation de sa mémoire cache. Afin de réduire la charge au niveau de ces zones, les cœurs concernés glissent leur propres données dans leurs voisinages respectifs.

À la réception d'une nouvelle donnée, le cœur saturé envoie une requête de glissement vers un cœur de son voisinage. Ce dernier doit satisfaire la requête reçue. D'un coté, la donnée à stocker doit directement être mise à disposition du cœur demandeur pour répondre à sa demande d'accès au plus tôt. Pour ce faire, un espace de stockage flottant est réservé dans tous les caches des cœurs actifs. Cet espace flottant prend la place du dernier bloc migré d'où son nom. Le voisin demandeur n'a donc pas besoin d'attendre le retour des cœurs de son voisinage pour pouvoir réceptionner la nouvelle donnée. D'un autre coté, le voisin recevant la demande d'hébergement utilise son espace flottant pour accueillir la donnée glissée. Dans le cas où le cœur d'accueil est saturé il transfère une donnée de son cache local vers son propre voisinage afin de créer un nouvel espace flottant. Le processus de propagation des données s'arrête au premier voisinage à faible charge mémoire considéré comme non stressé.

La figure 2.9 décrit de façon détaillée le processus de glissement :

1. La donnée a est chargée dans le cache local du cœur 1 occupant l'espace flottant

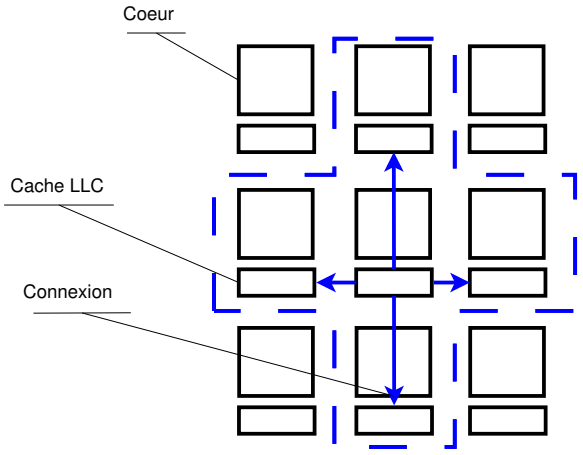


FIGURE 2.8 – Voisinage constitué d'un cœur central et de ses quatre voisins directs. Une zone de coopération est créée entre les différents caches des cœurs du voisinage.

libre destiné à la réception des nouveaux blocs de données chargés de la mémoire extérieure.

2. La donnée b est glissée vers le cache du cœur 2. Ce dernier étant à son tour saturé, il enchaîne le processus de glissement en envoyant la donnée c vers son voisin le moins stressé (cœur 3).

3. Le cœur 3 est libre. Il reçoit la donnée c dans son cache local.

4. Le glissement de donnée s'arrête au premier cœur à cache libre.

5. Si aucun cœur n'est disponible pour coopérer, ce qui correspond au cas où toute la puce est tressée, les données sont éjectées vers la mémoire extérieure.

2.4.2.2 Implémentation des compteurs d'accès

Deux questions de prise de décision se posent autour de la migration des données : le choix de la donnée à déplacer et le choix du voisin destinataire. Pour répondre à ces questions, nous proposons deux techniques développées dans la suite de ce chapitre. Ces deux techniques considèrent principalement la fréquence d'accès aux données pour le choix de la donnée à glisser, et le niveau de saturation des cœurs voisins pour le choix du voisin destinataire.

Le choix de la donnée à migrer ainsi que le choix du voisin destinataire reposent sur le niveau de stress de chacun des cœurs d'un voisinage. Il est plus facile pour un cœur d'estimer le niveau de sollicitation de son cache local. Tandis qu'il est plus souvent coûteux d'avoir les informations sur le niveau de stress d'un autre cœur. Nous proposons dans cette section une technique pour estimer le niveau de stress local ainsi que celui du voisinage. Cette technique utilise deux types de compteur (figure 2.10) :

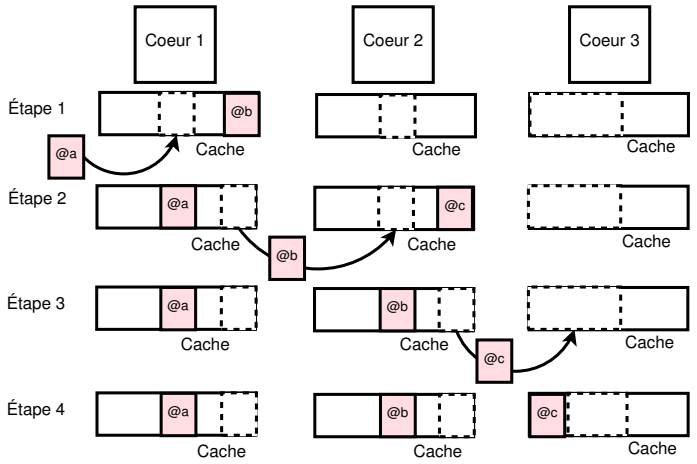


FIGURE 2.9 – Processus de glissement des données. Lorsqu'un cœur migre une donnée de son cache local vers un voisin, celui-ci l'accueille dans son espace flottant. Si le cache d'accueil est saturé, le cœur glisse une donnée locale vers un autre voisin afin de libérer de nouveau un espace flottant et préparer l'accueil de futures données. La propagation des données se fait de proche en proche jusqu'à atteindre une zone à faible charge mémoire.

- Un compteur local *LHC (Local Hit Counter)* : Le compteur local s'incrémente à chaque accès réussi aux données privées.

- Des compteurs de voisinage *NHC (Neighbor Hit Counter)* : Un compteur par voisin. Chacun des compteurs s'incrémente à chaque accès à une donnée hébergée dans le cache local du cœur qui l'accueille.

L'utilisation de ces compteurs permet à chaque cœur de comparer localement sa charge et celle de ses voisins sans avoir besoin d'une vision globale sur l'ensemble des cœurs ni d'échanges de messages d'information sur le réseau. Un autre avantage des compteurs utilisés est dû à leur faible coût matériel.

Le stockage des données est géré de manière répartie où chaque cœur prend ses décisions en tenant compte de la charge de son cache local et de celle des autres caches coopératifs. Deux types de données sont distingués dans le cache en fonction de leur cœurs d'origine : données locales *L (Local)* ou données issues des cœurs voisins *M (Migrated)*.

Le protocole de glissement peut ainsi distinguer entre données privées et données hébergées. Le partage de l'espace cache se fait en fonction du niveau d'utilisation de chacun de ces deux types de données. Le choix du voisin destinataire est basé sur la charge mémoire de chacun des cœurs coopératifs.

La section suivante explique les deux techniques utilisées pour implémenter le mécanisme de glissement de données. Elles sont basées sur les compteurs d'accès aux données : *LHC* et *NHC*.

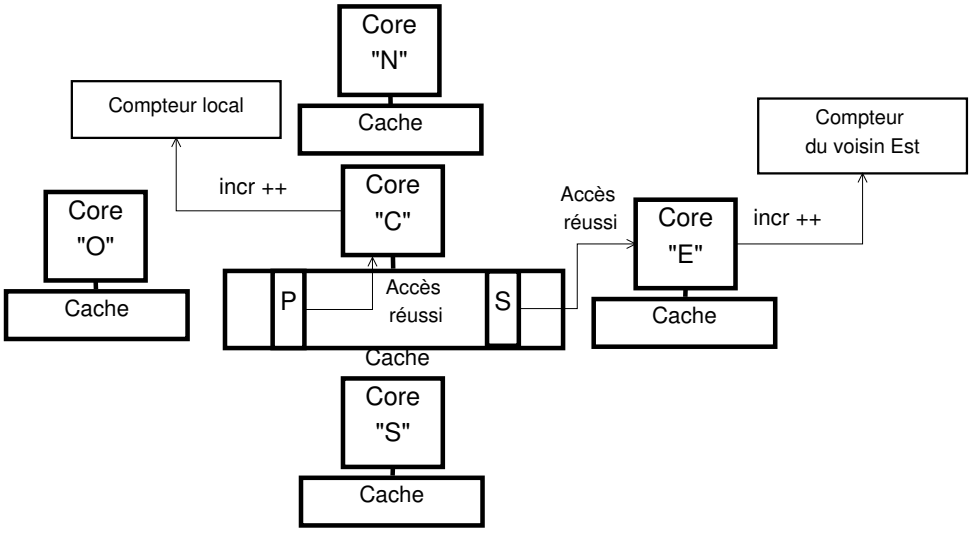


FIGURE 2.10 – À chaque cœur du voisinage est associé un compteur qui s'incrémente pour tout accès réussi effectué par ce cœur à sa donnée dans le cache coopératif local.

2.4.2.3 Politique de remplacement

Le protocole de glissement de données permet de réduire la charge dans les zones stressées en glissant les données de chaque cœur stressé dans son voisinage jusqu'à atteindre des cœurs à caches libres.

Pour chaque opération de glissement le cœur doit choisir une donnée à migrer. Le choix est effectué entre les données locales stockées dans son cache privé et les données issues d'une migration en provenance des cœurs voisins.

Le choix de la donnée à glisser doit prendre en compte le taux d'utilisation de chaque groupe de données afin d'éviter de pénaliser les cœurs propriétaires.

La technique de remplacement proposée est basée sur les compteurs de charge définis dans le paragraphe précédente 2.4.2.2. Pour répondre à une requête de stockage d'une donnée trois cas se présentent :

- *LHC est fortement supérieur à la somme des NHCs* : le cœur effectue plus d'accès à ses données privées que l'ensemble de ses voisins aux données hébergées dans son cache local. Les données privées sont donc prioritaires. Une nouvelle donnée remplace la donnée la moins récemment utilisée parmi les données hébergées.

- *LHC est fortement inférieur à la somme des NHCs* : les accès en provenance des cœurs voisins sont plus élevés qu'aux sollicitations locales du cache. Les données hébergées sont donc plus prioritaires que les données locales. La donnée la moins récemment utilisée (LRU) parmi les données locales sera glissée vers un des voisins directs.

- *LHC est équivalent à la somme des NHCs* : c'est le cas d'une double sollicitation du cache local par des accès fréquents aux données privées et à celle hébergées. Il

est difficile de décider de la priorité de remplacement dans une telle situation. Dans l'approche de partitionnement du modèle élastique, le cas d'une forte sollicitation des deux zones, privée et partagée, engendre une oscillation entre mode privé et mode partagé. Le cache est donc amené à éjecter les données des deux zones à chaque oscillation. En revanche, le modèle de remplacement proposé pour le protocole de glissement de données consiste à favoriser le stockage des données voisines afin de répartir la charge mémoire sur une zone plus large. L'idée du protocole proposé est de faire en sorte que chaque cœur stocke ses données dans son voisinage direct. À chaque requête de stockage d'une donnée voisine, le cœur fait le choix de libérer son cache en déplaçant une donnée privée. À son tour, cette dernière est envoyée vers un cœur voisin qui, étant stressé, va procéder de la même manière. Le glissement se fait ainsi de cœur en cœur jusqu'à atteindre une zone à faible charge mémoire.

Cette technique de remplacement favorise le glissement de données en engendrant une propagation des données privées de la zone stressée vers une zone moins chargée de la puce tout en respectant la contrainte de proximité (*1-chance forwarding*). Dans cette approche, chaque donnée a une seule chance de migrer. Cette hypothèse permet d'éviter une propagation infinie des données, ce qui peut engendrer une surcharge du réseau sur puce. Le glissement est donc limité au plus proche voisinage de chaque cœur.

Ce processus de propagation permet de maintenir les données le plus longtemps possible sur la puce ce qui diminue le nombre de défauts de cache sur la puce.

Grâce à la proximité physique des cœurs et de leurs données, la latence d'accès reste réduite et relativement proche du coût d'accès au cache local.

La politique de remplacement basée sur la comparaison des compteurs évite d'éjecter les données fréquemment utilisées. Elle n'active ainsi le glissement de données qu'au-delà d'un seuil de forte sollicitation du cache local. La valeur du seuil de glissement permet de contrôler le processus de migration et de ne le déclencher qu'en cas de détection d'un grand niveau de stress dans une zone. Le choix de cette valeur est très important car il permet d'éviter ces deux cas de fonctionnement défaillants du protocole :

- Glissement précoce : Si la valeur du seuil est très petite, un grand nombre de cœurs non stressés va déclencher la migration de ses données de manière précoce. Par conséquent, le réseau sur puce va être chargé par un trafic élevé et pas forcément nécessaire.

- Glissement tardif : Si la valeur du seuil est très grande, la migration des données dans les zones stressées va se déclencher tardivement. Cette décision tardive peut coûter cher en nombre de défauts de cache et par conséquent dégrade les performances globales du système.

Il existe différentes pistes pour choisir le seuil de stress au niveau d'un cœur ce qui laisse une large possibilité d'optimisation du protocole proposé. Il est notamment possible de le calculer en fonction de la taille du cache local, de la fréquence d'accès aux données et en fonction du nombre de données manipulées par chaque cœur. Une autre

solution est de définir la valeur du seuil de façon adaptative en l'ajustant dynamiquement au comportement de l'application.

Nous avons fait le choix de définir le seuil de stress de manière empirique car cela nous a permis de l'adapter au fonctionnement des applications étudiées et en fonction de nos besoins expérimentaux.

2.4.2.4 Choix du meilleur voisin

Nous présentons dans ce paragraphe une deuxième approche permettant de décider du voisin destinataire. Dans le mécanisme de cache coopératif élastique (§ 2.3.1) l'unité d'allocation des blocs se base sur les données fournies par l'unité de partitionnement, tout en suivant une stratégie uniforme de *tourniquet* (*Round Robin*). Les cœurs stockant le plus de données partagées reçoivent équitablement plus de données éjectées du cache local vers le voisinage.

Dans une configuration où il faut migrer un grand nombre de données, le *tourniquet* (*Round Robin*) n'est plus efficace. Lorsque le transfert des données ne prend pas en compte l'état du cœur destinataire, il risque de pénaliser des voisins stressés alors que d'autres voisins sont libres. Les données peuvent alors être envoyées vers des voisins possiblement plus stressés que d'autres.

La politique du choix du *meilleur voisin*(définition 6) proposée dans cette section vise à répartir les requêtes de stockage de façon non uniforme sur les différents voisins en prenant en compte la charge de travail de chacun. Ceci est rendu possible grâce à l'utilisation des compteurs.

Définition 6 (Meilleur voisin)**.** Le meilleur cœur dit *Best Neighbor* est défini comme étant celui dont le compteur d'accès associé est minimal.

Le contrôle de cache utilise les mêmes données pour décider du bloc à remplacer et sa destination via un processus de comparaison simple. Cette technique améliore l'efficacité du mécanisme de coopération, en évitant de solliciter les cœurs en pic de charge. Elle évite donc de provoquer des évictions supplémentaires. Le cœur *Best Neighbor* est finalement celui le plus disponible du voisinage. Comme dans notre protocole de glissement, un cœur ne peut transférer ses données privées qu'à ses voisins proches. Si son compteur associé chez l'un de ses voisins est faible, ceci signifie qu'il n'a pas demandé de l'aide chez ce voisin ou au moins que sa demande d'aide est plus faible par rapport aux autres cœurs. Le gain obtenu avec cette technique par rapport à la technique du cache élastique est présenté plus tard dans la partie expérimentale.

2.5 Extension du protocole de glissement à l'aide du modèle physique masse-ressort

Le protocole de glissement proposé autorise à chaque cœur de stocker ses propres données chez ses voisins directs. Comme décrit dans le paragraphe 2.4.2, la distance entre un cœur et ses données ne doit pas dépasser un pas (*1-Chance Forwarding*). Le

glissement à un pas a l'avantage de réduire le temps d'accès aux données sur la puce en limitant la distance entre le cœur et ses données. Cette condition a par ailleurs l'inconvénient de limiter la durée de vie de la donnée sur la puce ce qui revient à éjecter plus de données à l'extérieur.

Le temps d'accès à la mémoire principale étant beaucoup plus lent comparé au temps d'accès à un cœur distant sur la puce [VGRN08], l'augmentation du rayon de glissement des données permettra d'obtenir de meilleures performances.

Nous proposons dans cette section une technique du choix du degré de liberté accordé à chaque donnée en considérant deux contraintes :

- Le coût de récupération d'une donnée distante doit rester inférieur au coût d'accès à la mémoire extérieure
- Le rayon de migration des données doit s'adapter à la charge mémoire de la zone stressée. Il doit augmenter lorsque le niveau de stress est élevé pour maintenir les données sur la puce et diminuer dans le cas inverse pour respecter la première contrainte.

Cette section propose une extension du protocole de glissement de données. Elle est inspirée du modèle physique *masse-ressort*. Le modèle a été développé en collaboration avec *Stéphane Louise*, ingénieur chercheur au *CEA*.

2.5.1 Présentation du modèle physique masse-ressort

Nous introduisons dans ce paragraphe le protocole de glissement à rayon variable en se basant sur le modèle physique *masse-ressort*.

Par analogie avec ce modèle, chaque donnée est considérée comme une masse attachée à son cœur propriétaire par un ressort. Ce dernier étant caractérisé par une constante de raideur, il permet de contrôler la distance entre les données et leurs cœurs propriétaires. Pour chaque déplacement d'une donnée d'un point A à un point B sur la puce, le protocole met à jour la nouvelle distance que peut prendre la donnée de son point d'origine. Cette distance est calculée en fonction de :

- L'historique des déplacements effectués par la donnée : ceci permet de respecter la contrainte de proximité d'accès à la donnée en prenant en compte ses mouvements de migrations à chaque nouveau déplacement.
- La constante de raideur associée au ressort : ce qui permet de donner plus ou moins de liberté aux mouvements des données sur la puce. Ce paramètre est commun à tous les déplacements de la donnée durant sa disponibilité sur la puce.
- La différence de potentiel entre la source et la destination : le potentiel d'un cœur correspond à sa charge mémoire. Par analogie avec les mouvements d'une masse dans un bac à sable, la différence de potentiel entre les cœurs voisins permet d'orienter le mouvement des données vers les zones les moins chargées (voir figure 2.11).

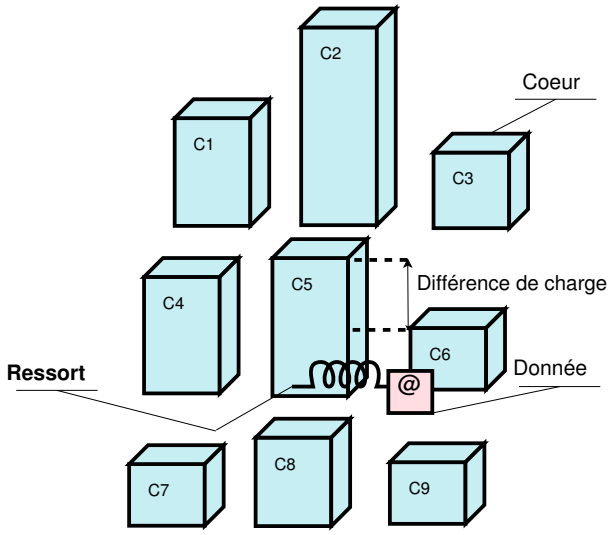


FIGURE 2.11 – Modèle du *masse-ressort* : une donnée est attachée à son cœur propriétaire par un ressort. Le sens de migration des données est orienté vers les cœurs les moins chargés. la différence de potentiel entre les cœurscorrespond à la différence de charge de leurs caches locaux.

L'utilisation d'un tel modèle a pour objectif de donner plus de liberté de déplacement aux données. La prise en compte de l'historique des déplacements de chaque donnée ainsi que l'état de charge des cœurs destinataires assurent une meilleure adaptation de la propagation des données à la répartition de la charge mémoire dans la puce.

Une autre propriété physique exploitée dans ce modèle est l'élasticité du ressort. Elle permet à une donnée de retourner vers son point d'origine en cas de baisse de charge. Le sens de migration s'adapte à la répartition de la charge des cœurs dans le voisinage.

Le mouvement des données s'effectue progressivement de cœur en cœur. La propagation de la donnée continue, tant que les cœurs d'accueil sont saturés, jusqu'à atteindre la distance maximale calculée pour le déplacement en cours. Au-delà de cette distance, la donnée est éjectée hors-puce.

Le paragraphe suivant explique le modèle mathématique issu de l'analogie entre le glissement des données sur la puce et le modèle physique du *masse-ressort*.

2.5.2 Modèle mathématique du masse ressort

Le mécanisme de glissement de données à N-pas de migration est moins efficace si la donnée est déplacée très loin de son cœur propriétaire. Le modèle de *masse-ressort* permet de prendre la décision sur :

- Le degré de liberté de la donnée en utilisant la notion de constante de raideur du ressort associé à chaque donnée

- Le schéma de migration des données en utilisant la notion de potentiel qui correspond à la charge du cache

Notons Δh la différence de potentiel entre le cœur source et cœur destination. Une masse attachée à un ressort est poussée de son point d'origine par la force de gravité due à la pente. La projection du déplacement de la donnée sur l'axe des ordonnées x donne l'équation suivante :

$$m\frac{d^2x}{dt^2} = \alpha mg\Delta h - c\frac{dx}{dt} - kx \tag{2.1}$$

Avec k : la constante de raideur du ressort, c : la constante de fluidité, m : la masse.

Le changement de constante simplifie l'équation pour obtenir :

$$\frac{d^2x}{dt^2} + c'\frac{dx}{dt} + k'x = \alpha'\Delta h \tag{2.2}$$

Le problème est rapporté à la résolution d'une équation du deuxième degré. Dans le cas critique où $c'^2 - 4k' = 0$, notons $a = c'/2$. Nous obtenons donc le modèle :

$$\frac{d^2x}{dt^2} + 2a\frac{dx}{dt} + a^2x = b\Delta h \tag{2.3}$$

Une discrétisation temporelle du modèle où D_i correspond à la distance associée à la donnée à l'instant $t = i$ (discret, $i \in \mathbb{N}$), on obtient :

$$D_i = 2AD_{i-1} - A^2D_{i-2} + B\Delta h \tag{2.4}$$

Comme le montre l'équation la distance du prochain déplacement dépend de l'historique des deux précédentes distances, de la constante de raideur du ressort (incluse dans les deux constantes A et B) définissant le degré de liberté de la donnée, et de la différence de charge (*potentiel*) entre le cœur source et le cœur destination. L'intérêt d'utiliser un modèle mathématique discrétisé est de profiter de sa simplicité algorithmique. Il utilise en effet un nombre limité d'opérations élémentaires peu coûteuses en terme de calcul.

reflète la simplicité algorithmique de l'utilisation d'une telle analogie avec le modèle physique.

2.5.3 Implémentation

L'implémentation de l'algorithme basé sur le modèle décrit précédemment se fait de manière répartie. Grâce à la simplicité du modèle mathématique composé de 4 multiplications et de 2 additions le coût d'implémentation reste raisonnable. Le calcul de la distance utilise des données locales en plus de l'historique des deux dernières distances parcourues ce qui représente un faible surcoût mémoire.

L'algorithme suivant présente les différentes étapes de décision pour le calcul de la distance et le choix de la direction de migration de la donnée.

Algorithme 1 Calcul de la distance D_i

1. Calcul de la distance à l'étape i pour chacun des N voisins :

$$D_i \leftarrow 2 * A * D_{i-1} - A^2 * D_{i-2} + B * \Delta h$$

Avec :

$$A \leftarrow \frac{1}{K}$$

$$B \leftarrow \frac{g}{K}$$

où g : constante de pesanteur et K : constante de raideur du ressort.

2. Choix de la plus grande distance :

$$distance \leftarrow max(D_i)$$

3. Transfert de la donnée vers le voisin correspondant à la plus grande distance.

Les informations concernant la charge des voisins directs sont stockées localement à travers les compteurs de charge du voisinage NHC définifis dans la section 2.4.2.2. Ces compteurs sont incrémentés à chaque réception d'une requête d'accès aux données hébergées dans le cache local, ce qui permet d'avoir une visibilité sur l'état de stress des cœurs voisins sans effectuer des communications supplémentaires.

2.6 Discussion

Nous avons présenté dans ce chapitre le protocole de glissement de données à rayon fixe et son extension basée sur le modèle physique du masse-ressort. Le processus de migration des données est guidé par des valeurs de seuil qui sont définies dans notre cas de manière empirique.

Le choix de ces seuils peut jouer un rôle important dans l'amélioration des performances du protocole. Par exemple, si le seuil qui permet de déclencher le processus de migration en fonction du niveau de stress du cœur est très bas, ceci peut engendrer des mouvements de données qui ne sont pas forcément nécessaires et donc dégrader les performances d'exécution. Pour le modèle masse-ressort, par exemple le paramètre correspondant à la constance de raideur associée au ressort dans le modèle physique est capable de limiter les distances de migration des données sur la puce.

Ces seuils peuvent être définis en fonction de la charge mémoire globale au niveau de la puce mais aussi en fonction de la capacité de stockage des caches. Si les valeurs choisies sont bien adaptées à la répartition de la charge sur la puce et des tailles de caches les performances des protocoles de données seront meilleures.

Chapitre 3

Étude analytique des protocoles de glissement de données

Les protocoles de glissement de données et de masse-ressort présenté dans le chapitre 2 sont destinés à la gestion des voisinage stressés dans une puce. Ils permettent de mieux répartir la charge mémoire entre les caches voisins pour alléger le stress mémoire des cœurs les plus chargés.

Ce chapitre présente des résultats de validation de ces protocoles en les comparant à d'autres protocoles de l'état de l'art. La première section 3.1 décrit le contexte expérimental. Les résultats d'analyse des performances du protocole de glissement de données comparé à d'autres protocoles sont discutés dans la section 3.2.

3.1 Contexte expérimental

Nous décrivons dans la sous-section 3.1.1 la plate-forme de validation, le protocole *baseline* est décrit dans la sous-section suivante 3.1.2. Enfin, dans les scénarios d'accès aux données utilisés dans l'analyse des protocoles sont présentés dans la sous-section 3.1.3.

3.1.1 Plate-forme de validation

La plate-forme de validation utilisée est basée sur une approche analytique. Cette approche nous permet d'obtenir des statistiques sur le trafic généré sur la puce en comptabilisant tous les messages échangés à chaque fois qu'une requête d'accès est satisfaite.

Notre outil de validation prend en entrée la trace des accès mémoire aux données partagées dans l'application. Cette trace mémoire est générée grâce à l'outil d'instrumentation *Pin* [BCC+10]. *Pin* est une structure logicielle d'analyse dynamique du code binaire. Il utilise la technique de compilation à la volée *JIT(Just-In-Time)* basée sur la représentation intermédiaire du code appelée *Byte-code*. Comme pour les autres outils d'analyse dynamique (ex. Valgrind [NS07], Dynamo [BGA03, BDB00]), *Pin* permet

d'analyser le code binaire des applications indépendamment du langage de programmation, par contre le résultat obtenu est fort dépendant de la plate-forme d'exécution.

Plusieurs outils d'analyse de programme sont fournis par *Pin/Pintools* allant du plus simple outil qui calcule le nombre d'instructions à d'autres outils beaucoup plus élaborés. Nous utilisons dans notre plate-forme de validation une version modifiée de l'outil *PIN/Pinatrace* [BCC+10]. Cette version permet de noter tous les accès aux données partagées, avec des détails d'information sur le type d'accès, la taille de la donnée ainsi que l'identifiant du cœur ayant effectué l'accès. Le listing 3.1 est un exemple de trace mémoire résultant de l'analyse d'un programme à l'aide de *Pinatrace*.

```
0xb5d79c80: W 0x8057284 4 0 3
0xb5d79c60: W 0x8057274 4 0 3
0xb5d78cb7: R 0x8057280 4 0 1
0xb5d78ce9: R 0x8057274 4 0 1
...
```

Listing 3.1 – Extrait d'une trace mémoire. Chaque ligne représente les informations d'un accès à une donnée

Les champs représentés dans la trace correspondent respectivement à l'adresse de l'instruction, le type d'accès (Lecture ou Écriture), l'adresse de la donnée, la taille de la donnée, l'indicateur d'instruction préchargée (non utilisé dans notre cas) et enfin l'identifiant du cœur qui a effectué l'accès.

La trace mémoire retournée par *Pinatrace* est issue d'une exécution possible du programme et pas la seule. Pour un même programme analysé, chaque nouvelle exécution peut générer une nouvelle trace mémoire en fonction des données initiales ou bien du résultat d'une condition d'accès à une branche du programme. La plate-forme d'exécution peut également faire varier la trace de sortie à cause des entrelacements entre les différents fils d'exécution. Dans le cadre de cette étude et pour des raisons d'absence d'une plate-forme manycœurs, nous avons effectué nos exécutions sur un ordinateur de bureau core *i*3. Ceci ne nous permet pas d'avoir un niveau de parallélisme élevé mais nous permet d'obtenir les différents comportements d'accès mémoire ce qui nous intéresse le plus.

L'évaluation des différents protocoles de cohérence étudiés est basée sur l'exploration de ces traces mémoire. Nous avons donc développé un outil de simulation analytique appelé le *Cache Validator*. Cet outil permet de simuler le comportement du protocole de cohérence affecté à chaque accès mémoire. Le *CacheValidator* permet par la suite de collecter les informations sur les communications point à point générées par le protocole choisi. L'outil peut distinguer différents types de messages échangés entre les cœurs de la puce (ex. Messages de lecture, Messages d'écriture, Messages de contôle). Il permet également de changer la configuration de l'architecture cible selon différents paramètres :

- Taille de la puce (Nombre de cœurs)
- Topologie du réseau sur puce (ex. maille 2D, Tore)

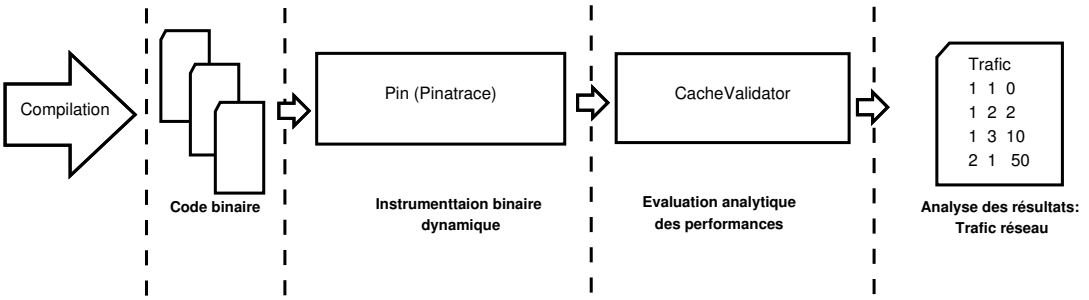


FIGURE 3.1 – Schéma de l'étude analytique des protocoles de gestion des données. Ce processus est composé d'une phase d'extraction des accès mémoire à l'aide de l'outil *Pinatrace* et d'une phase d'analyse du trafic réseau effectuée par le *Cache Validator*.

Le processus de validation utilisant le *Cache Validator* est basé sur l'évaluation du trafic sur la puce ainsi que la répartition de la charge mémoire qui en résulte à travers les statistiques des communications obtenues.

Pour résumer, le schéma 3.1 décrit le processus de validation à travers trois étapes principales :

1. Analyse du code binaire par instrumentation dynamique à l'exécution et génération d'une trace d'accès mémoire.
2. Évaluation du protocole de cohérence et des mécanismes de gestion de stockage associés par simulation des accès mémoire.
3. Analyse des données statistiques décrivant le trafic sur puce et interprétation des résultats obtenus.

Ce schéma expérimental permet de réaliser une évaluation préliminaire des mécanismes proposés en les comparant à certains mécanismes coopératifs de l'état de l'art.

La sous-section suivante décrit les résultats de comparaison d'une première évaluation basée sur le trafic sur la puce.

3.1.2 Protocole de cohérence *Baseline*

Il existe plusieurs modèles de gestion de cohérence pour les systèmes répartis. Nous avons fait le choix d'utiliser un protocole de référence nommé *baseline*. Il appartient à la famille des protocoles à répertoire proposé par Tang [Tan76] et Censier et al. [CF78] (*en anglais Directory-based protocols*) largement utilisés pour des systèmes à mémoires réparties [BKT07] comme la *SGI Origin* [LL97] et la *Alpha 21364* [MBL+01]. Chaque cœur est constitué en plus de la hiérarchie mémoire, d'interface réseau et mémoire et d'un répertoire de contrôle implémentant le protocole de cohérence des données accessibles directement via son contrôleur mémoire.

Le système *baseline* à répertoire associe à chaque donnée un nœud référent appelé *HomeNode*. Ce dernier est défini par un simple *Round Robin* ou bien à l'aide d'une

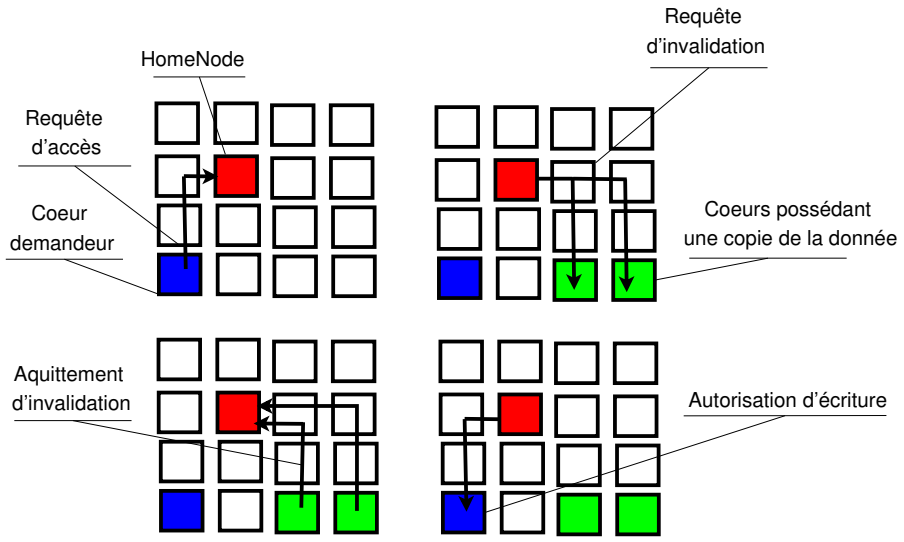


FIGURE 3.2 – Le protocole *Baseline* est un protocole à répertoire où tous les accès à une donnée passe par le nœud référent *HomeNode*. Une demande d'accès est envoyée à ce dernier qui envoie une requête d'invalidation à tous les cœurs possédant une copie de cette donnée. Il envoie ensuite un message de contôle autorisant l'accès à cette donnée avec les informations sur l'emplacement mémoire de sa copie la plus récente.

fonction de hachage. Les informations de cohérence sont stockées sous forme de vecteurs de $N+2$ bits triés par adresse mémoire où N représente le nombre de cœurs sur la puce et 2 les bits d'état de la donnée.

Pour le premier champs composé de N bits, la présence d'un 1 sur le $n^{\text{ième}}$ bit signifie qu'une copie de la donnée est présente dans le cache du $n^{\text{ième}}$ cœur, ce bit est mis à 0 dans le cas contraire.

Concernant le champs d'état, il s'agit des quatre états du modèle *MESI* qui sont : *Modified* (Donnée modifiée), *Exclusive* (Donnée en accès exclusif), *Shared* (Plusieurs copies valides dans différents caches) et *Invalid* (Donnée invalide et ne peut pas être utilisée).

La figure 3.2 illustre le déroulement d'une requête d'écriture dans une puce de $4*4$ cœurs. 1) Le cœur demandeur envoie un message au *HomeNode* en charge de la donnée en question. 2) Le *HomeNode* vérifie le vecteur de présence de la donnée et transfère la requête aux cœurs possédant une copie valide de cette donnée. 3) Ces cœurs invalident la donnée sur leur cache et renvoient un message d'acquittement. 4) Finalement, le *HomeNode* accorde la permission d'écriture au cœur demandeur. En cas d'absence d'une copie valide de la donnée sur la puce, le *HomeNode* oriente le cœur demandeur vers la mémoire extérieure.

3.1.3 Scénarios d'accès mémoire

Nous analysons dans cette section les performances des protocoles proposés dans des cas de forte charge mémoire sur certaines zones de la puce afin de démontrer leur efficacité à gérer le stress sur la puce.

Nous étudions des traces d'accès mémoire générées de manière synthétique afin de créer des voisinage stressés (i.e. points chauds) sur la puce. Ensuite, nous varions la répartition et le nombre de ces points chauds (définition 9) ce qui permet de générer différents scénarios de stress mémoire dans la puce.

Définition 7 (cœur saturé)**.** Un cœur saturé est un cœur à cache totalement occupé.

Définition 8 (cœur stressé)**.** Un cœur stressé est un cœur fortement sollicité à cache saturé et où le taux d'accès aux données fréquemment utilisées est très élevé. Un cœur saturé n'est pas forcément un cœur stressé.

Définition 9 (Point chaud)**.** Un point chaud désigne une zone de la puce dont la mémoire est fortement sollicitée par des accès fréquents aux données (ex. un cœur stressé 8). Dans notre étude nous considérons qu'un point chaud est un voisinage constitué d'un cœur central et de ses quatre voisins directs dont les caches sont stressés.

Afin de stresser un voisinage, il faut saturer les caches des cœurs de ce voisinage et ensuite multiplier les accès aux données. Les accès répétitifs aux données sont effectués de façon alternée entre le cœur central et ses cœurs voisins.

La figure 3.3 décrit les deux étapes de saturation d'un voisinage de 5 cœurs. Soient $@a$, $@b$, $@c$, $@d$ des accès aux données associés respectivement aux cœurs voisins N (Nord), O (Ouest), S (Sud), E (Est) et permettant d'occuper toutes les lignes de cache disponibles dans leurs mémoires locales. Le cœur central est à son tour saturé par les accès au cache notés $@e$.

En alternant les accès aux données associées au cœur central et à ses cœurs voisins, tout le voisinage se trouve dans un état de stress. Ce scénario crée un point chaud dans la puce, ce qui favorise le processus de migration des données.

La figure 3.4 décrit les quatre distributions de la charge mémoire utilisées dans cette analyse sur une puce $8 * 8$. Chaque voisinage stressé est caractérisé par un nombre d'accès égale à 80000 accès mémoire. Les cœurs de couleur rouge représentent les points stressés de la puce (i.e. cœurs stressés).

Les résultats de cette analyse sont présentés et discutés dans ce qui suit selon 3 axes : Une première analyse des performances du protocole de glissement de données comparé au protocole à cache élastique (§ 2.3.1) et une deuxième analyse des performances du protocole masse-ressort (§ 3.2.4).

3.2 Analyse des performances du protocole de glissement

Le protocole de glissement de données 2.4 ainsi que le protocole de cache élastique (§ 2.3.1) proposent des techniques de coopération entre les caches d'une même puce. L'analyse du protocole élastique a démontré sa capacité limitée à gérer des situations de

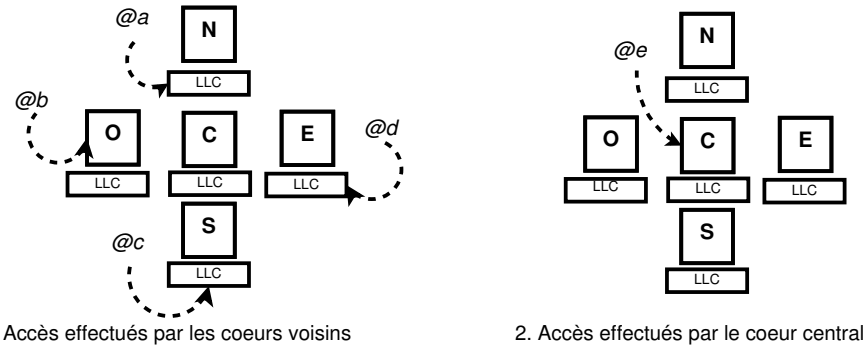
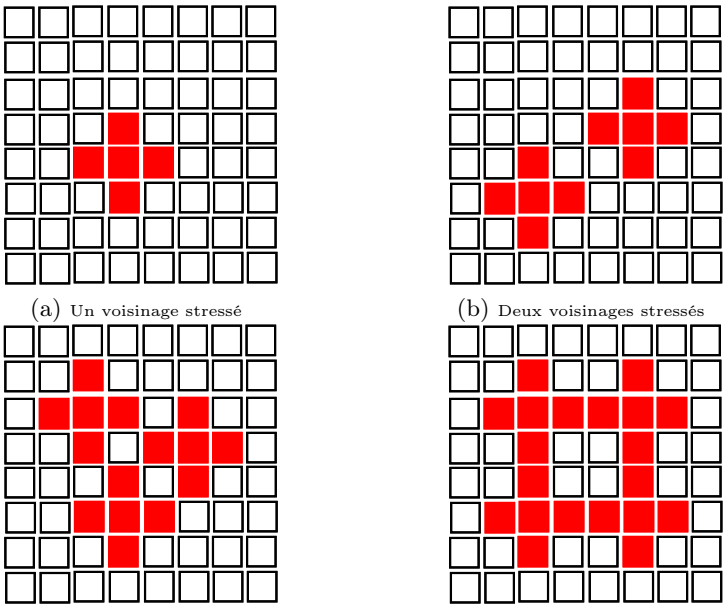


FIGURE 3.3 – Scénario 1 : Après saturation des caches des 5 cœurs du voisinage, les accès aux données sont ensuite effectués de manière alternée entre le cœur du centre et ses voisins directs. Ce scénario permet de créer un état de stress localisé (*point chaud*) où tout le voisinage est fortement stressé.



(a) Un voisinage stressé (b) Deux voisinages stressés

(c) Trois voisinages stressés (d) Quatre voisinages stressés

FIGURE 3.4 – Les charges mémoire des caches sur la puce

stress localisées dans certaines zones de la puce. Le protocole de glissement de données proposé dans ces travaux a donc pour but de réduire la charge dans ces zones stressées.

Nous analysons dans ce paragraphe l'efficacité des techniques de glissement de données comparées au protocole de cache élastique.

3.2.1 Technique de remplacement des données par priorité dans un voisinage stressé

La coopération entre les caches repose sur l'efficacité du système à gérer l'espace de stockage entre données locales et données hébergées. Le protocole de cache élastique propose un partitionnement dynamique du cache local en zone *privée* et zone *partagée*, Ce partitionnement est mis à jour de manière cyclique afin de l'adapter aux besoin des deux cotés d'utilisation du cache (utilisation local, utilisation du voisinage). L'efficacité d'une telle approche est limitée par une forte concurrence entre le cœur local et ses voisins proches.

Le protocole de glissement de données proposé utilise la technique de *remplacement par priorité*. Cette technique est basée sur une estimation des besoins mémoire en fonction du niveaux d'utilisation des données locales et de celles hébergées. Lors du remplacement d'une donnée, cette estimation permet de donner plus de poids aux données les plus sollicitées.

Afin d'évaluer l'efficacité de l'approche par priorité par rapport à celle par partitionnement, nous étudions différents niveaux de charge mémoire. Le niveau de charge est défini à partir du nombre de voisinages stressés répartis sur la puce (i.e. points chauds). L'objectif de ce scénario est d'évaluer la capacité des protocoles étudiés à gérer efficacement la distribution des données sur la puce.

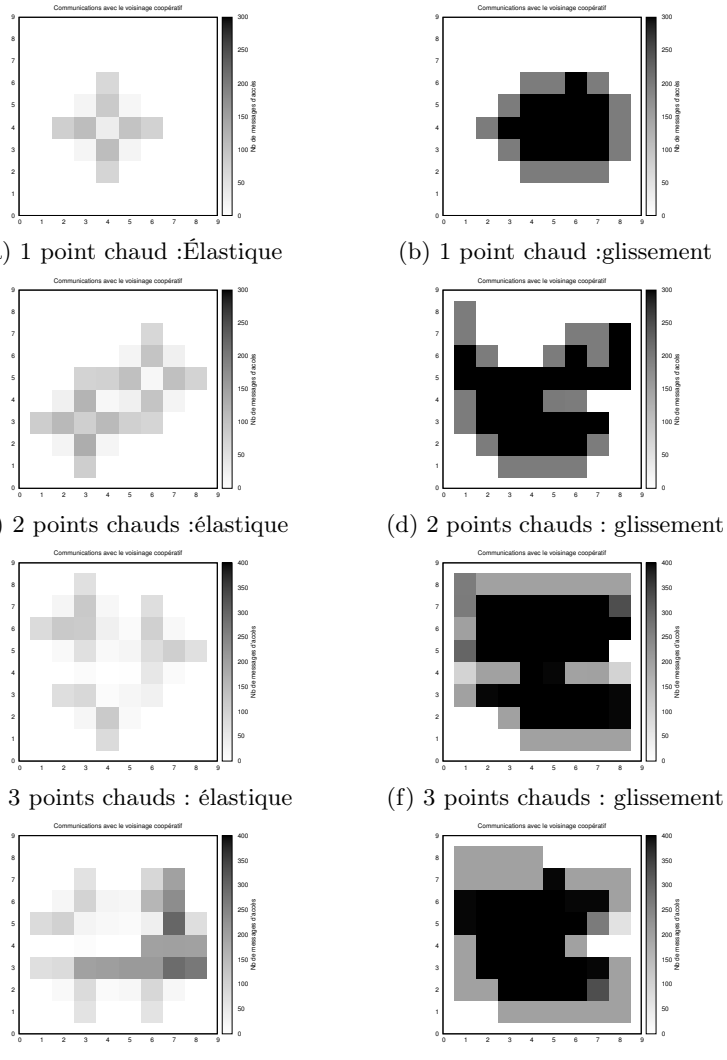
Nous comparons dans ce qui suit le protocole de glissement de données et le protocole à cache élastique selon différentes charges mémoire. Le protocole de glissement de données est configuré avec un rayon de glissement égal à 5 pour les deux premiers niveaux de stress et à 7 pour les deux derniers.

Les figures (3.4a, 3.4b, 3.4c, 3.4d) correspondent respectivement aux trafics dans les zones coopératives créées par le protocole à cache élastique et le protocole de glissement de données pour les 4 niveaux de stress étudiés. Nous considérons qu'une communication entre un cœur stressé et un autre cœur de son voisinage coopératif représente un accès réussi à une donnée stockée chez le cœur voisin. Cette analyse porte sur deux principaux points qui sont la répartition des données sur la puce (i.e surface du voisinage coopératif) et sur le nombre de succès de cache dans les zones stressées :

- Zones de coopération : Les figures ci-dessus montrent que les zones de voisinages coopératifs créées par le mécanisme de glissement de données sont plus étalées que les zones de coopération du protocole à cache élastique. Ceci est dû au fait que le protocole de glissement de données donne plus de liberté de migration aux données sur la puce, tandis que l'approche élastique limite la coopération au voisinage direct en donnant une seule chance de migration à chaque donnée.

- Nombre de succès d'accès au cache : Grâce à la migration des données vers les zones les moins stressées de la puce, le protocole de glissement de données permet d'avoir un meilleur nombre de succès d'accès dans le cas d'une forte charge mémoire (figure 3.5f, figure 3.5h).

Le protocole de glissement des données entre les voisins permet de mieux gérer la répartition de la charge mémoire des points stressés sur l'ensemble de la puce.



(a) 1 point chaud :Élastique

(b) 1 point chaud :glissement

(c) 2 points chauds :élastique

(d) 2 points chauds : glissement

(e) 3 points chauds : élastique

(f) 3 points chauds : glissement

(g) 4 points chauds : élastique

(h) 4 points chauds : glissement

FIGURE 3.5 – Carte des communications entre les cœurs du voisinage coopératif. Un message d'accès envoyé d'un cœur stressé à un cœur voisin est un accès à une donnée stockée sur un cache distant. Les zones de glissement sont plus chargées par un trafic réseau plus intense constitué des échanges de messages entre les cœurs coopératifs. Le trafic représenté dans ces figures correspond aux accès réussis aux données stockées sur la puce. Un trafic plus intense signifie donc de meilleures performances.

Les résultats obtenus démontrent que la technique de *remplacement par priorité* utilisée dans ce protocole améliore l'utilisation de l'espace de stockage des données dans les mémoires cache en l'adaptant au besoin de l'ensemble des cœurs du voisinage.

La deuxième approche du protocole de glissement de données est la technique du choix du *meilleur voisin*. Le paragraphe suivant évalue les performances de cette technique comparée à la technique de *tourniquet* (*Round Robin*) utilisée dans le protocole à cache élastique.

3.2.2 Évaluation de la technique du choix du meilleur voisin

Le scénario de test pour évaluer la technique du choix du meilleur voisin consiste à stresser un cœur central, ainsi que deux de ses voisins *Nord* et *Sud* (figure 3.6). Les données @a, @b, @c, @d sont chargés respectivement dans les caches de ces voisins. Ensuite le cœur central accède successivement aux données @e. Le stress au niveau du voisinage active un processus coopératif, qui consiste à solliciter les caches voisins.

L'objectif d'utiliser un tel scénario est d'évaluer la capacité de chaque protocole à gérer le niveau de stress des cœurs à travers leurs politiques du choix du cœur destinataire des données déplacées. Le protocole à cache élastique utilise une approche de *tourniquet* (*Round Robin*) qui envoie de façon équilibrée les données éjectées du cache local vers ses voisins directs. Le protocole de glissement de données proposé utilise la technique du choix du meilleur voisin qui se base sur une estimation locale de la charge mémoire des voisins.

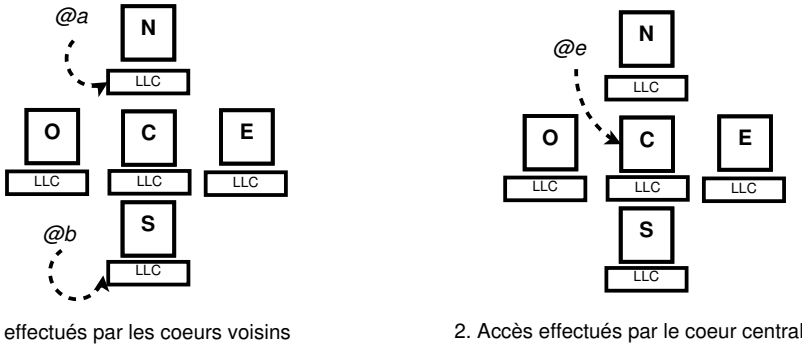
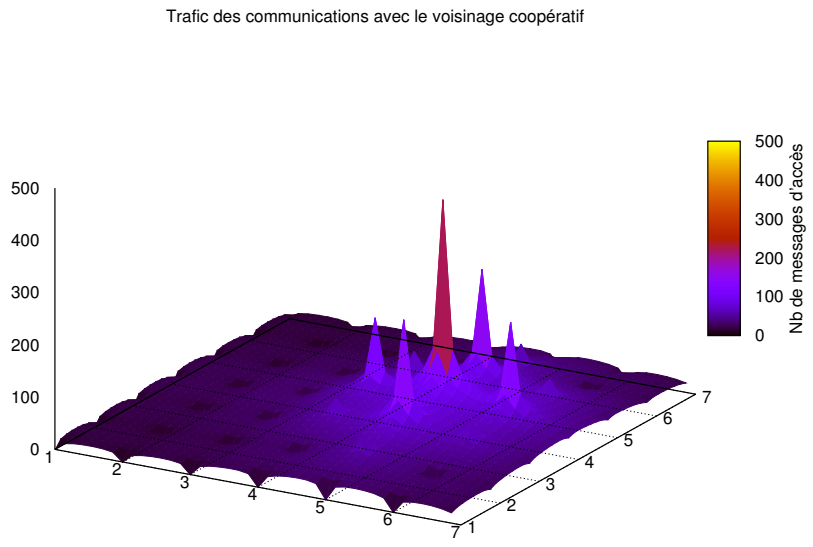


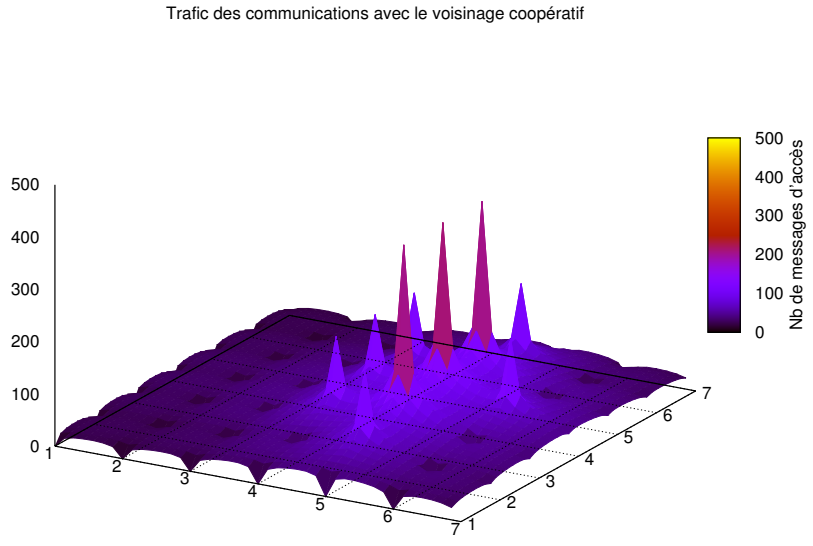
FIGURE 3.6 – Scénario 2 : Le cœur central et ses deux voisins (*Nord* et *Sud*) sont stressés. Ce scénario permet d'évaluer l'efficacité de la technique du choix du meilleur voisin dans le cas d'une répartition déséquilibrée de la charge mémoire dans le voisinage.

Les figures 3.7a et 3.7b représentent les cartes de communications au niveau des voisinage coopératifs. Le niveau de chaque pic dans les graphes représente le nombre d'accès réussis au cœurs coopératifs voisins.

Nous observons dans ces figures que pour le protocole à cache élastique le cœur central est celui qui a le plus de succès d'accès comparé aux cœurs stressés voisins (*Nord* et *Sud*). Ce phénomène est dû au fait que le cache central distribue sa charge mémoire locale de manière équilibrée sur son voisinage sans prendre en compte la charge des



(a) Trafic au voisinage avec le protocole à cache élastique



(b) Trafic au voisinage avec le protocole de glissement

FIGURE 3.7 – Cartes des communications avec le voisinage pour le protocole à cache élastique et le protocole de glissement de données. Le nombre de messages transmis par un cœur vers un voisin reflète le niveau de succès d'accès aux caches du voisinage.

cœurs destinataires (*Nord et Sud*). Par conséquent, la politique de *tourniquet* (*Round Robin*) choisie par le protocole de cache élastique répond aux besoins mémoires du cœur central tout en ignorant le niveau de stress des cœurs voisins.

La figure 3.7b décrit la répartition de la charge mémoire dans le voisinage coopératif avec le protocole de glissement de données. Nous y constatons une répartition équilibrée des données entre les 3 cœurs stressés (*Central, Nord et Sud*).

3.2.3 Analyse des performances par variation du rayon de glissement

Ce paragraphe a comme objectif d'évaluer les performances des protocoles de glissement proposés selon le rayon de glissement accordé aux données. La valeur du rayon de glissement sur une puce peut varier entre un rayon minimal égal à 1 pas réseau (i.e. voisinage direct) et un rayon maximal défini en fonction de la taille de la puce (ex. la plus grande distance entre deux cœurs sur la puce).

Dans cette analyse, nous faisons varier le rayon associé au protocole de glissement de chaque donnée entre la valeur *min*, la valeur *moyenne* et la valeur *max* pour chaque scénario d'accès et nous constatons la variation des performances mémoire obtenues pour chaque rayon.

Nous utilisons les mêmes scénarios de stress décrits précédemment (§ 3.2.1) sur une puce $8 * 8$. Les valeurs de rayon utilisées dans ce cas varient d'une valeur minimale de 1 à la valeur maximale 14 correspondant à la plus grande distance entre deux cœurs sur la puce.

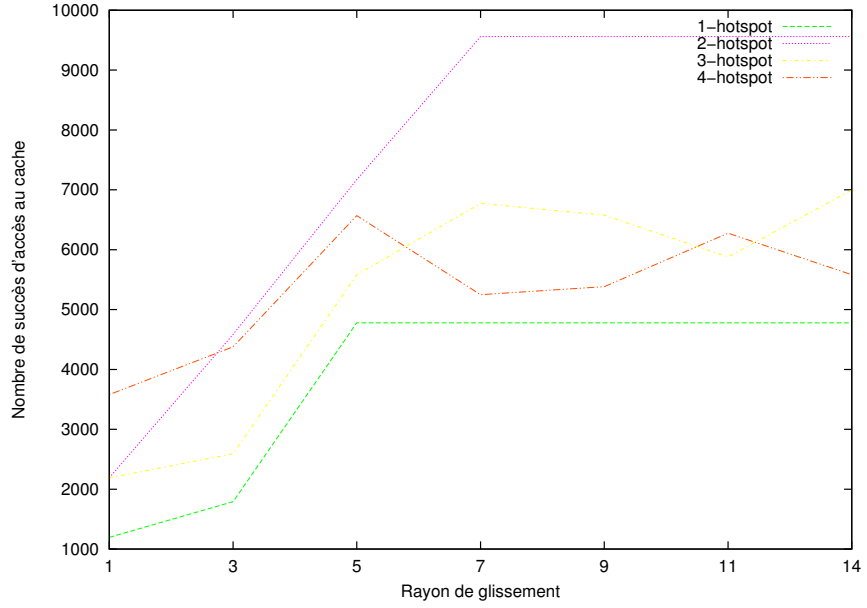


FIGURE 3.8 – Le nombre de succès d'accès au cache en fonction du rayon de glissement pour 4 scénarios de stress de la puce. Les scénarios vont du moins stressant (1 point chaud) au plus stressant (4 points chauds).

La figure 3.8 présente la variation des performances (en nombre de succès d'accès) du protocole de glissement de données en fonction du rayon de glissement associé aux données.

L'augmentation du rayon du glissement permet d'atteindre les zones les moins chargées de la puce et donc de préserver les données le plus longtemps possible sur la puce. La valeur optimale du rayon dépend de la charge des zones stressées de leur répartition sur la puce. Nous distinguons dans ce graphe (figure 3.8) deux tendances :

- Dans le cas de faible stress au niveau de la puce (1 ou 2 voisinages stressés) les performances du protocole augmentent linéairement avec le rayon de glissement et ensuite stagnent à partir d'une valeur moyenne de ce rayon (5 ou 7). Ce phénomène est dû au fait que ce rayon moyen permet d'atteindre les zones froides autours des voisinages stressés ce qui permet de baisser la charge de ces zones et d'atteindre les meilleures performances possibles.

- En augmentant le stress de la puce en augmentant le nombre de voisinages stressés, les performances ne sont plus proportionnelles à la valeur du rayon de glissement. Les zones à caches libres sont sollicitées simultanément par les différents cœurs stressés ce qui pénalise l'ensemble des caches.

Nous déduisons à partir de cette analyse que la valeur du rayon de glissement peut influencer le comportement du protocole et par conséquent améliorer ou détériorer les performances mémoire. Il est donc très important de trouver le rayon de glissement qui soit le mieux adapté au comportement de l'application. Ce point sera traité avec plus de détails dans le chapitre 6.

3.2.4 Évaluation de l'approche masse-ressort

Le protocole masse-ressort est une extension à rayon statique du protocole de glissement de données. Le rayon de glissement est défini pendant l'exécution en fonction de la charge mémoire et de la distance entre les cœurs propriétaires et leurs données distantes.

Nous analysons dans cette section les performances de ce protocole et sa capacité à s'adapter aux besoins imprévisibles de l'application et à maintenir un faible coût d'accès aux données glissées.

Notre analyse est basée sur 3 métriques : le trafic réseau (i.e le nombre de messages d'accès), le coût moyen d'accès aux données (i.e la distance de *manhattan* en nombre de pas réseau entre le cœur et sa donnée glissée) ainsi que le nombre de succès de cache.

Nous comparons le protocole de glissement de données paramétré avec deux valeurs différentes du rayon de glissement ($Rayon_{min} = 1$ et $Rayon_{max} = 14$) et le protocole de glissement masse-ressort. La première analyse comparative du protocole *masse-ressort* avec le protocole de glissement de données à rayon fixe porte sur le trafic réseau généré par chacun des protocoles étudié. La figure 3.9 montre que le glissement de données à Rayon_{min} (=1) génère un minimum de messages dans le voisinage coopératif contrairement au glissement de données à Rayon_{max}, qui crée une zone de coopération plus

large. Le protocole *masse-ressort* permet de mieux gérer la surface de la zone coopérative en limitant la propagation du glissement tout en respectant les performances de la mémoire.

La deuxième analyse porte sur le compromis entre les performances du cache et la distance cumulée d'accès réseau aux données. Le coût d'accès aux données est proportionnel à la distance cumulative par cœur.

Les barres *rouges* de la figure 3.10 présentent la distance cumulative sur l'ensemble de la trace mémoire pour chaque protocole. Le tracé *vert* de la même figure montre la variation du taux de succès d'accès au cache en fonction du choix du rayon du glissement.

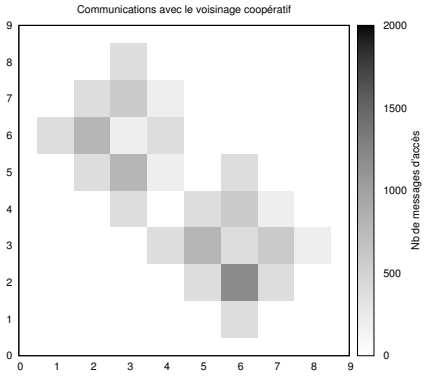
Nous constatons que le rayon de glissement maximal donne de meilleures performances (taux de succès d'accès : 97%) comparé au rayon min (taux de succès d'accès : 47%), et au rayon moyen (taux de succès d'accès : 54%) avec une plus grande distance cumulative. Ces résultats montrent que le protocole masse-ressort permet de réduire la distance cumulative globale en diminuant les performances du système (taux de succès d'accès : 68%) par rapport au rayon maximal, mais il reste plus performant que le rayon statique minimal ou moyen.

Lorsque le rayon de glissement est grand les données ont plus de liberté de migration ce qui les garde plus longtemps sur la puce, d'où le taux élevé des succès de cache avec un rayon égal à 14 pas réseau. L'inconvénient d'un grand rayon est que le coût d'accès aux données devient important (grande distance entre la donnée et son cœur propriétaire). Tandis que le rayon de glissement minimal assure une latence d'accès minimale avec une zone de migration limitée au voisinage proche, mais pénalise les performances du système en éjectant plus de données hors puce. Le glissement de données à rayon fixe nécessite un choix de performance entre l'amélioration de la disponibilité des données sur la puce et la réduction de la latence d'accès aux données (la limitation de la distance d'accès). L'approche dynamique basée sur la technique *masse-ressort* permet de trouver un bon compromis entre la distance de migration et les performances obtenues en variant la valeur du rayon en fonction de la charge de l'application et de sa répartition sur la puce. La figure 3.11 montre la variation du rayon de glissement dans l'intervalle de définition du rayon $[1, 14]$ pour 5 données différentes.

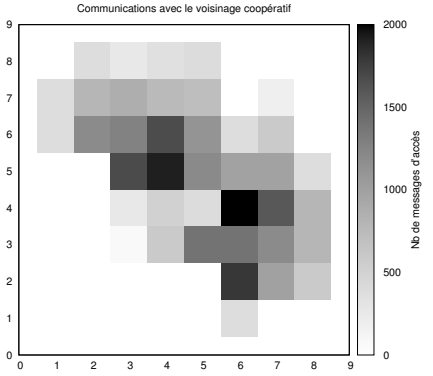
Le protocole *masse-ressort* repose sur une technique adaptative assurant une gestion dynamique du rayon de glissement de données. Les résultats obtenus démontrent que cette technique permet d'avoir un compromis entre le coût d'accès aux données et leur durée de vie sur la puce.

3.3 Discussion

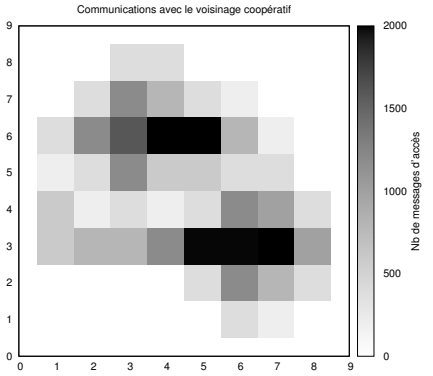
Le protocole de glissement de données à rayon dynamique a démontré de meilleures performances de caches par rapport au glissement de données à rayon statique. Malgré cela la technique de *masse-ressort* peut présenter un coût de gestion important quand l'application génère un très grand nombre de déplacements. Ce coût est dû à la nécessite de garder l'historique des deux précédents déplacements de chaque donnée. Le protocole de glissement à rayon fixe peut être intéressant si on veut limiter les déplacements des



(a) Protocole de glissement à rayon fixe min (=1)



(b) Protocole masse-ressort



(c) Protocole de glissement à rayon fixe max (=14)

FIGURE 3.9 – Zones coopératives pour les trois configurations de protocoles de glissement de données. Les niveaux de gris désignent le nombre de messages échangés entre les différents cœurs.

données dans la puce. La valeur de ce rayon change le comportement du protocole et

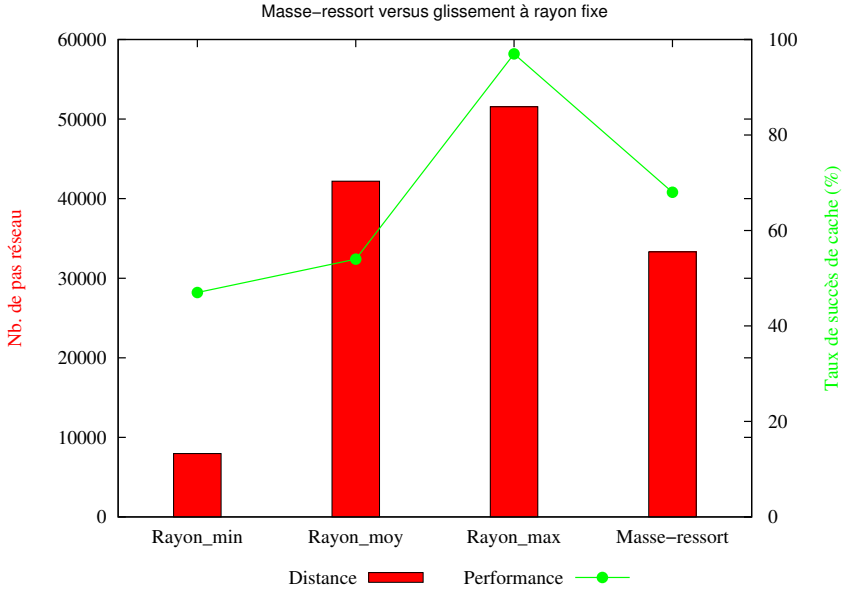


FIGURE 3.10 – La variation du taux de succès de cache et de la distance cumulative en nombre de pas réseau en fonction du rayon de glissement. Étude comparative entre les performances du glissement à deux valeurs de rayon (1,14) et le protocole de glissement adaptatif masse-ressort.

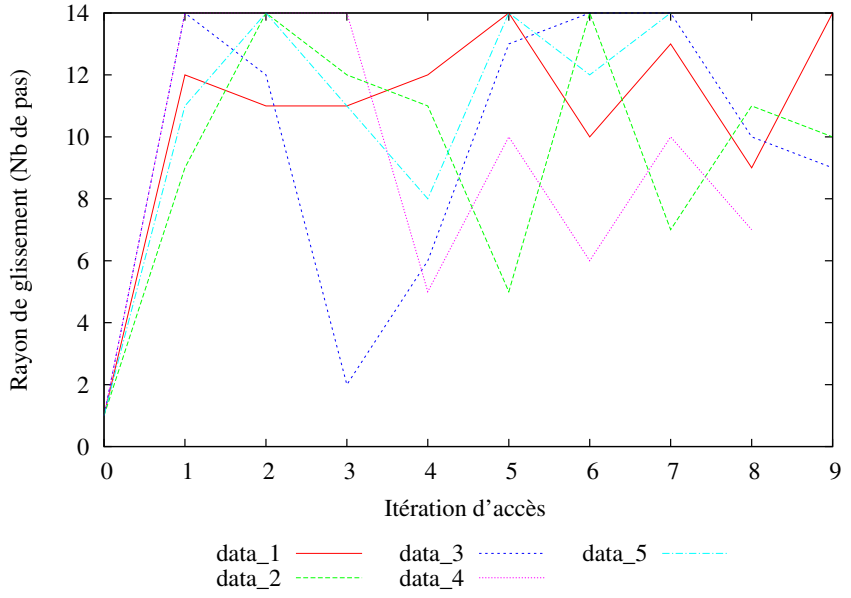


FIGURE 3.11 – Variation du rayon de glissement en ligne avec un protocole de masse ressort.

donne des performances différentes en fonction de l'application (figure 3.10).

L'étude précédente des protocoles de glissement de données a permis de constater que le comportement d'un même protocole peut être très différent en fonction du comportement de l'application et de la charge mémoire qu'elle génère. Aussi l'organisation de l'architecture cible peut impacter le comportement du protocole. Un protocole basé sur la migration des données génère plus de trafic réseau qu'un protocole *baseline*, il peut donc profiter des performances du réseau sur puce, notamment de sa topologie et des petits temps de communication entre les cœurs d'un *NoC*.

Ce constat mène vers la conclusion qu'il n'existe pas de protocole de cohérence unique qui soit adapté à tous les contextes applicatifs et à toutes les cibles d'exécution. La suite de ce travail porte sur la proposition d'une plate-forme à mémoire virtuellement partagée à protocoles multiples, permettant le choix et le paramétrage des protocoles de cohérence à partir de la phase de compilation. L'objectif d'une telle plate-forme est d'adapter le choix du protocole de cohérence, à la fois, au comportement de l'application caractérisé grâce à l'analyse statique et à l'architecture cible définie par l'utilisateur. L'utilisateur a également le choix d'un profil de performance (ex. basse consommation d'énergie, hautes performances) pour orienter les choix de protocoles et leur configuration. Un autre avantage d'adaptabilité est que la plate-forme permet une granularité fine d'association des protocoles de cohérence. Il est possible d'associer une instance de protocole pour chaque accès mémoire. Ceci facilite l'adaptation en ligne, à moindre coût (puisque toutes les décisions sont prises hors ligne), des protocoles de cohérence à l'état courant de la puce.

La plate-forme proposée est composée de plusieurs briques. Les chapitres suivants décrivent ces différentes briques. Chaque étape de description est suivi d'une étape de validation présentant les différents résultats d'évaluation obtenus.

Chapitre 4

Plate-forme de compilation multi-protocolaire pour les manycœurs

Nous présentons dans ce chapitre la plate-forme de compilation multi-protocolaire permettant de faire choix de protocoles de cohérence à la compilation. Le chapitre présente les différentes briques développés dans ces travaux de thèse allant de la caractérisation de l'application au choix et paramétrage des protocoles.

Il est organisé comme suit : La section 4.1 porte sur l'état de l'art des mémoires virtuellement partagées, la section 4.2 présente une vue globale sur l'architecture de la plate-forme de compilation proposée, la section 4.3 présente les différentes méthodes et techniques d'analyse statique existantes dans l'état de l'art, la section 4.4 propose un modèle d'analyse statique qui permet de caractériser les accès partagés à la mémoire pour l'orientation des choix de protocoles de cohérence, la section 2.6 pour conclure le chapitre avec une discussion sur les solutions d'analyse statique et d'aide á la décision proposées.

4.1 Les systèmes à mémoire virtuellement partagée

Cette section présente l'état de l'art des systèmes à mémoire virtuellement partagée. Nous nous intéressons particulièrement à la problématique de la gestion de la cohérence des données dans ce genre de systèmes et des problématiques sous-jacentes. La sous-section 4.1.1 décrit les principaux axes de décision pour la conception d'un système à mémoire virtuellement partagée. La section 4.1.2 discute des différents détails d'implémentations de tels systèmes.

4.1.1 Défis de conception des systèmes à mémoire virtuellement partagée

La conception d'une mémoire virtuellement partagée (*MVP*) dépend des choix effectués au niveau de l'organisation et de la granularité des données partagées, du modèle de cohérence, de la taille de l'infrastructure visée et de son hétérogénéité.

Organisation et granularité des données : les données peuvent être organisées dans la mémoire de façon linéaire simple (organisation en tableau) comme elles peuvent être structurées en objets selon les types de données définis par le langage de programmation ou la forme d'associativité. La granularité signifie l'unité (le plus petit élément) de partage de données : par exemple un octet, une page ou une structure de données plus complexe. Lorsque la granularité est grande, le surcoût de la structuration de la mémoire est faible mais le partage des données peut engendrer des contentions. Ces contentions est dûe principalement au problème de faux partages 10. Le choix de la taille de la granularité affecte également la taille des répertoires utilisés pour le stockage des informations sur les données.

Définition 10 (Problème de faux partages)**.** Lorsqu'un cœur modifie une partie d'une ligne de cache, et un autre cœur écrit dans une autre partie de cette même ligne. Du point de vue du système la ligne de cache a subit deux modifications différentes. Le problème de faux partage se présente donc par l'invalidation de toutes les autres copies de la ligne même chacun des cœurs n'a modifié que sa partie de la ligne.

La gestion de la granularité peut se baser sur l'associativité de la mémoire. Il s'agit d'un organisation similaire aux bases de données. Une telle organisation nécessite l'utilisation de fonctions spécifiques pour l'accès aux pages mémoire ce qui permet une gestion transparente des accès entre les différents fils d'exécution.

Un autre type de granularité selon la taille des types de données définie par le langage de développement présente l'avantage de s'adapter dynamiquement à l'application.

Cohérence des données : Le modèle de cohérence (traité dans le chapitre 1) définit comment le système gère les accès parallèles à la mémoire. C'est un élément principal dans la conception d'une mémoire virtuellement partagée (*MVP*). Les exemples de *MVPs* (Table 4.1) sont basés sur des choix différents de modèles de cohérence : *Ivy* (cohérence forte), *Munin* (cohérence faible). La majorité des *MVPs* offrent des mécanismes permettant de forcer la cohérence, mais ceci doit être exprimé explicitement par le programmeur ou le compilateur. La cohérence relâchée permet généralement une gestion plus efficace du partage des données car elle nécessite moins de synchronisations et de déplacements de données que la cohérence stricte qui peut dégrader les performances si le système d'exécution ne lui est pas adapté.

Passage à l'échelle : Le concept des *MVPs* permet de donner l'impression d'un espace de stockage global et unique ce qui facilite la gestion des accès à de nombreuses

mémoires réparties, et promet par conséquent un meilleur passage à l'échelle par rapport à une vision répartie de la mémoire.

Cet avantage est tout de même limité par deux facteurs dûs à la gestion centralisée du partage des données : goulot d'étranglement (généré par de multiples accès à une même zone mémoire partagée), propagation des informations nécessaires à une connaissance globale de l'état de la mémoire (diffusion générale de messages, saturation des répertoires). Afin de réduire les effets négatifs de la gestion centralisée, les travaux de *Li et Hudak* ont proposé un algorithme dynamique réparti pour leur système *Ivy* [LH89].

Le réseau de communication sur lequel une *MVP* est implémentée peut aussi limiter sa capacité à passer à l'échelle. Le système *Ivy* par exemple ne peut dépasser 100 nœuds sur un réseau *Ethernet* de 10 $Mbit/s$ (goulot d'étranglement).

Les implémentations *Shiva* [KCDZ94a] et *Koan* [LP92] ont été proposées afin de supporter des architectures *hypercubes* (ex. Intel iPSC/2) ce qui a permis d'augmenter le nombre de nœuds jusqu'à 128. Les nœuds dans un système à *MVP* de type *Dash* sont connectés sous forme de 2 mailles ce qui permet théoriquement une meilleure extensibilité. Cette dernière est néanmoins limitée par le fait que le protocole *Dash* utilise des vecteurs de bits complets pour garder la trace des données dupliquées (1 bit par nœud).

Hétérogénéité : La complexité d'implémenter une *MVP* sur une plate-forme hétérogène relève en partie de la différence entre les représentations des données dans les différentes mémoires réparties (entiers, virgule flottante, etc ...). Dans une *MVP* pour plate-forme hétérogènes, les données peuvent être vues comme des objets du langage de programmation d'origine, qui seront ensuite converties par le compilateur aux formats adéquats. Deux exemples de *MVP* qui gèrent l'hétérogénéité : *Agora* [FBC87] et *Mermaid* [TBD+87]. *Agora* [FBC87] définit les variables comme des *contextes*. Elle fournit une bibliothèque de fonctions pour créer, détruire, lire, écrire des *contextes* vus comme des extensions des langages supportés par le système. Tandis que *Mermaid* procède différemment pour la gestion de l'hétérogénéité. Elle se base sur une répartition de la mémoire en pages. Chaque page ne peut contenir qu'un seul type de données. Si une page est déplacée vers une nouvelle architecture, la donnée contenue dans la page est convertie vers le format approprié. Le surcoût dû aux opérations de conversion peut nuire au performance d'exécution et limiter le passage à l'échelle.

Les choix précédents sont décisifs dans la conception d'une *MVP*. Ils ont un impact direct sur les performances du système. L'implémentation de la *MVP*, basée sur ces choix de conception, peut également varier en fonction des choix de mécanismes et protocoles liés à la gestion de la mémoire. La section suivante présente les différentes implémentations de *MVP*.

4.1.2 Techniques d'implémentation d'une mémoire virtuellement partagée

Les accès aux données partagées nécessitent des algorithmes de localisation, d'accès, des mécanismes de gestion de la cohérence et de remplacement des données.

Techniques de localisation

Les techniques de localisation permettent au système de repérer les emplacements des données pour répondre à chaque accès. Si les données sont réparties statiquement sur les caches, chaque donnée a un emplacement unique ce qui facilite sa localisation. Il est facile dans ce cas d'utiliser une table de hachage pour distribuer les données sur les caches. La répartition statique permet une localisation simple et rapide mais peut causer des goulots d'étranglement si toutes les données sont affectées au même cache. Une autre alternative à l'approche statique est d'autoriser que les données changent d'emplacement mémoire librement en fonction de leur utilisation. La localisation des données dans ce cas nécessite une connaissance globale centralisée des traces de mouvements des données. L'approche dynamique représente tous les inconvénients d'une gestion centralisée à savoir la surcharge du système, la réduction du parallélisme (les requêtes d'accès sont sérialisées au niveau du répertoire central).

Des systèmes plus récents autorisant la migration et la réplication des données utilisent la technique basée sur des répertoires répartis où chaque donnée est affectée à un nœud propriétaire, nœud avec la première copie de la donnée. Lorsqu'un nœud veut accéder à une donnée, il envoie une requête au propriétaire. Si ce dernier n'a plus de copie de la donnée il transfère la requête vers le nœud qui récupère la donnée de son cache. L'inconvénient de cette technique est le risque de multiples redirections de la requête ce qui revient au coût d'une diffusion générale.

Techniques de la gestion de la cohérence

Les plate-formes à mémoire virtuellement partagée doivent offrir une forme de gestion de la cohérence des données. Une manière intuitive de gérer les accès à la *MVP* est d'interdire la réplication des données dans les différents caches. Il est plus naturel de choisir un modèle de cohérence stricte où chaque modification d'une donnée est tout de suite rapportée à sa copie principale dans la mémoire physique. Les accès aux données partagées sont gérés de manière séquentielle, ce qui pénalise les performances de l'exécution parallèle. Afin de surmonter cette limitation, certaines *MVPs* autorisent plusieurs copies d'une même donnée. Cette solution permet des accès parallèles (ex. plusieurs lectures) aux multiples copies d'une donnée. Les principales approches de gestion des mises à jour sont traitées précédemment dans la section 2.1.1. Un modèle basé sur l'invalidation des copies est plus efficace que le modèle basé sur la diffusion de mises à jour pour certaines applications comme la résolution d'équations linéaires [LH89]. D'autant plus que la diffusion des mises à jour peut être très coûteuse à cause de la latence du réseau de communication.

Le choix d'un protocole de cohérence est fortement dépendant du comportement de l'application. Afin de réduire le surcoût lié à la gestion de la cohérence certaines *MVPs* offrent une forme relâchée de la cohérence des données, comme dans le cas du système *DASH* [LLG+90a]. La cohérence à la libération est une forme de cohérence relâché, qui consiste à réduire le nombre de communications réseau dû à la gestion de la cohérence. Les mises à jour des donnée ne sont effectuées qu'à l'entrée et la sortie des sections critiques. Il existe deux formes de cohérence à la libération :

- Cohérence à la libération immédiate : les mises à jours sont diffusées juste après la libération du verrou protégeant la section critique. *Munin* [BCZ90] est un exemple de ce type de cohérence.

- Cohérence à la libération paresseuse : les données ne sont mises à jour qu'à la prochaine entrée dans une section critique. Treadmarks [KCDZ94b] utilise ce type de cohérence.

Cette forme de faible cohérence a l'avantage de réduire le nombre de synchronisations et de déplacements des données. Dans plusieurs travaux le partage des données est effectué selon un modèle de cohérence fixé : cohérence à la libération, séquentielle, etc. C'est au programmeur d'adapter son application à la *MVP* et à la plate-forme d'exécution ce qui pose de nombreuse limitations de performance.

Les travaux de thèse de *Gabriel Antoniu* [Ant01] proposent la plate-forme DSM-PM2 permettant de mettre en place différents protocoles de cohérence dans un système à *MVP*. Elle supporte plusieurs types de cohérences telles que la cohérence Java, la cohérence à la libération et la cohérence séquentielle. La plate-forme *JuxMEM* proposée dans les travaux de thèse de *Mathieu Jan* propose un service de partage de données hybride inspiré à la fois par les systèmes à *MVP* et les systèmes *P2P*. Ce service offre à l'utilisateur de choisir un modèle et un protocole de cohérence et c'est le service qui fournit les primitives nécessaires à la gestion des accès concurrents.

Techniques de remplacement

Un autre facteur de performance des *MVPs* est la stratégie de remplacement utilisée. Lorsque un nœud du système veut accueillir une nouvelle donnée, le choix de la donnée à remplacer ainsi que sa destination est d'une grande importance. Les techniques de remplacement utilisées dans les *MVPs* sont inspirées des techniques utilisées dans les systèmes multiprocesseurs à mémoire partagée comme l'algorithme *LRU* (en anglais *Least Recently Used*) qui consiste à remplacer la donnée la moins récemment utilisée. Les systèmes *MVPs* gèrent les remplacements des données en fonction de leur priorité en privilégiant celles partagées par exemple, une technique utilisée dans le système *Shiva* [LS89]. Une donnée remplacée ne doit pas être perdue. Une première façon sûre est de l'envoyer vers la mémoire principale. À cause du coût de transfert à l'extérieur de la puce, une autre manière de garder les données sur la puce est de réserver dans le cache local de chaque nœud un espace dédié aux données non utilisées. En dépit de son bas coût d'implémentation, cette technique mène à une perte mémoire importante sur la puce ce qui limite les performances globales du système. Une amélioration possible, proposée par *Shiva*, est d'envoyer les données déplacées vers les nœuds avec un espace mémoire libre suffisant.

Les systèmes à *MVP* sont particulièrement sujets au phénomène d'emballement (en anglais *Thrashing*). Il s'agit d'un état de la mémoire où deux processeurs se partagent l'accès à une donnée à une très grande fréquence ce qui engendre un effet *ping pong* entre les caches des processeurs. Différents travaux se sont intéressés à résoudre ce problèmes. Le système à *MVP* appelé *Munin* propose de nouveaux types de données tels que :

mode *producteur-consommateur*, mode *read-mostly* (lecture dans la plupart du temps), mode privé (accès privé), mode write-many (plusieurs écritures) ou mode *write-once* (écriture unique). Le programmeur a la possibilité de choisir un mode de partage lui permettant d'affecter à chaque donnée partagée un mode d'accès ce qui empêche un état d'emballement. L'utilisateur n'est toujours pas en mesure de prédire parfaitement le comportement de son programme ce qui affaiblie l'efficacité de cette méthode. La plate-forme *Mirage* [FP89] propose une autre méthode de gestion de l'emballement. Elle consiste à examiner les situations particulières où l'effet *ping pong* se présente et l'arrête grâce à l'aspect dynamique du protocole de cohérence utilisé dans ce système. Le protocole de cohérence prend ses décisions en se basant sur un paramètre variable. Ce dernier calcule le temps minimal de disponibilité d'une donnée dans le cache d'un processeur. Le calcul de ce temps est basé sur les schémas d'accès aux données afin d'adapter automatiquement le protocole de cohérence au comportement de l'application. Les travaux de Shen et al. [SR+99] proposent un nouveau protocole de cohérence appelé *CACHET* afin de résoudre le problème d'emballement dans les *MVPs*. *CACHET* regroupe un ensemble de trois sous-protocoles chacun adapté à un motif particulier d'accès à la mémoire. Chaque sous-protocole est composé de plusieurs règles qui ne sont pas déclenchées forcément par une instruction particulière ou par message de protocole. Il est, à tout moment, possible d'initier une action du coté du cache (ex : envoyer une requête d'accès à la mémoire) ou bien du coté de la mémoire principale (ex : envoyer une requête de rechargement à partir d'un cache). Cette méthode se montre efficace en terme de limitation de l'effet *ping pong*, notamment parce qu'elle autorise plusieurs accès simultanés en écriture. Cependant, elle ne supporte que des modèles de cohérence faible. Elle nécessite également plusieurs techniques de reconnaissance de motifs d'accès ce qui peut exiger l'implication du programmeur pour la définition de ces motifs en annotant le code par exemple.

Techniques de synchronisation

La synchronisation est un élément crucial dans l'implémentation d'une *MVP*. Les travaux de *Debois et al.* [DSB+88] étudient les techniques de synchronisation et sa relation avec la cohérence des données. Plusieurs techniques de synchronisation sont utilisées dans les systèmes multiprocesseurs. Des primitives de synchronisation sont implémentées sur différents niveaux matériels ou logiciels afin de garantir l'exclusion mutuelle (définition 11).

Définition 11 (Exclusion mutuelle)**.** Une technique de gestion des accès concurrents aux données garantissant qu'une ressource mémoire ne soit utilisée que par une seule tâche à la fois. Elle est définie comme une primitive de synchronisation qui peut être implémentée de différentes manières (gestion centralisée des requêtes d'accès, gestion par jeton, gestion par permission).

Une façon simple de gérer la synchronisation au niveau matériel est de forcer des accès atomiques aux données. Cette technique associe à chaque processus une variable de contrôle lui permettant de prendre la main sur sa section critique en interdisant qu'un

processus concurrent s'exécute simultanément. L'inconvénient d'une telle approche est qu'elle peut générer un interblocage (en anglais *Deadlock*) entre les deux processus. une autre implémentation matérielle des primitives de synchronisation est l'utilisation des interruptions entre processus. Un processus peut envoyer une interruption à un autre processus s'exécutant sur un processeur différent. Les verrous sont également utilisés pour mettre en attente (boucle vide) un processus en attendant que le verrou de synchronisation se libère. La mise en attente des processus dégrade les performances du système (certains processeurs se retrouvent inexploités avec des processus en attente). Un autre type de verrous basé sur les interruptions consiste à signaler la libération d'un verrou à tous les processeurs à travers une interruption. Si un processeur ne réussit pas à récupérer le verrou, il mémorise sa situation courante et coupe son attente. L'intérêt de ce type de verrou est de réduire le temps d'attente excessif.

Au niveau des implémentations logicielles des primitives d'exclusion mutuelle, il existe deux principales techniques : Les sémaphores et le passage de message. Pour les sémaphores, deux opérations sont définies (P et V). Une sémaphore est une variable qui peut être binaire (0 pour acquisition et 1 pour blocage), ou une valeur entière supérieure à 0. Les opérations P et V peuvent faire partie du noyau système permettant de manipuler des verrous. Une autre implémentation logicielle de la synchronisation est basée sur les barrières. Une barrière est un point de synchronisation où toute tâche qui y arrive est mise en attente jusqu'à ce qu'un nombre spécifique de tâches soient arrivées à ce même point. C'est une approche qui a montré son efficacité pour l'exécution parallèle.

4.1.3 Mémoires virtuellement partagées : l'existant

Les différents choix de conception des *MVPs* donnent naissance à plusieurs implémentations de systèmes à mémoire virtuellement partagée. Ces implémentations présentent un surcoût bas au niveau du moteur d'exécution. Les trois principaux modes d'implémentations qui existent dans la littérature sont :

- Implémentation matérielle : extension des mécanismes de gestion des données existants sur support matériel.
- Implémentation dans le système d'exploitation : utilisation des mécanismes de gestion de mémoire virtuelle.
- Implémentation dans le compilateur : les accès aux données partagées sont automatiquement remplacés par des appels à des routines de gestion de la cohérence.

Le niveau d'implémentation des *MVPs* est un point de décision très important car il affecte directement les performances et le coût du système. Le tableau 4.1 présente les différents systèmes à mémoire virtuellement partagée et leur implémentation ainsi que les modèles de cohérence correspondants.

Les *MVPs* sont implémentées sur plusieurs types d'architectures réparties et différentes échelles et topologies. Les systèmes présentés dans cette section représentent chacun des choix de conception distingués au niveau de la cohérence, de la synchronisation, de la structuration et de la granularité des données partagées. Des implémentations

Nom du système	Référence	Implémentation	Modèle de cohérence	Synchronisation
Agora	Bisianiand Forin, CarnegieMellon University [RB88]	Implémentation logicielle	Cohérence faible	Moniteur
Amber	Chase, Feeley, and Levy, University of Washington [Cha89]	Implémentation logicielle	Cohérence stricte	Verrous
Dash	Lenoski, Laudon, Gharachorloo,Gupta, and Hennessy, Stanford University [LLG+90b]	Implémentation matérielle	Cohérence relâchée	Verrous
Ivy	Li, Yale University [HGC02]	Implémentation logicielle	Cohérence stricte	Sémaphores, comptage d'événements
Memnet	Delp and Farber, University of Delaware [DSF88]	Implémentation matérielle	Cohérence stricte	Synchronisation d'horloges
Mermaid	Carrier and Gelernter,Yale University [ZSLW98]	Implémentation logicielle	Cohérence à la libération stricte	Messages pour sémaphores et signaux de mise en attente
Treadmarks	Pete Keleher, Alan L. Cox, Sandhya Dwarkadas and Willy Zwaenepoel, Rice University [KCDZ94b]	Implémentation logicielle	Cohérence à la libération relâchée	Messages pour sémaphores et signaux de mise en attente
Mirage	Fleisch and Popek, University of California [FP89]	Implémentation logicielle	Cohérence stricte	Sémaphores
Munin	Bennett, Carter, and Zwaenepoel, Rice University [BCZ90]	Implémentation logicielle	Cohérence faible	Objets synchronisés
Plus	Bisiani and Ravishankar, Carnegie Mellon University [BR90]	Implémentation matérielle et logicielle	Cohérence par processeur	Instructions complexes de synchronisation
Shiva	Li and Schaefer, Princeton University [LS89]	Implémentation logicielle	Cohérence stricte	Messages pour sémaphores et signaux de mise en attente
DSM-PM2	Gabriel Antoniu et Luc Bougé ENS Lyon, 2001 [Ant01]	Implémentation logicielle	Cohérence séquentielle cohérence à la libération et cohérence Java	Verrous
JuxMEM	Gabriel Antoniu and Luc Bougé et Mathieu Jan Inria Rennes, 2005 [ABJ05]	Implémentation logicielle	Cohérence configurable (ex. Cohérence à l'entrée)	Verrous

TABLE 4.1 – Tableau comparatif des différents systèmes à mémoire partagée et leurs modèles de cohérence respectifs.

matérielles par exemple supportent une petite granularité de partage de données tandis que les implémentations logicielles ou hybrides offrent plus de flexibilité au niveau de la taille et de l'organisation des données manipulées.

Les performances d'un système peuvent varier selon le comportement de l'application et l'efficacité de l'architecture à gérer les communications parallèles entre processeurs et à manipuler la migration des données entre les mémoires. Des systèmes à protocoles adaptatifs ont été proposés. Par exemple, *Mirage* grâce au paramétrage de son protocole de cohérence réussit à gérer dynamiquement le problème de l'emballement, mais il ne prend en charge qu'un seul protocole de cohérence.

4.1.4 Discussion

Les systémes á mémoire virtuellement partagée offrent á l'utilisateur la possibilité de manipuler l'ensemble des mémoire réparties sur la puce comme un espace de stockage global sans se soucier de la gestion des accés partagés. L'efficacité d'un modéle repose sur le modéle de cohérence associé et de sa capacité á gérer de maniére transparente les accés paralléles aux données. L'étude des mécanismes et protocoles de cohérence constituant un modéle de cohérence a permis de montrer qu'il n'existe pas un protocole de cohérence qui soit adapté á tous les contextes applicatifs et á toutes les architectures cibles.

Le choix d'un protocole de cohérence efficace est un des facteurs clé de performance d'un système. Les systèmes existants proposent deux principales techniques. Une première technique déléguant au programmeur la décision sur le protocole de cohérence en fonction de ses besoins applicatifs. Les choix de l'utilisateur sont souvent limités par les contraintes de l'architecture. De plus, lors du développement de son programme, il ne possède pas toujours les moyens de vérifier l'efficacité de ses choix à l'exécution (difficulté d'accès à l'architecture). La deuxième technique consiste à implémenter un ou plusieurs protocoles de cohérence dans le système et d'adapter en ligne le choix des protocoles en fonction de l'évolution de l'exécution. Cette technique représente un surcoût élevé ce qui peut dégrader les performances du système. Nous nous intéressons dans le cadre de cette thèse à la conception d'une plate-forme de compilation multi-protocolaire qui, pendant la compilation, permet de choisir et de configurer une combinaison de protocoles de cohérence en prenant en compte l'application et son environnement d'exécution.

4.2 Architecture de la plate-forme de compilation

Afin de répondre efficacement à la question du choix des protocoles de cohérence, nous proposons une chaîne de compilation multi-protocolaire qui a comme objectif d'associer un ensemble de protocoles de cohérence à une application en prenant en compte sa charge mémoire aussi bien que l'architecture cible. Les sections suivantes présentent avec plus de détails sur le rôle des différentes briques de la plate-forme de compilation.

Le processus de décision proposé dans le cadre de ce travail consiste à choisir et instancier une combinaison de protocoles adaptés aux besoin en performances en fonction du comportement de l'application ainsi que les spécificités de la cible d'exécution.

Ce processus de décision fait partie d'une chaîne de compilation orientée gestion de la mémoire. La chaîne de compilation comporte plusieurs briques allant de la phase de caractérisation de l'application à la phase d'instanciation des protocoles en passant par une phase d'évaluation des performances.

La figure 4.1 présente une vue d'ensemble des principales étapes et briques qui constituent la plate-forme de compilation proposée : caractérisation de l'application, décision et le choix des protocoles, instanciation et paramétrage des protocoles, évaluation et validation des choix effectués.

La phase de **caractérisation de l'application** repose sur des techniques d'analyse statique de code permettant d'identifier les tâches parallèles, les accès mémoires concurrents aux variables partagées.

La phase de **décision et du choix des protocoles** consiste à associer un protocole de cohérence à chaque accès à une variable partagée. Elle vient après l'étape d'analyse statique de code et se base sur un arbre de décision prédéfini.

La phase de **instanciation et de paramétrage des protocoles** est l'étape qui suit la décision sur les choix des protocoles. À l'issue de cette phase chacun des protocoles choisi est configuré avec les paramètres adaptés au contexte de performance choisi. Elle utilise un processus d'optimisation génétique afin de réduire le surcoût lié à la recherche des paramètres. Au final, la phase d'**évaluation et validation de choix** permet de vérifier à moindre coût les performances obtenues avec la combinaison d'instances de protocoles définis.

Le modèle de compilation présenté permet une granularité fine d'association des protocoles de cohérence (i.e. un protocole par accès). L'affectation d'un protocole par accès partagée peut avoir un coût élevé aussi bien au niveau de la compilation qu'au niveau de l'exécution. Lors de la phase de compilation , la complexité de la configuration et du paramétrage des protocoles choisis augmente exponentiellement en fonction du nombre de protocoles à configurer, ce qui relève directement de la granularité considérée dans notre analyse. De plus, une granularité fine peut engendrer une grande fréquence de changement de protocoles en ligne ce qui dégradera les performances du système.

Afin de limiter ce surcoût, il est possible définir différents niveaux de granularité selon le besoin de l'utilisateur et la précision souhaitée. Les détails de l'ensemble des briques sont présentés dans les sections suivantes.

4.3 Méthodes d'analyse statique du code

Il est généralement nécessaire que l'application soit adapté à la plate-forme d'exécution pour mieux profiter des performance du système. De plus, le programmeur doit faire les bons choix de gestion des accès concurrents aux données, souvent dirigés par le modèle de cohérence de la *MVP*. La section précédente présente une vue globale de l'architecture de la plate-forme de compilation pour le choix et le paramétrage des protocoles de cohérence. L'objectif d'une telle plate-forme est d'aider l'utilisateur dans la gestion des accès concurrents aux données partagées de manière transparente. Le processus de décision présenté prend en considération à la fois la plate-forme d'exécution

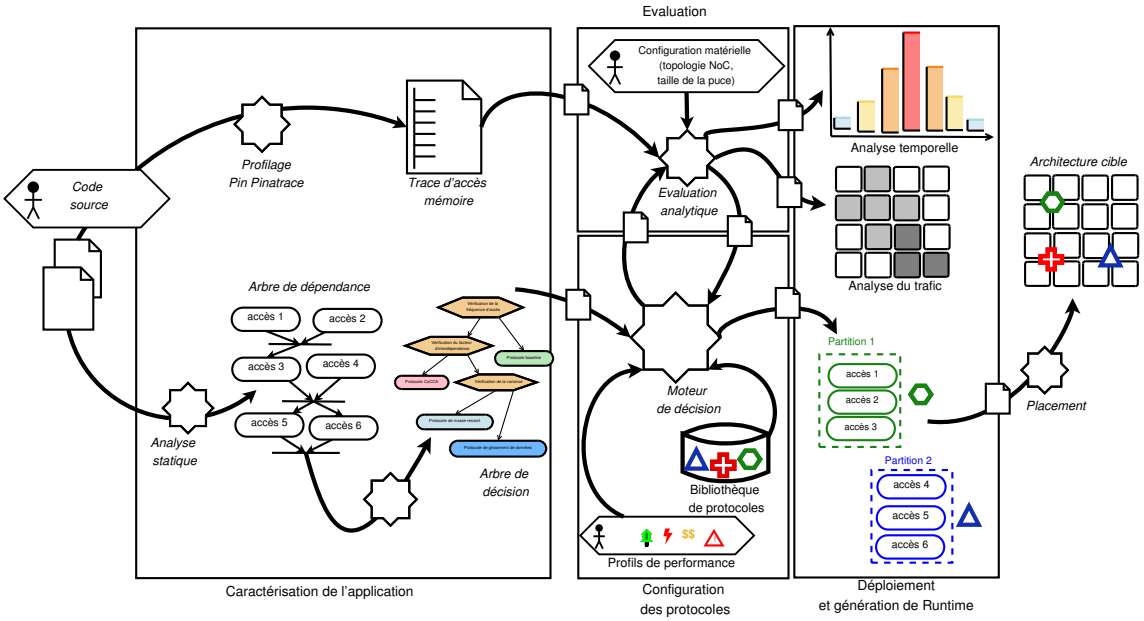


FIGURE 4.1 – La plate-forme de compilation pour le choix et le paramétrage des protocoles de cohérence. Le processus de compilation décrit prend en compte le comportement de l'application et les caractéristiques de la cible d'exécution ainsi que le profile de performance choisi par l'utilisateur.

et le comportement de l'application. L'analyse du comportement de l'application pour la gestion de la cohérence consiste à caractériser les accès concurrents aux variables partagées à travers l'analyse des graphes de dépendances entre les accès aux données.

Nous présentons dans cette section la brique de caractérisation du comportement de l'application basée sur l'analyse statique de code.

4.3.1 Techniques d'analyse statique

L'analyse statique consiste à extraire des informations sur le déroulement d'un programme sans l'exécuter.

L'analyse statique des programmes peut s'inscrire dans un processus de vérification afin de détecter certains défauts liés aux accès concurrents tels que la famine, l'interblocage, les situations de compétition d'accès aux données, etc. Ce type d'analyse est aussi utilisé pour l'optimisation des programmes parallèles afin d'améliorer les performances durant l'exécution (par exemple, l'optimisation de boucles imbriquées). Des outils d'analyse *syntaxique* qui portent sur la syntaxe du code sont également rangés dans la catégorie d'analyse statique. La vérification fiable d'un programme nécessite de procéder à une analyse qui considère l'ensemble de ses exécutions possibles, ou ce qu'on appelle *une analyse sémantique*. Il s'agit d'un problème complexe et indécidable surtout dans le cas des programmes parallèles où il devient très difficile de se prononcer sur l'ordre d'exécution des processus concurrents. Des méthodes exhaustives comme le

test ou le raisonnement déductif atteignent leurs limites pour les systèmes complexes.

Les travaux sur cet axe ont conduit vers deux grandes familles de méthodes formelles qui sont **l'interprétation abstraite** [CC77] et **la vérification de modèles** [CGL94] *(en anglais model checking).*

Nous définissons quelques notions utilisées dans la suite de cette section :

Définition 12 (Trace d'exécution)**.** La trace d'exécution est une séquence des états consécutifs d'exécution possible d'un programme. Plusieurs traces d'exécution peuvent dériver d'un même programme.

Définition 13 (État d'exécution)**.** À chaque pas du programme correspond des valeurs instantanées de la mémoire et de l'environnement (ex. les horloges). Un état est donc constitué de l'instant où est rendue l'exécution (c-à-d la valeur du compteur de programme) et des valeurs instantanées des variables et de l'environnement sous-jacent.

Définition 14 (Sémantique)**.** La sémantique d'un programme est le modèle de calcul qui décrit les exécutions effectives possibles de ce programme. La sémantique d'un langage est donnée par tout programme syntaxiquement correct de ce langage [Cou00].

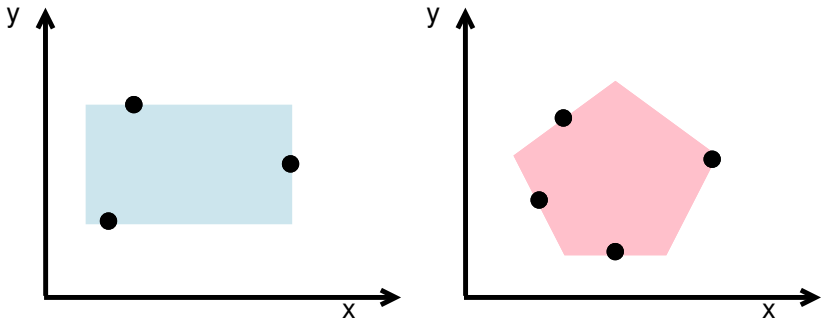
Les deux approches d'analyse statique par interprétation et par modèle se basent sur l'abstraction d'une trace d'exécution réelle du programme analysé.

L'interprétation abstraite [Cou00] :

elle consiste à déterminer des comportements (ou sémantiques) à partir de relations d'abstraction. La précision d'une sémantique peut varier en fonction du niveau d'abstraction choisi. La sémantique la plus concrète correspond à une description précise de l'exécution réelle d'un programme. Des sémantiques plus abstraites sont alors des approximations de la sémantique concrète qui ne considèrent qu'une partie de la trace d'exécution du programme analysé selon la précision souhaitée. Le calcul d'une sémantique abstraite permet donc de cerner le comportement d'une application, de façon grossière mais suffisante selon le besoin, avant même de l'exécuter. Plusieurs travaux se sont intéressés à la normalisation de la représentation des sémantiques abstraites afin de faciliter leur manipulation algorithmique. Prenons l'exemple de l'analyse numérique des variables où deux grands domaine d'abstraction numérique ont été développés : *le domaine des intervalles [CC76]* et *le domaine des polyèdres [KMW67].* Ce dernier a pu démontrer de bonnes performances en analyse de code et en optimisation automatique des boucles parallèles grâce à de nombreux avantages tel que : l'expressivité et la précision (représentation proche d'une exécution effective) et la facilité d'intégration de nouvelles heuristiques d'optimisation [BPCB10].

La vérification par modèle [CES86] :

est une méthode de vérification formelle (automatique) des systèmes à états finis concurrents. Elle est destinée pour la vérification des systèmes critiques aussi bien informatiques que électroniques. Il s'agit d'une méthode algorithmique de vérification appliquée sur le système ou sur une abstraction du système.



(a) Intervalle (b) Polyèdre

FIGURE 4.2 – Les deux figures représentent les valeurs possibles pour les variables x et y dans deux domaines numériques différents qui sont : l'intervalle (figure (a)) et le polyèdre 2D (figure (b)).

La vérification par modèle est constituée de 3 principales étapes :

- La modélisation : elle consiste à réécrire le programme sous forme d'une représentation spécifique (ou modèle). Une façon de le faire est d'utiliser *les graphes orientés*. Un graphe est composé de nœuds et d'arcs. Un nœud est un état du système et un arc est une transition entre deux états. Chaque état du graphe est étiqueté par un ensemble de formules logiques atomiques (ex. $a = 10$).

- L'expression des propriétés : Avant de procéder à la vérification, il est nécessaire de définir toutes les propriétés à vérifier. Ces propriétés sont formalisées grâce à des expressions logiques. La vérification par modèle utilise généralement la logique temporelle pour la formulation des propriétés. Certaines méthodes offrent des langages de spécification formelles basés sur des modèles formels tels que les réseaux de Pétri ou les logiques temporelles.

- La vérification : il s'agit d'un processus automatique qui vérifie si la propriété est bien respectée au niveau de la spécification. Tous les défauts détectés par l'algorithme de vérification sont ensuite reportés à l'utilisateur afin de modifier son programme.

La méthode de vérification par modèle présente l'avantage de résoudre le problème de complexité en ajustant les paramètres du modèle d'abstraction sous-jacent (ex. ajouter des niveaux d'abstraction). Cependant, la méthode de vérification par modèle peut générer de faux positifs dûs à une spécification incorrecte ou à une erreur dans la modélisation du système.

Il existe dans la littérature un large spectre d'outils d'analyse statique de code qui relèvent des domaines industriels et académiques différents. Le tableau 4.2 présente quelques uns des plus connus.

Outils	Référence	Méthode d'analyse	Cibles	Type de vérification
Lint	Laboratoires Bell [Joh77]	Vérification syntaxique	C	Détection d'anomalies et constructions inutiles
FindBugs	University of de Maryland [HP03]	Vérification syntaxique	Java	Détection d'anomalies
Spin	Laboratoires Bell [Hol04]	Vérification par modèle	C	Vérification des propriétés de logique temporelle pour les protocoles de communication
Polyspace	INRIA Grenoble [Cou00] acheté par MathWorks	Interprétation abstraite	C/C++ Ada	Détection des défauts de mémoire, division par zéro, boucle infinie, code inaccessible, etc.
Frama-C	CEA-LIST et INRIA-Futurs [CLIF08]	Interprétation abstraite et vérification formelle	C	Analyse des valeurs, preuve de programme avec la logique de Hoare
Goanna	NICTA [HFSB08]	Interprétation abstraite et vérification par modèle	C/C++	analyse de flot de données
Astrée	ENS,CNRS,INRIA [CCF$^+$05]	Interprétation abstraite	C	Garantie de l'absence de défauts à l'exécution pour des programmes critiques et complexes mais ne prend pas en compte l'allocation dynamiques de la mémoire
C Global Surveyor (CGS)	NASA [BBHK05]	Interprétation abstraite	C	Vérification de logiciels pour les missions spatiales
PAG	AbsInt GmbH [HBH$^+$07]	Estimation des pires temps d'exécution (WCET)	C	Vérification des propriétés liée à l'architecture
CodeSonar	GRAMMATECH [JJA08]	Interprétation abstraite et exécution symbolique	C/C++	Détection des défauts de code et des fragments suspects et analyse inter-procédurale

TABLE 4.2 – Outils d'analyse statique.

4.3.2 Discussion

Une des problématiques traitée dans le cadre de la thèse est d'adapter le choix des protocoles de cohérence au comportement de l'application. La réponse à cette question nécessite d'analyser les accès aux variables partagées. Nous proposons dans ces travaux un modèle abstrait pour la représentation des accès concurrents aux variables partagées. L'analyse de ce modèle permet d'affecter un protocole de cohérence à chaque accès à une donnée partagée. Une méthode de décision est également proposée pour permettre l'arbitrage entre plusieurs solutions possibles.

La suite de ce chapitre décrit le modèle d'analyse statique ainsi que l'outil d'aide à la décision proposés dans le cadre de cette thèse.

4.4 Contribution : modèle d'analyse des accès aux données partagées

La brique d'analyse statique de la chaîne de compilation proposée est chargée de la caractérisation du comportement de l'application pour l'aide à la décision dans le choix des protocoles de cohérence. Elle est constituée de deux principales étapes qui sont : la modélisation des accès aux données partagées sous forme de graphe de dépendance, le partitionnement et l'analyse de ce graphe pour le choix des protocoles de cohérence. La sous-section 4.4.1 présente le modèle d'analyse statique proposé. La section 4.4.2 décrit les différentes métriques utilisées pour l'analyse du programme d'aide à la décision.

4.4.1 Graphe de dépendance étiqueté pour l'analyse des accès aux données partagées

L'analyse statique de code basée sur des méthodes d'interprétation abstraite ou d'analyse de modèle (§ 4.3.1) permet de recueillir un certain nombre d'informations sur le déroulement d'un programme avant même de l'exécuter. Grâce à ces informations, le compilateur peut détecter certaines anomalies de fonctionnement, qui peuvent surgir lors de l'exécution. La fiabilité de certains systèmes critiques (ex. logiciels avioniques) en dépend fortement. Elles permettent aussi d'optimiser le code et de mieux l'organiser.

Les informations issues de l'analyse statique du code sont organisées sous forme de *graphes* (ou *arbre*).

Il existe différents types de graphes :

Arbre syntaxique : la première étape d'analyse d'un code est de vérifier qu'il respecte bien les règles syntaxique imposées par le langage de programmation. Un arbre syntaxique est donc une réécriture du programme initial en écartant tous les détails inutiles tels que les commentaires.

Graphe de flot de contrôle : graphe dont les nœuds sont des instruction et les arcs représentent des relations de dépendance entre les instructions. Il permet de représenter les boucles les instructions conditionnelles et les branchements. Chaque chemin dans le graphe est un scénario d'exécution possible du programme.

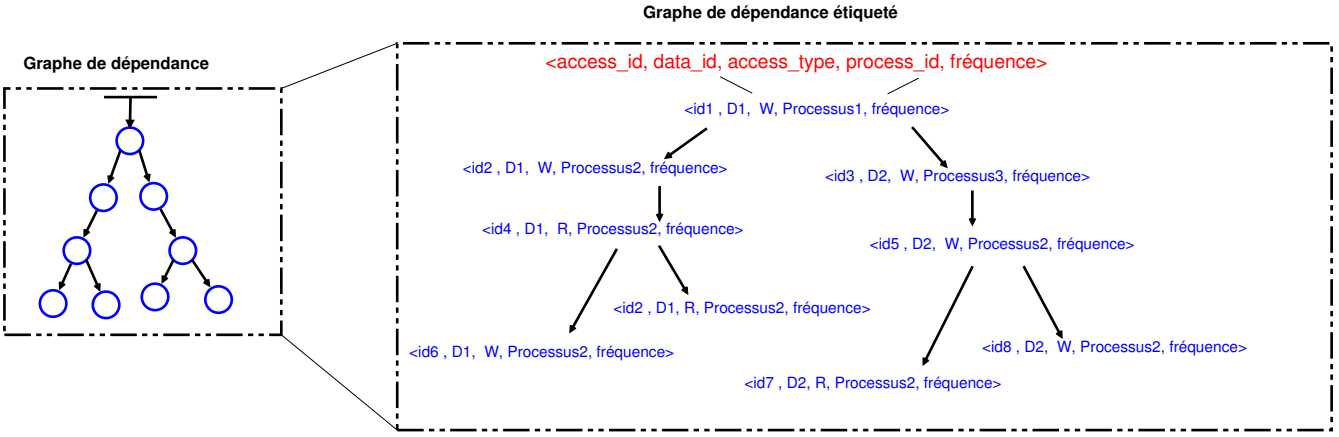


FIGURE 4.3 – Exemple de graphe de dépendance transformé en graphe de dépendance étiqueté. Les étiquettes représentent des informations sur les accès aux données.

Graphe de flot de données : représentation qui donne des informations sur le déplacement des données dans le programme ou entre le programme et la mémoire.

Graphe de dépendance de données : représente les dépendances entre les accès aux données. Les nœuds du graphe sont des instructions. Un arc relie deux nœuds que si une variables modifiée par le première instruction est utilisée par la deuxième sans qu'elle soit modifiée entre temps.

Le modèle de graphe proposé est une abstraction du graphe de dépendance des données où l'on s'intéresse qu'aux accès aux variables partagées. Les nœuds du graphe représentent des accès aux données partagées. Un arc relie le nœud a au node b s'il existe une relation de causalité de a vers b.

Définition 15 (Causalité)**.** La notion de causalité comme présentée par *Lamport* [Lam78] revient à définir un ordre partiel pour un ensemble d'événements du système. Il y a une dépendance causale entre deux événements e_1 et e_2 si le premier doit se dérouler avant le deuxième. $e_1 \rightarrow e_2$ désigne la relation de causalité entre e_1 et e_2. La causalité est établie si l'une des conditions suivantes est vraie :

- e_1 et e_2 appartiennent à un même processus et e_1 précède localement e_2.
- e_1 est l'émission d'un message et e_2 est la réception de ce message.
- il existe un événement e_3 tel que : $e_1 \rightarrow e_3$ et $e_3 \rightarrow e_2$.

Chaque nœud du graphe est étiqueté avec des informations liées à l'accès à la donnée partagée. Ces informations, issues de l'analyse statique du code, incluent : l'identifiant de l'événement effectuant l'accès, le type d'accès, l'identifiant de la variable partagée, ainsi que la fréquence d'accès à cette variable lorsqu'il s'agit d'une boucle.

La figure 4.3 est un exemple de graphe de dépendance étiqueté.

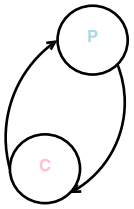


FIGURE 4.4 – Motif d'accès en producteur-consommateur.

Le graphe de dépendance étiqueté permet de caractériser les schémas d'accès aux données partagées ainsi que les spécificités de chacun de ces accès. L'analyse du graphe de dépendance étiqueté aide dans la prise de décision sur le choix des protocoles appropriés.

Une première analyse possible de ce graphe est de le partitionner. Le partitionnement d'un graphe consiste à le diviser en plusieurs sous-graphes selon différents critères. La méthode de partitionnement basée sur des motifs d'accès permet de mieux cerner le comportement de l'application grâce au repérage de motifs dans le graphe global. Un motif d'accès est un sous ensemble du graphe de dépendance qui peut être défini comme suit (définition 16) :

Définition 16 (Motif d'accès)**.** Un motif d'accès est une combinaison d'accès caractérisés par un comportement particulier. Il correspond à un schéma d'accès successifs ou concurrents à une ou plusieurs zones de la mémoire. Le *producteur-consommateur* décrit dans la figure 4.4 est un motif d'accès basé sur un schéma de succession des accès en lecture et écriture. La forme minimale d'un motif est un accès unique. La forme maximale correspond à l'ensemble des accès de l'application.

La répétition d'un motif permet de partitionner le graphe de contrôle d'une application en plusieurs phases. Chaque phase représente une tendance de comportement de l'application ce qui peut orienter le choix du protocole de cohérence associé à chaque accès dans la phase en cours.

Notons que les protocoles de cohérence sont classifiés selon leur efficacité avec les différents motifs d'accès mémoire connus. Le choix d'un protocole de cohérence va donc dépendre des motifs d'accès repérés dans chaque partition du graphe ainsi que d'une deuxième analyse portant sur le taux d'utilisation de chaque variable. Pour ce faire, la section 4.4.2 décrit les principales métriques d'analyse et d'aide à la décision pour le choix des protocoles de cohérence.

4.4.2 Métriques d'analyse et de décision pour le choix des protocoles de cohérence

Le graphe de dépendance défini précedemment (section 4.4.1) contient deux types d'informations : l'information sur les motifs d'accès de l'application et l'information concernant les caractéristiques de chaque accès.

Afin de caractériser les différents accès nous définissons la *fréquence d'utilisation* qui correspond au nombre d'accès à une donnée. C'est une métrique qui permet d'associer

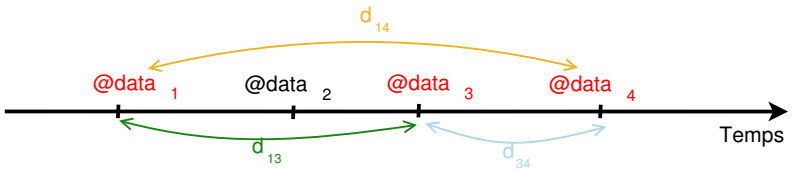


FIGURE 4.5 – La distance d'accès entre une séquence d'accès mémoire pour un même thread.

un taux d'utilisation par *thread* à chaque donnée partagée. Cette information reflète la répartition de la charge mémoire entre les *threads* aussi bien que le niveau d'utilisation de chacune des données. Il est possible d'obtenir cette information par analyse statique du code selon différentes approches. Par exemple, pour la définition du nombre d'itérations d'une boucle qui effectue plusieurs accès à une donnée trois cas se présentent :

1. Le nombre d'itérations est défini statiquement dans le programme. Il est facilement connu lors de l'analyse statique.
2. Le nombre d'itérations n'est pas explicitement défini dans le programme, mais il est possible de définir des bornes inférieure et supérieure de sa valeur.
3. Il n'est pas possible d'estimer le nombre d'itérations.

Dans la suite de ce travail nous considérons que le nombre d'itérations est connu.

Soient f_{ij} la fréquence d'utilisation de la donnée D_i par le *thread* T_j et Nb le nombre de données accédées par T_j. La charge mémoire moyenne du *thread* T_j est calculée par la formule 4.1 suivante :

$$charge_{moyenne/thread} = \frac{\sum f_{ij}}{Nb} \tag{4.1}$$

Afin de mesurer le degré de dépendance entre les accès aux variables partagées, une deuxième métrique est destinée à la détection des données fortement liées. Il s'agit de la *distance d'accès* aux données les plus fréquemment utilisées. Soit Nb le nombre de données d_i accédées par le *thread* T_j avec les fréquences f_{ij}. Pour toute donnée d_i dont la fréquence f_{ij} est supérieure à une fréquence seuil F_{seuil}. La distance entre deux données d_i et d_{i+k} présentée dans la figure (figure 4.5) est définie comme suit (définition 17) :

Définition 17 (Distance d'accès)**.** La distance d'accès $distance_{ij}$ entre deux données d_i et d_{i+k} est égale au nombre d'accès qui séparent les accès à ces données par un même *thread* T_j.

Pour un nombre Nb de données accédées par un *thread* T_j la somme des distances $distance_{ij}$ entre toutes les paires de données constitue la $distance_j$ globale associée au *thread* T_j. La distance d'accès moyenne par *thread* est donc égale à :

$$\text{distance moyenne} = \frac{distance_j}{Nb} \tag{4.2}$$

La définition de la distance d'accès pour un *thread* permet de mesurer le degré de dépendance entre les nombreux accès associés à ce *thread*. Une autre métrique en est déduite, c'est le *facteur d'interdépendance* défini comme :

$$\text{facteur d'interdépendance} = \frac{1}{distance_{moyenne} + 1} \tag{4.3}$$

Le facteur d'interdépendance permet de mesurer la dépendance entre les données d'un *thread* à partir de la distance entre les accès à ses données. Une grande distance correspond donc à une faible dépendance et inversement.

Une troisième métrique est proposée dans ce modèle qui consiste à calculer la variance de la fréquence d'utilisation des données d'un motif pour chaque *thread*. Cette métrique permet de mesurer la variation de la charge mémoire d'un même *thread*. Soit $\sum f_{ij}$ la somme des fréquences d'utilisation correspondant à la séquence des données d_i les plus fréquemment utilisées par le *thread* t_j, la *variance* est donc définie comme suit :

$$\text{variance} = \frac{\sum f_{ij} - f_{moyenne}}{f_{moyenne}} \tag{4.4}$$

Les trois métriques proposées *(la fréquence d'accès, le facteur d'interdépendance et la variance)* sont simples et intuitives. Elles représentent l'avantage d'un bas coût computationnel tout en permettant de caractériser la charge mémoire à chaque phase de l'application.

Elles sont utilisées par la suite par le processus de décision. La section 4.4.3 décrit une implémentation du modèle d'analyse statique ainsi que le processus de décision pour le choix des protocoles.

4.4.3 Implémentation du modèle d'analyse statique

L'analyse statique du code permet d'obtenir le graphe de contrôle qui décrit un déroulement possible du programme analysé. Le graphe de dépendance étiqueté présente les accès aux variables partagées en indiquant les spécificités de chaque nœud tels que le type et la fréquence d'accès.

Le travail proposé dans cette section a été effectué en collaboration avec *Tien Thanh Nguyen*, étudiant en master 2 recherche à l'université de Paris-Sud. Il porte sur l'implémentation du graphe de dépendance étiqueté à l'aide des réseaux de Pétri (*RdP*).

Représentation en graphe de Pétri

Un réseau de Pétri est une représentation mathématique des systèmes (informatiques ou autres) proposée en 1962 dans les travaux de thèse de *Carl Adam Petri* [Pet62]. Chaque réseau de Pétri est un graphe composé de quatre éléments : (1) *place*, (2) *transition*, (3) *arc* et *jetons*. Un *arc* relie une *place* à une *transition*. Il n'est pas possible de connecter deux *places* ni deux *transitions* entre elles. Le passage d'une place vers une transition est contrôlé par des *jetons*. Chaque *place* est dotée d'un nombre fini de *jetons* ce qui permet de représenter le déroulement d'une ou plusieurs évolutions du

Graphe de Pétri	Graphe de dépendance étiqueté
Place	correspond à un *thread* ou bien à un accès à une variable partagée. Les étiquettes permettent de distinguer entre une création de *thread* (place étiquetée *CREATE*) et une place d'accès à une variable (place étiquetée *WRITE/READ*).
Transition	une transition correspond à une action effectuée par le *thread*. L'étiquette de la transition détermine la nature de l'action effectuée.
Jeton	Un jeton signifie la présence d'une ressource (ex. variable partagée).

TABLE 4.3 – Analogies entre les éléments du réseau de Pétri et ceux du graphe de dépendance étiqueté

système. La distribution des *jetons* dans les places constitue un *marquage* du graphe qui correspond à état spécifique du système. Chaque état d'un *marquage* peut avoir plusieurs évolutions.

Dans notre étude le graphe de Pétri est créé à partir des éléments du graphe du graphe de contrôle qui nous concernent : (1) la création des *threads* (2) la création et d'accès aux données partagées, (3) les points de synchronisation et de communication entre *threads*.

La création du graphe de Pétri est basée sur une projection entre la structure d'un graphe de Pétri et les éléments du graphe de contrôle. Le tableau 4.3 décrit la correspondance entre chaque élément de Pétri et son analogue dans le réseau de dépendance étiqueté.

La figure 4.6 montre les étapes de construction détaillée d'un réseau de pétri pour des *threads* parallèles.

Chaque étape du graphe de contrôle est transformée dans la représentation formelle de Pétri selon la syntaxe (*place*, *transition*) définie précédemment. Un exemple simple d'une transformation du graphe de contrôle en un graphe de Pétri est présenté dans la figure 4.7.

Arbre de décision

L'association d'un protocole de cohérence à une variable partagée est effectuée sur plusieurs étapes. Chaque niveau de décision est basé sur les métriques définies dans la sous-section 4.4.2. L'approche proposée est un *arbre de décision* où chaque chemin correspond à un choix de protocole. La décision est raffinée en avançant vers les branches à chaque vérification d'une condition de décision. C'est un outil qui a l'avantage d'être

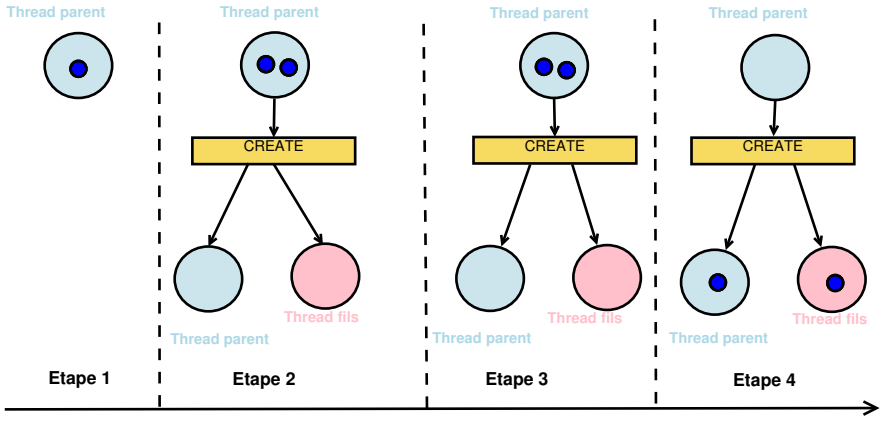


FIGURE 4.6 – Exemple de construction d'un réseau de Pétri en étapes : (étape 1) place de *thread* parent contenant un jeton qui représente la présence du *thread* parent. (étape 2) un deuxième jeton est ajouté à la place parent, il correspond à une nouvelle demande de création d'un *thread*. (étape 3) Les deux nouvelles places correspondent aux deux *threads* (parent/enfant) créés. (étape 4) La présence de deux *threads* sur les deux nouvelles places indique les disponibilités des deux *threads* créés.

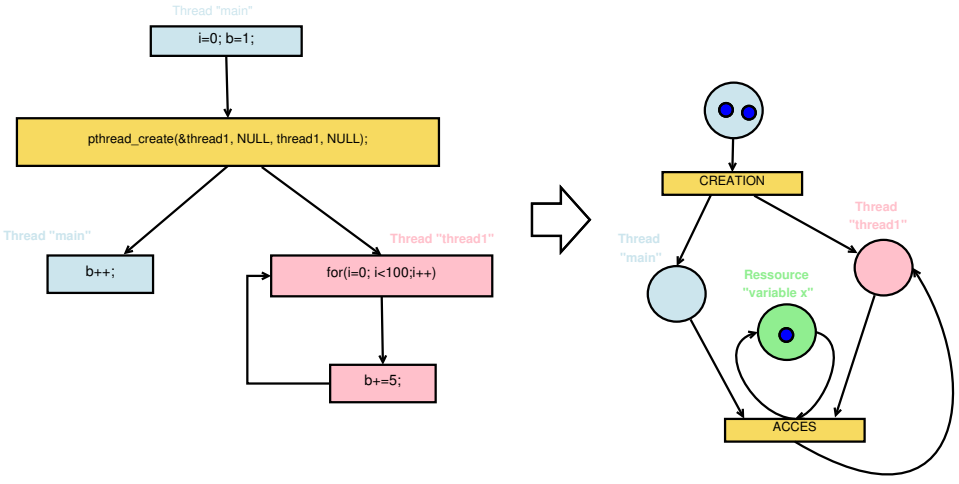


FIGURE 4.7 – Transformation d'un exemple de graphe de contrôle en un graphe de Pétri.

lisible, rapide à parcourir et flexible (on peut facilement intégrer de nouveaux protocoles). Un inconvénient de cet arbre de décision est qu'il se base sur l'expertise humaine au niveau de la caractérisation des protocoles de cohérence.

L'arbre de décision utilisé dans cette étude est constitué de trois étages de décision. Il considère les éléments suivants :

- Quatre protocoles de cohérence : Protocole *baseline* (protocole à répertoire), protocole *CoCCA*, protocole de *glissement de données*, protocole de *glissement de données* étendu par le *masse-ressort*.
- Trois conditions de décision basées sur : la *fréquence*, le *facteur d'interdépendance* et la *variance*.

Notons bien que les décisions sur les protocoles sont prises pour chaque *thread* et pour chaque variable. À la fin, les différents choix effectués sont fusionnés dans une phase d'arbitrage présentée plus loin.

Le **premier étage** de décision est basé sur la valeur de la fréquence globale $\sum f_{ij}$ qui correspond à la somme des fréquences d'utilisation de la séquence des données d_i les plus fréquemment utilisées. Cette dernière reflète le niveau de réutilisation des données par le *thread*. Deux cas se présentent :

1. Si $\sum f_{ij} < F_{seuil}$: le *thread* représente un faible taux de réutilisation des données. Le protocole *baseline* est bien adapté à ce genre de scénario.
2. Si $\sum f_{ij} >= F_{seuil}$: ceci signifie que le *thread* est caractérisé par une forte charge mémoire dûe aux accès multiples à la même séquence de données. il faut continuer à explorer les branches de l'arbre de décision afin d'aboutir à un choix de protocole de cohérence.

Le **deuxième étage** de décision porte sur l'analyse du *facteur d'interdépendance*. Deux branches sont possibles. Elles se présentent comme suit :

1. Si le *facteur d'interdépendance* est supérieur à un seuil D_{seuil} : ceci signifie que les données constituent une séquence d'accès qui sont fortement liés. Lorsque le *thread* accède à la donnée d_i, il y a de forte chance que le prochain accès portera sur la donnée d_{i+1} de la séquence. Ce cas est très favorable pour le protocole *CoCCA* qui se base sur le préchargement des motifs d'accès dans le cache afin de minimiser les requêtes de demandes d'accès.
2. Sinon : Les données sont fréquemment utilisées mais à des distances éloignées. Le préchargement n'est donc pas la meilleure solution dans ce cas.

Le **troisième étage** de décision permet de choisir entre le *protocole de glissement* à rayon fixe et celui à rayon adaptatif basé sur l'approche *masse-ressort*. Rappelons que ces deux protocoles permettent de préserver les données le plus longtemps possible sur la puce afin de garantir un coût bas d'accès à ces données lorsqu'elles sont fréquemment

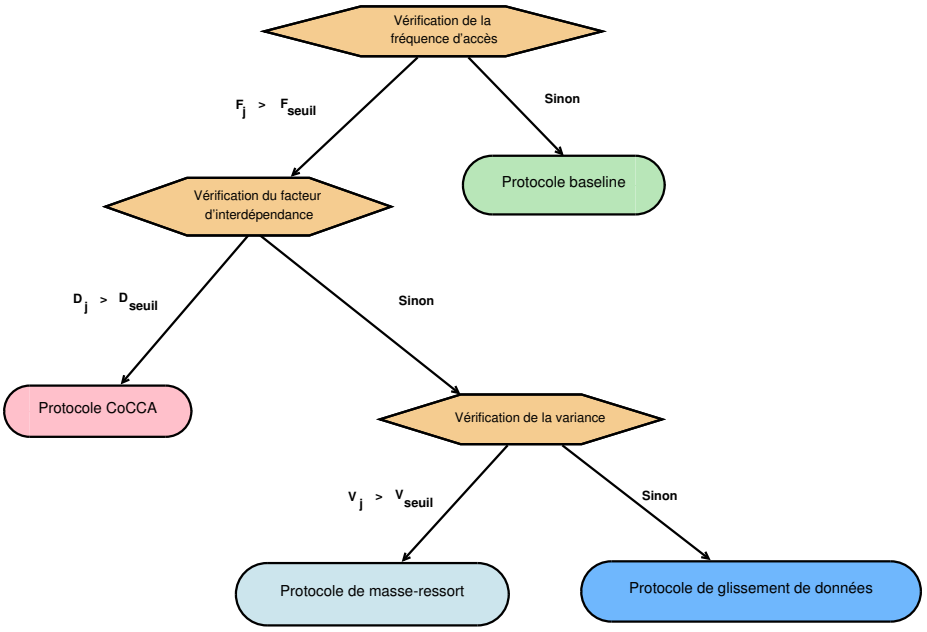


FIGURE 4.8 – Arbre de décision pour le choix des protocoles. Chaque passage dans une nouvelle branche est filtré par un seuil qui dépend de l'une des trois métriques qui sont : la fréquence d'accès, le facteur d'interdépendance et la variance. Les seuils de décision sont définis de manière empirique.

utilisées. Le premier est adapté à un comportement stable des accès mémoire, ce qui veut dire une faible variance de la fréquence d'accès aux données et dans ce cas il n'y a pas besoin de faire varier le rayon de glissement. Le deuxième est mieux adapté à comportement moins stable où la fréquence d'accès varie entre les différents accès mémoire. Ces deux cas se présentent comme suit :

1. Si la *variance* est supérieure à un seuil V_{seuil}, le protocole le mieux choisi est le glissement par *masse-ressort* car il permet d'adapter dynamiquement le rayon de glissement en fonction du besoin de l'application.

2. Sinon : le protocole choisi est le *glissement* à rayon fixe ce qui permet de définir un rayon de glissement fixe pour la totalité de la phase de l'application.

La figure 4.8 résume l'ensemble des étages et des conditions de décision ainsi que les protocoles correspondants.

La **dernière phase** de décision consiste à fusionner les choix des protocoles pour chaque donnée. L'arbre de décision précédent permet d'associer un protocole par donnée et par *thread*. Il est donc possible que plusieurs protocoles soient affectés à une même variable partagée ce qui ne garantit pas le respect du modèle de cohérence. Deux façons d'arbitrage sont possibles :

1. Privilégier la décision prise par le *thread* qui accède le plus à la variable.

2. Choisir le protocole dominant sur l'ensemble des décisions prises par chaque *thread*.

La définition des seuils de décision dépend de l'architecture cible (taille des caches) et de la granularité des données manipulées. Ces seuils peuvent en partie être calculés automatiquement.

Les techniques proposées pour le choix et l'arbitrage sont des basées sur une estimation grossière du comportement de l'application. Le raffinement des choix et du paramétrage des protocoles s'effectue dans la phase de configuration et de paramétrage des protocoles présentée plus loin (chapitre 6).

L'implémentation du modèle de réseau de Pétri utilise la bibliothèque *SNAKES*, un outil développé en python facilitant la construction et l'exécution des réseaux de Pétri [Pom04]. Un exemple de programme est traité dans le rapport de stage de *Tien Thanh* [Ngu15] proposant une étude de cas d'une application réelle de type *producteur-consommateur*.

4.5 Discussion

Nous proposons dans ce chapitre un modèle d'analyse statiques des accès aux données partagées utilisant une modélisation par réseau de Pétri. Cette modélisation formelle présente plusieurs avantages tels que la possibilité d'affecter différentes sémantiques aux éléments du graphe ou encore de leur associer plusieurs détails d'informations.

Nous proposons également un arbre de décision qui permet de faciliter l'arbitrage entre les différents choix de protocoles possibles. Un inconvénient de ce type d'outils décision est qu'il dépend de certain choix apportés par l'expertise humaine.

Nous envisageons comme perspectives d'étudier d'autres outils d'aide à la décision tels que la méthode par score (*en anglais Scoring*) par exemple qui permettra d'évaluer les performances des protocoles (leur donner des scores) lors de la phase de décision et non seulement à durant la configuration.

Chapitre 5

Modèles temporels d'évaluation des protocoles de cohérence

Dans le processus de raffinement du choix et de configuration des protocoles de cohérence de données la phase d'évaluation des solutions est une des principales briques du processus de décision. L'outil d'estimation des performances des solutions doit être suffisamment précis pour orienter la décision sur la configuration des protocoles suffisamment rapide pour s'intégrer dans la chaîne de compilation.

Afin de répondre à ces besoins, nous présentons dans cette section deux modèles temporels d'évaluation des protocoles de cohérence. Le premier modèle permet de calculer les temps d'accès aux données en cycle. Le deuxième modèle est une extension du premier basé sur un modèle *TLM (Transaction Level Modeling)* de *NoC* qui permet en plus d'estimer les temps d'accès, de prendre en charge la contention du *NoC* et de faire varier les paramètres du système (ex. différentes topologies du réseau sur puce).

La suite de ce chapitre est organisée comme suit : la section 5.1 présente l'état de l'art des techniques de simulation, la section 5.2 décrit le premier modèle d'analyse temporelle proposé, la section 5.3 présente la version étendu du modèle précédent qui prend en compte les contentions du *NoC*, la section 5.4 est une analyse des résultats d'évaluation des protocoles de cohérence avec les modèles temporels proposés.

5.1 État de l'art et motivation

Le choix des protocoles de cohérence nécessite d'évaluer plusieurs configurations de protocoles pendant la compilation. Une façon de faire serait de déployer toutes les solutions et de les tester sur une plate-forme réelle. Cette solution n'est pas toujours possible pour deux raisons ; la première est le coût temporel que le déploiement et l'exécution peuvent générer, la deuxième est liée à la disponibilité de la plate-forme d'exécution (disponibilité de la plate-forme d'exécution). D'autres méthodes de simulation sont possibles telle que la simulation *SystemC* [Sys04]. Ce type de simulation permet plusieurs niveaux de précision selon le niveau d'abstraction choisi. La vitesse des outils de simulation avec un niveau d'abstraction bas ne dépasse pas 10 MIPS

(ex. 3 MIPS pour le simulateur SESAM [VGS+10]). D'autres simulateur moins précis tels que *OVPsim* [OVP08] promettent d'atteindre des centaines de MIPS en fonction du comportement de l'application et de la complexité des processeurs utilisés dans la plate-forme. Le niveau d'abstraction et la complexité du système simulé jouent un rôle très important dans la détermination des performances de simulation. Cependant, la simulation des systèmes avec un grand nombre de cœurs reste un processus complexe et coûteux. Si la puce *MPPA 256-core* est capable d'atteindre des performances de 60 fps pour un encodeur vidéo *h264*, l'exécution de cette même application sur un simulateur de type *ISS* (*Instruction Set Simulator*) nécessite plusieurs minutes pour le calcul d'une seule image [ABB+13].

D'autres travaux proposent de la modélisation matérielle *FPGA (Field Programmable Gate Array)* [WHS07, MMB+03, GADM+05]. Ce type de modélisation promet une grande précision dans la conception du système. La principale difficulté d'une telle implémentation est la complexité de conception des circuits *FPGA*.

Une telle précision n'est forcément nécessaire dans notre cas d'étude et peut être très coûteuse en temps de simulation ce qui ne correspond pas forcément à nos besoins.

Pour la validation des protocoles de cohérence proposés précédemment (chapitre 2) nous avons proposé un modèle d'évaluation (*CacheValidator*) pour estimer le trafic sur le réseau sur puce dû au messages des protocoles de cohérence. Il s'agit d'un modèle analytique un modèle analytique de *NoC* qui fournit des statistiques sur les échanges de messages générés par les protocoles de cohérence sous-jacents. Il permet également d'obtenir la carte de répartition de la charge sur la puce pour une meilleure analyse du comportement de l'application en fonction de l'architecture cible. Le *CacheValidator* prend en entrée une trace d'accès mémoire de l'application en plus de certains paramètres de l'architecture tels que la topologie, la taille des caches et la taille de la puce.

Cette analyse garantit des temps de simulation du *NoC* réduits avec un niveau de précision et de flexibilité suffisants pour répondre à nos besoin d'évaluation des protocoles de cohérence. Cependant, ce modèle analytique ne donne aucune information sur les temps d'accès, les latences de communication ou encore la contention sur le *NoC*

Afin d'améliorer la qualité de notre évaluation sans sacrifier le temps de compilation, nous proposons un nouveau modèle temporel permettant d'obtenir plus d'information sur les performances temporelles de nos protocoles.

5.2 Modèle d'analyse temporelle pour l'évaluation des protocoles de cohérence

Cette section est dédiée à la présentation de deux modèles d'évaluation temporelle des protocoles de cohérence de données. Un premier modèle prenant en compte les communications générées par les protocoles qui gèrent les accès aux données mais qui ne prend pas en compte les conflits du réseau causés par ces communications. Pour répondre à ce besoin, une extension de ce premier modèle a été proposée. l'idée de cette proposition est d'utiliser un simulateur TLM de NoC pour le calcul des pénalités temporelles de chaque communication dans le réseau.

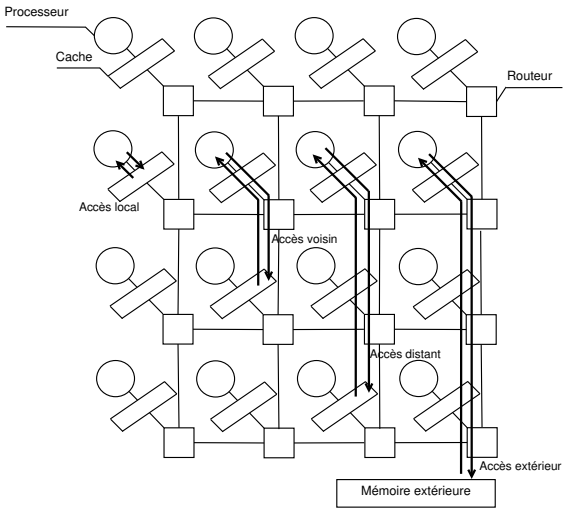


FIGURE 5.1 – Les quatre scénarios d'accès du modèle temporel : accès local, accès voisin, accès distant et accès extérieur.

5.2.1 Conception du modèle temporel

Le modèle temporel [CCD+15] décrit dans cette section a été réalisé en collaboration avec *Hamza Chaker*, étudiant en master 2 recherche à l'université de Bretagne Sud.

Ces travaux permettent de calculer les pénalités temporels de chaque accès à une donnée en fonction de l'emplacement de la donnée et de la topologie du réseau *NoC*. Il existe deux possibilités d'accès aux données : dans la mémoire sur puce (i.e. sur le cache d'un cœur) ou bien dans la mémoire extérieure.

En fonction du modèle de cohérence, les données peuvent avoir des emplacements différents. Le modèle proposé considère 4 emplacements mémoire qui sont (voir figure 5.1) :

- Accès local : la donnée est sur le cache local du cœur demandeur
- Accès voisin : la donnée est disponible dans le cache du voisin direct
- Accès distant dans la puce : la donnée est disponible sur le cache d'un cœur distant
- Accès en mémoire extérieure : la donnée n'est pas disponible sur la puce. La requête d'accès est transférée vers la mémoire extérieure.

Les cœurs sont connectés grâce à réseau en maille *2D*. Nous considérons dans cette proposition un algorithme de routage *X-Y*.

Le calcul des temps d'accès est décidé selon l'emplacement des données et le type d'accès (lecture ou écriture).

Temps d'accès au cache local cela correspond au cas d'un succès d'accès au cache local. Il est composé de deux temps :

- le temps d'accès au répertoire des métadonnées afin de repérer l'emplacement de la donnée dans la hiérarchie mémoire ($T_{meta-rd-cache-proc}$).
- le temps nécessaire à une lecture ($T_{data-rd-cache-proc}$) ou à une écriture ($T_{data-rd-cache-proc}$).

Le temps d'un accès en lecture est égal à :

$$T_{local-access-rd} = T_{meta-rd-cache-proc} + T_{data-rd-cache-proc} \tag{5.1}$$

Le temps d'accès en écriture est égal à :

$$T_{local-access-wr} = T_{meta-rd-cache-proc} + T_{data-wr-cache-proc} \tag{5.2}$$

Temps d'accès au cache voisin Lorsque la donnée est disponible sur le cache d'un voisin. Comme dans le cas précédent le temps d'accès comprend le temps nécessaire à localisation de la donnée et le temps d'accès en lecture/écriture à celle-ci dans le cache local. Il s'ajoute à cela une pénalité temporelle liée à l'acheminement de la donnée du cache voisin vers le cache local. Cette pénalité dépend principalement de la configuration du *NoC*. Dans le cas d'une lecture d'une donnée, celle-ci est transférée sur *NoC* et ensuite copiée localement. Le temps d'accès global pour une lecture est donc égal à :

$$\begin{aligned} T_{local-access-rd} = T_{meta-rd-cache-proc} + T_{ctrl-req-NoC} + T_{data-rd-NoC} \\ + T_{data-NoC} + T_{data-wr-cache-NoC} + T_{data-rd-cache-proc} \end{aligned} \tag{5.3}$$

Lorsque le type d'accès est une écriture il n'est pas nécessaire de copier la donnée sur le cache local, le calcul du temps d'accès est réduit à :

$$T_{local-access-wr} = T_{meta-rd-cache-proc} + T_{ctrl-req-NoC} + \quad T_{data-wr-cache-proc} \tag{5.4}$$

Temps d'accès au cache distant Lorsque la donnée est stockée dans le cache d'un cœur distant l'accès à cette donnée dépend de la manière dont sont gérés les déplacements de la donnée sur la puce. Dans le modèle proposé nous distinguons deux principales techniques : *le suivi et le chaînage*. **Le suivi** 5.2 correspond au cas où le cœur propriétaire d'une donnée reçoit une mise à jour à chaque fois que celle-ci est transférée dans le cache d'un autre cœur. Lorsque ce dernier veut accéder à sa donnée, il envoie directement une requête d'accès au dernier cœur qui a effectué accédé. Le calcul du temps d'accès est équivalent au cas précédent (*accès voisin*). L'information sur l'emplacement de la donnée est récupérée du répertoire local du cache ($T_{meta-rd-cache-proc}$), ensuite une requête est envoyée au cache distant ($T_{ctrl-req-NoC}$). Dans le cas d'une lecture ($T_{data-rd-NoC}$), la donnée est

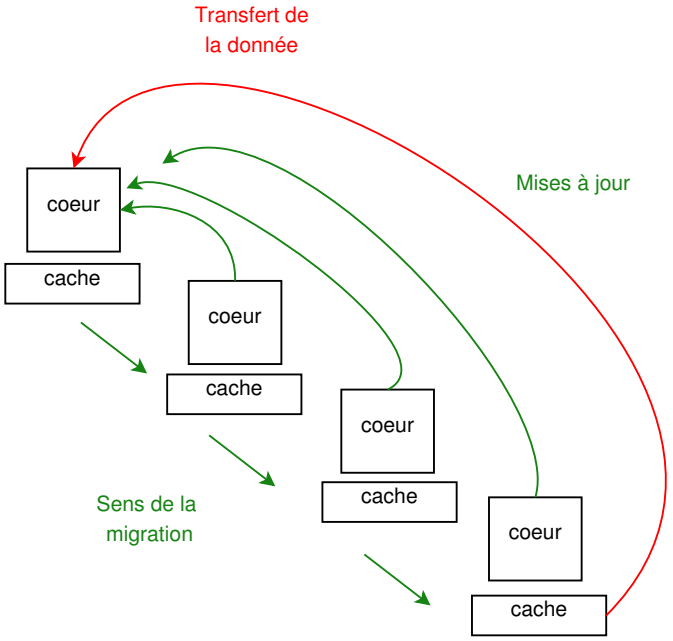


FIGURE 5.2 – Le mode suivi. Les mises à jours des déplacements des données sont toujours envoyées au cœur initial.

transférée vers le cache du cœur demandeur ($T_{data-NoC}$) et copiée dans le cache destinataire ($T_{data-wr-cache-NoC}$). Le temps d'accès global est exprimé comme suit :

$$T_{local-access-rd} = T_{meta-rd-cache-proc} + T_{ctrl-req-NoC} + T_{data-rd-NoC} + T_{data-NoC} + T_{data-wr-cache-NoC} + T_{data-rd-cache-proc} \quad (5.5)$$

Pour un accès en écriture l'équation 5.5 est simplifiée comme suit :

$$T_{local-access-wr} = T_{meta-rd-cache-proc} + T_{ctrl-req-NoC} + T_{data-wr-cache-proc} \quad (5.6)$$

Avec le mode **chaînage** 5.3 chaque cœur est prévenu du prochain déplacement de ses données. L'accès à la donnée consiste à acheminer la requête d'accès de cœur en cœur jusqu'à atteindre le dernier emplacement de la donnée. Le temps de récupération d'une donnée distante dépend entre autre du nombre de cœurs entre le cœur demandeur et celui ayant la donnée. Le calcul du temps global est présenté comme suit :

$$T_{local-access-rd} = \sum_{i=1}^{n} (T_{meta-rd-cache-proc} + T_{ctrl-req-NoC}) + T_{data-rd-NoC} + T_{data-NoC} + T_{data-wr-cache-NoC} + T_{data-rd-cache-proc} \quad (5.7)$$

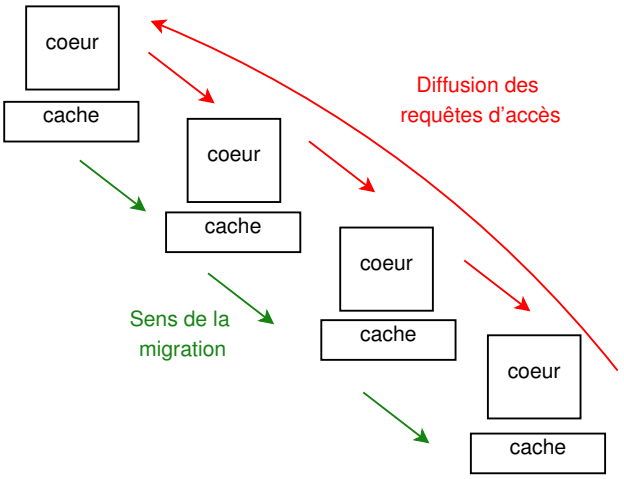


FIGURE 5.3 – Le mode chaînage. Les mises à jours des déplacements des données sont envoyées au cœur précédent. L'accès à une donnée engendre des requêtes d'accès diffusées de en cache en cache jusqu'à atteindre le cœur destinataire final

$$T_{local-access-wr} = \sum_{i=1}^{n}(T_{meta-rd-cache-proc} + T_{ctrl-req-NoC}) + T_{data-wr-cache-proc} \tag{5.8}$$

Temps d'accès à la mémoire extérieure L'accès à la mémoire extérieure se produit dans le cas d'un défaut de cache. Le temps d'accès à la mémoire extérieure est calculé de la même manière qu'un accès à un cache distant avec la prise en compte de la pénalité d'accès à la mémoire extérieure.

$$T_{local-access-rd} = T_{meta-rd-cache-proc} + T_{ctrl-req-NoC} + T_{data-rd-NoC} + T_{data-NoC} + T_{data-wr-cache-NoC} + T_{data-rd-cache-proc} \tag{5.9}$$

$$T_{local-access-wr} = T_{meta-rd-cache-proc} + T_{ctrl-req-NoC} + T_{data-wr-cache-proc} \tag{5.10}$$

5.2.2 Paramétrage du modèle temporel

Nous discutons dans cette sous-section des paramètres temporels du modèle proposé. Ce modèle propose des latences d'accès en nombre de cycles pour chaque pénalité temporelle considérée comme décrit dans la sous-section précédente 5.2.

De façon générale, nous considérons qu'un accès à une donnée quel que soit son emplacement est composé de 6 pénalités temporelles. Ces pénalités sont liées soit au réseau

NoC (transmission de la requête et le transfert de la donnée) soit à l'accès à la mémoire (accès aux métadonnées, accès au cache en lecture ou en écriture). L'équation 5.11 est une description générale du temps global d'accès à une donnée.

$$\begin{aligned} T_{total} = T_{meta-rd-cache-proc} + T_{ctrl-req-NoC} + T_{data-rd-NoC} \\ + T_{data-NoC} + T_{data-wr-cache-proc} + T_{data-rd/wr-cache-proc} \end{aligned} \tag{5.11}$$

Les temps nécessaire pour transférer respectivement des messages de contrôle $T_{ctrl-req-NoC}$ et des messages de données $T_{data-rd-NoC}$ dépendent des temps que passent chaque message dans les différents points de routage du *NoC*. Soit T_C le temps de commutation d'un paquet dans un routeur. T_C est exprimé comme suit :

$$T_C = T_{Arbitration} + T_{routing} + T_{switching-header} + T_{switching-payload} \tag{5.12}$$

Lorsqu'un paquet d'information arrive à l'entrée d'un routeur, il est stocké dans une file d'attente. Un routeur est constitué d'un commutateur qui assure la connexion entre les files (ou ports) d'entrées et files de sorties. Le routeur comprend également une unité de routage et d'arbitrage assurant l'aiguillage des paquets et la gestion des situations de contention.

Les algorithmes de routage et d'arbitrage gèrent les paquets en analysant leurs entêtes. Nous rappelons que l'entête contient toutes les métadonnées du paquet telle que sa taille et sa destination. La charge utile est le deuxième élément du paquet qui comprend les données transportées par ce dernier. Le nombre de cycles que nécessite le transfert de l'entête et de la charge utile d'un paquet dépend de la taille globale du paquet exprimée en nombre de *flits* (définition 18).

Définition 18 (Flit)**.** On appelle *flit (Flow Control Unit)* un mot de donnée de taille fixe qui compose un paquet dans un réseau sur puce. Le *flit* est souvent utilisé comme l'unité de mesure de la taille d'un paquet.

Nous considérons dans la suite de ce travail une implémentation logique de l'algorithme de routage ($T_{routing} = 0$). Dans une première étude aucune contention n'est prise en compte ($T_{Arbitration} = 0$). Le temps de commutation est simplifié à :

$$T_c = T_{switching-header} + T_{switching-payload} \tag{5.13}$$

Afin de calculer les temps de transfert d'un paquet, deux types de paquets sont considérés :

Paquet de contrôle : ce paquet est constitué de l'entête et des données de contrôle. Dans cette étude la taille de l'entête est fixée à *1 flit*. Les informations de contrôle occupent également *1 flit*. Le paquet de contrôle fait donc une taille de *2 flits*. Le temps associé à ce paquet est noté $T_{ctrl-req-NoC}$.

Paquet de données : ce paquet est composé de l'entête, de l'adresse mémoire et de la charge utile du paquet (i.e. donnée transférée). La taille de l'entête et de l'adresse mémoire est fixée à *1 flit* chacun. La taille de la charge utile du paquet dépend de la taille de la donnée. Elle est notée F_n *flits*. La taille totale du paquet de donnée est égale à : $2 + F_n$ *flits*. Le temps associé à ce paquet est noté $T_{data-NoC}$.

Notons bien que les temps considérés (en nombre de cycles) sont issus d'une étude de l'état de l'art des architectures sur puce [PH13]. Les scénarios d'accès distingués dans la sous-section 5.2 sont caractérisés par différents temps d'accès en fonction de la distance réseau parcourue par la donnée et de sa taille.

Pour un accès à un cache voisin : 2 routeurs sont impliqués : le routeur source et le routeur destination.

$T_{ctrl-req-NoC}$:

$$2(Routeurs) * (2flits * 1flit/cycle) = 4(cycles) \tag{5.14}$$

$T_{data-NoC}$:

$$2(Routeurs) * (2 + F_n flits * 1flit/cycle) = 2 * (2 + F_n)(cycles) \tag{5.15}$$

Le temps nécessaire afin d'effectuer un accès à une mémoire cache est fixé à $T_{data-rd-NoC} = 7$ cycles.

Pour un accès à un cache distant : Soit r le nombre de routeurs séparant les deux cœurs communicant.

$T_{ctrl-req-NoC}$:

$$r * (2flits * 1flit/cycle) = 2 * r(cycles) \tag{5.16}$$

$T_{data-NoC}$:

$$r * (2 + F_n flits * 1flit/cycle) = r * (2 + F_n)(cycles) \tag{5.17}$$

Pour un accès à la mémoire extérieure : La mémoire extérieure est vue comme un nœud du réseau *NoC*. Soit r le nombre de routeurs traversés qui dépend dans ce cas de l'emplacement du nœud mémoire.

$T_{ctrl-req-NoC}$:

$$r * (2flits * 1flit/cycle) = 2 * r(cycles) \tag{5.18}$$

$T_{data-NoC}$:

$$r * (2 + F_n flits * 1flit/cycle) = r * (2 + F_n)(cycles) \tag{5.19}$$

Le modèle proposé reste configurable en fonction des besoins de l'utilisateur et des caractéristiques de l'architecture cible. Le tableau 5.1 résume les temps considérés dans notre étude. Ces temps correspondent à des estimations moyennes pour notre étude qui peuvent être modifiées selon le besoin. Dans la suite de ce travail, nous proposons une extension de ce modèle qui prend en compte la congestion au niveau du réseau.

Temps d'accès	Accès local	Accès voisin	Accès distant		Accès extérieur
			Chaînage	Suivi	
$T_{meta-rd-cache-proc}$	7	7	7	7	7
$T_{ctrl-req-NoC}$	0	4	$2 \times r$	$2 \times r$	$2 \times r$
$T_{data-rd-NoC}$	0	7	7	7	100
$T_{data-NoC}$	0	$2 \times (2 + Fn)$	$r \times (2 + Fn)$	$r \times (2 + Fn)$	$r \times (2 + Fn)$
$T_{data-wr-cache-NoC}$	0	10	10	10	10
$T_{data-wr-cache-proc}$	0	10	10	10	10
$T_{data-rd-cache-proc}$	0	7	7	7	7

TABLE 5.1 – Modèle temporel (r correspond au nombre de routeurs traversés par un paquet, et F_n correspond au nombre de *flits* envoyés sur le NoC)

5.3 Modèle temporel étendu avec prise en compte de la contention réseau

Le modèle proposé précédemment permet de calculer les temps d'accès aux données dans différents emplacements dans la mémoire sur puce et à l'extérieur au niveau de la mémoire principale. Ce temps prend en compte les caractéristiques du réseau NoC ainsi que la taille de la donnée et sa distance par rapport au cœur demandeur. Cependant, nous avons considéré que la contention est nulle. Tous les paquets à l'entrée d'un routeur sont servis en même temps et aucun temps d'attente. Cette hypothèse simplifie le modèle temporel, mais ne correspond pas à ce qui se passe en réalité sur le réseau. Une extension de ce modèle à été développée afin d'améliorer la précision de notre modèle en prenant en compte la concurrence entre les requêtes d'accès dans le réseau.

Ce travail a été effectué en collaboration avec *Cédric Maignan*, étudiant en master 2 professionnel à l'université de Bretagne Sud, *Martha Johanna Sepúlveda* chercheuse postdoctorale à *l'Inria Rennes Bretagne Atlantique*.

5.3.1 Description du modèle d'analyse temporelle

L'analyse temporelle proposée est répartie sur deux principales phases. La figure 5.4 décrit ces étapes ainsi que les outils analytiques utilisés à chaque niveau.

La première phase consiste à extraire, à partir de la trace d'accès mémoire, toutes les communications réseau générées par les protocoles de cohérence associés à l'application. Ces communication sont présentées comme une liste d'événements correspondant à des accès mémoire. Pour ce faire, le *CacheValidator* (voir chapitre 2) est l'outil qui permet de simuler le comportement des protocoles sans passer par une exécution réelle de l'application.

La deuxième phase consiste à effectuer une analyse temporelle des communications issues du *CacheValidator*. Afin d'améliorer la précision du modèle temporel avec la prise en compte du phénomène de contention, la solution proposée utilise un modèle *TLM*

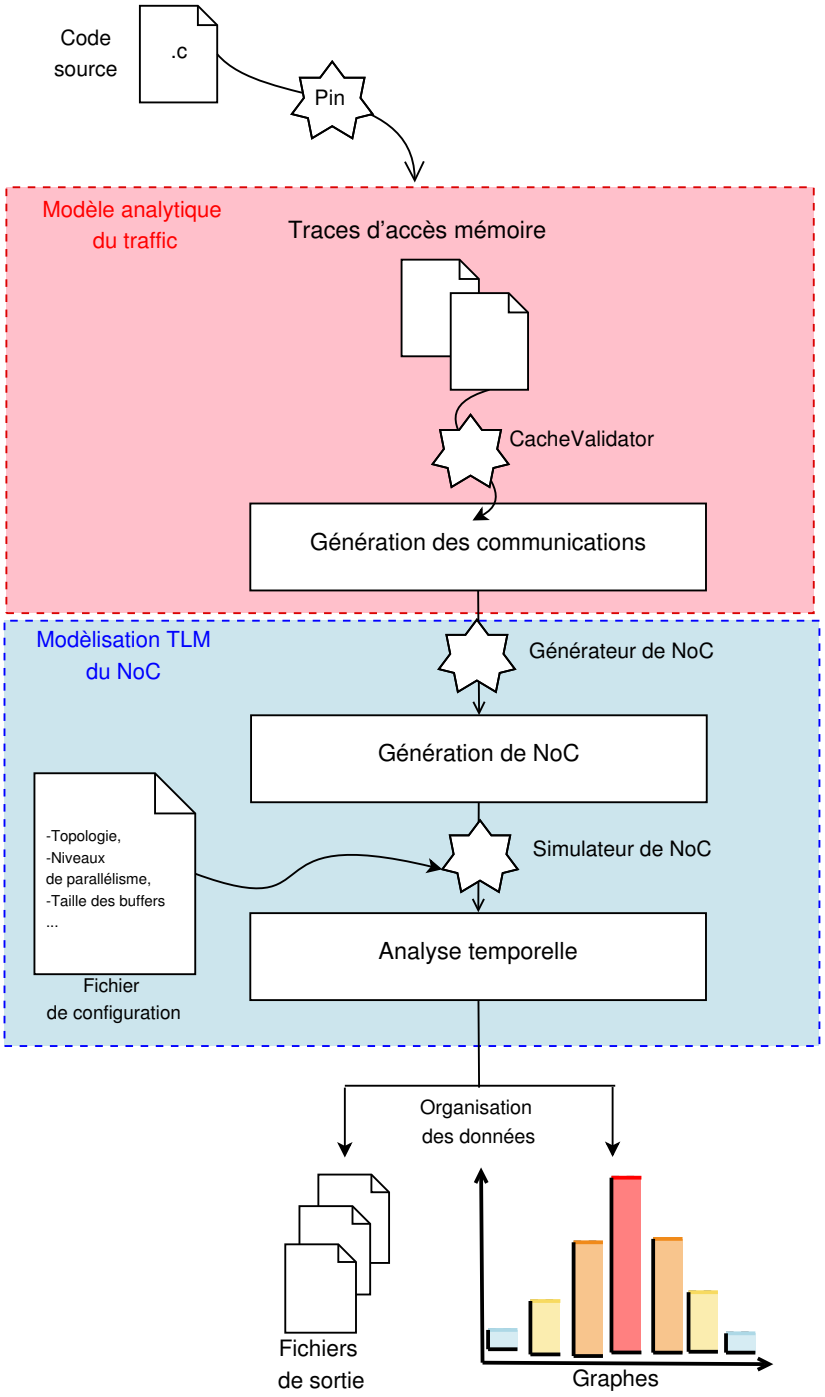


FIGURE 5.4 – Plate-forme d'analyse temporelle des temps de communication dans un *NoC* avec prise en compte de la contention au niveau des routeurs.

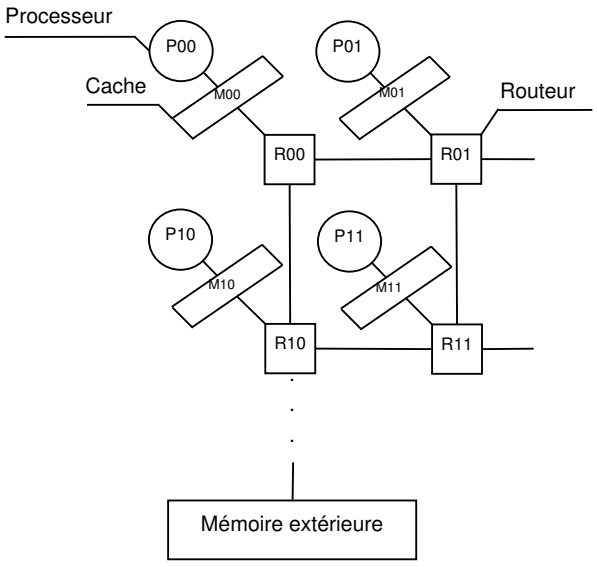


FIGURE 5.5 – La structure d'un réseau *NoC* dans le modèle proposé. Il consiste en 4 éléments qui sont les processeurs, les mémoires cache, les routeurs et la mémoire extérieure.

(Transactional-Level Modeling) de *NoC*. Ce modèle *TLM* de *NoC* a été développé dans les travaux de *Martha Johanna Sepùlveda et al.* [SSW07].

Cette phase est constituée de plusieurs étapes intermédiaires :

Génération du réseau *NoC* : la première étape de notre analyse consiste à générer le réseau *NoC* approprié au cas analysé. Le moteur de génération déduit les informations de configuration du *NoC* telles que la taille de la puce à partir de la trace des communications réseau. La figure 5.5 présente les différents éléments d'un réseau *NoC* dans notre modèle. La mémoire extérieure est modélisée par un nœud situé à l'extrémité du *NoC*. Afin de distinguer entre les paquets de contrôle et les paquets de données, deux types de nœuds sont représentés dans le modèle : le nœud mémoire et le nœud processeur. Enfin les routeurs sont les nœuds du réseau qui gèrent les communications entre l'ensemble des éléments du système. Chaque paire (processeur, mémoire cache) est reliée à un routeur qui assure le lien avec les autres pairs.

Modèle temporel du *NoC* : la création d'un modèle de *NoC* consiste à définir les paramètres temporels nécessaires pour la mesure des temps de communication. Les mêmes scénarios d'accès décrits dans la section 5.2 sont considérés dans ce modèle. Toute requête d'accès à une donnée passe d'abord par le cache local avant d'être mise sur le réseau. Les paramètres du modèle concernent le dimensionnement du *NoC* (ex. la taille des files d'entrée et de sortie, la bande passante), le dimensionnement des paquets de communication qui y circulent (ex. la taille de paquet de contrôle et de données) ainsi que les pénalités temporelles associées

Paramètre	Valeur
Taille de la puce	8x8
Type de NoC	Homogène
Mécanisme d'arbitrage	Tourniquet *(Round Robin)*
Topologie	Maille 2D, Tore 2D
Algorithme de routage	XY

TABLE 5.2 – Configuration des paramètres du *NoC*.

à chaque communication ce qui permet de définir le nombre de cycles pour la transmission d'un *flit* d'un point A à un point B du réseau. D'autres éléments de configuration peuvent être modifiés par l'utilisateur tels que les algorithmes de routage et d'arbitrage. Une modélisation de type *TLM* a pour avantage d'offrir un modèle configurable en fonction des besoins de l'utilisateur. Le tableau 5.2 montre la configuration considérée dans cette étude.

Métriques d'analyse des performances : le modèle d'évaluation dévelopé dans ces travaux propose plusieurs métriques d'analyse des performances. parmi ces métrique nous citons : (1) le temps de simulation qui permet d'estimer le coût d'évaluation associé au processus d'évaluation, (2) les temps d'accès par donnée et par cœur qui permettent d'analyser les pénalités temporelles générées par un protocole de cohérence, (3) le temps de contention par routeur qui met en valeur l'influence des conflits d'accès sur les performance du *NoC* ainsi que plusieurs autres métriques permettant une analyse plus détaillée selon les types d'accès mémoire et l'emplacement des données.

Le modèle analytique de *NoC* (*CacheValidator*) permet d'obtenir des statistiques sur les communications générées par les protocoles de cohérence mais il ne prend en compte l'impact de la concurrence car il traite tous les accès de façon séquentielle.

Le modèle temporel proposé offre donc la possibilité de définir le nombre d'accès mémoire envoyés parallèlement sur la puce. Ce paramètre permet de configurer la charge mémoire du système (e.i. différents niveaux de stress de la puce). Il est possible d'envoyer tous les accès en même temps ce qui correspond à un taux de parallélisme d'accès de 100%. Il est aussi possible de traiter les accès de façon séquentielle, ce qui signifie un taux de parallélisme d'accès nul. L'utilisateur a la possibilité de choisir des taux de parallélisme variables pour analyser les performances des protocoles dans différents contextes.

La suite de cette section est consacréé à une étude expérimentale comparative des protocoles de cohérence en fonction du comportement de l'application et la configuration du *NoC* sous-jacent.

Trace mémoire	Nombre d'accès
$Trace_1$	329
$Trace_2$	669
$Trace_3$	3396
$Trace_4$	3403
$Trace_5$	13758

TABLE 5.3 – Nombre d'accès mémoire par trace

5.4 Évaluation temporelle des protocoles de glissement de données

Cette section présente des résultats expérimentaux d'anslayse temporelle des protocoles de glissement de données à l'aide du modèle *TLM* décrit dans la section précédente. La première analyse porte sur la variation des performances du système en fonction des différentes configurations de plate-forme. Une deuxième analyse porte sur l'étude de différentes applications synthétiques (correspondant à des charges mémoire différentes) et sur l'étude de l'impact de la variation de niveau de parallélisme sur les performances du *NoC*.

5.4.1 Variation de la topologie du *NoC*

L'objectif de cette première analyse est de montrer la dépendance entre le comportement des protocoles de cohérence étudiés et l'organisation de la plate-forme cible. Les protocoles de glissement de données autorisent à chaque donnée de se déplacer dans la puce en fonction de la spécification de l'architecture d'exécution. Comme expliqué dans le chapitre 2 le rayon maximal de migration d'une donnée est fortement dépendant de la taille de la puce. Lorsque le nombre de cœurs augmente, le diamètre de migration des données devient plus grand. Un autre axe de dépendance entre les protocoles de glissement de données et l'architecture est la topologie du *NoC*. Le prochain cœur destinataire d'une donnée est l'un des k voisins du cœur source. La taille du voisinage est donc un paramètre clé pour ce type de protocoles. Nous considérons dans cette étude deux types de topologies 2D : topologie en maille, topologie en tore. Contrairement à la topologie en maille, les cœurs sur les bordures d'une topologie tore ont le même nombre de voisins que ceux situés au milieu de l'architecture. La métrique utilisée pour l'évaluation des performances est la latence d'accès cumulative pour l'ensemble des accès mémoire. Le tableau 5.3 donne le nombre d'accès mémoire pour chaque trace d'accès générée aléatoirement afin d'obtenir des comportements et des charges mémoire différents. La charge mémoire est répartie de façon déséquilibrée sur les cœurs pour créér des points chauds sur la puce.

Le tableau 5.4 montre une comparaison entre les latences d'accès obtenues avec une topologie en maille et celles obtenues avec une topologie en tore.

Trace mémoire \ Topologie	Maille	Tore
$Trace_1$	7778 cycles	9813 cycles
$Trace_2$	89625 cycles	87588 cycles
$Trace_3$	828885 cycles	772475 cycles
$Trace_4$	1836805 cycles	1969467 cycles
$Trace_5$	3719665 cycles	3584895 cycles

TABLE 5.4 – Les latences cumulatives obtenues pour chaque trace mémoire étudiée avec les deux topologies maille et tore. Une plus faible latence correspond à une meilleure performance.

Nous constatons dans ces résultats que pour une partie des traces mémoire de meilleures performances sont obtenues grâce une topologie Tore. Pour les protocoles de glissement de données une topologie en tore permet d'étendre la migration des données sur les extrémités de la puce. La topologie tore peut avoir un effet négatif dans d'autres cas si la coopération entre les cœurs des deux extrémités de la puce pénalise les performances de ces derniers. Le choix d'une configuration du *NoC* est un compromis entre le comportement général de l'application et la répartition de la charge sur la puce.

5.4.2 Variation du rayon de glissement

Dans cette deuxième analyse nous étudions l'impact de différents niveaux de parallélisme d'accès et différents rayons de glissement. Les messages générés par le *CacheValidator* n'importent aucune information sur les temps de transmission de chaque message. Ce temps de transmission dépend de nombreux facteurs dont le niveau de contention générée dans le *NoC*. Cette contention est contrôlée par le paramètre du niveau de parallélisme. En premier lieu le niveau de parallélisme est maintenu à 50%, seul le rayon de migration est varié entre trois valeurs ($valeur_{min} = 1$, $valeur_{moy} = 7$, $valeur_{max} = 14$). La figure 5.6 montre la variation du taux de contention pour les 5 traces mémoire présentées précédemment en fonction des valeurs du rayon de glissement.

Le taux de contention augmente généralement avec l'augmentation du rayon de glissement. Cette information permet de limiter le rayon de glissement afin d'éviter des taux de contention du réseau très élevés ce qui dégraderait drastiquement les performances du *NoC*.

Ensuite, le rayon de glissement est fixé à 1 et c'est le niveau de parallélisme qui prend des valeurs entre : 0% et 100%. Le tableau 5.5 montre les taux de contention correspondants à différents niveaux de parallélisme. Ce tableau montre que pour les traces avec une plus faible charge mémoire ($Trace_1$) une simulation complètement parallèle des accès double le taux de contention du réseau. En revanche, pour des traces plus stressées (avec un plus grand nombre d'accès mémoire) le niveau de parallélisme de la simulation a moins d'impact sur la dégradation des performances du *NoC*. Ce dernier constat est expliqué par le fait que la migration des données dans la puce génère un grand trafic d'abord pour éloigner les données et ensuite pour les récupérer. C'est un

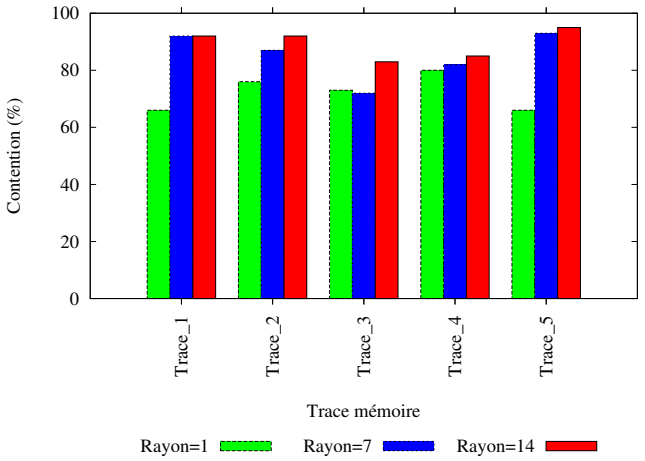


FIGURE 5.6 – Le taux de contention en fonction du rayon de glissement.

Traces - Niveaux de para.	0%	25%	50%	75%	100%
$Trace_1$	39%	39%	28%	45%	57%
$Trace_2$	83%	88%	93%	89%	95%
$Trace_3$	80%	76%	87%	87%	89%
$Trace_4$	66%	76%	73%	80%	66%
$Trace_5$	97%	81%	71%	78%	92%

TABLE 5.5 – Taux de contention pour différents niveaux de parallélisme allant de 0% (simulation séquentielle) à 100% (simulation parallèle).

des désavantages d'utiliser une telle approche. Il n'est donc pas judicieux d'utiliser le glissement de données avec toutes les charges mémoire et avec tout type de réseau *NoC*.

5.4.3 Étude comparative entre les protocoles de cohérence

Dans cette étude 5 protocoles sont considérés : le protocole *baseline*, le protocole de glissement à rayon minimal (1), le protocole de glissement à rayon moyen (7), le protocole de glissement à rayon maximal (14) et le protocole masse-ressort. L'idée est de comparer les performances obtenues (en terme de latence) pour chacune des traces mémoire (tableau 5.3) avec les différents protocoles considérés tout en faisant varier le taux de parallélisme entre les valeurs 0%, 25% et 50%.

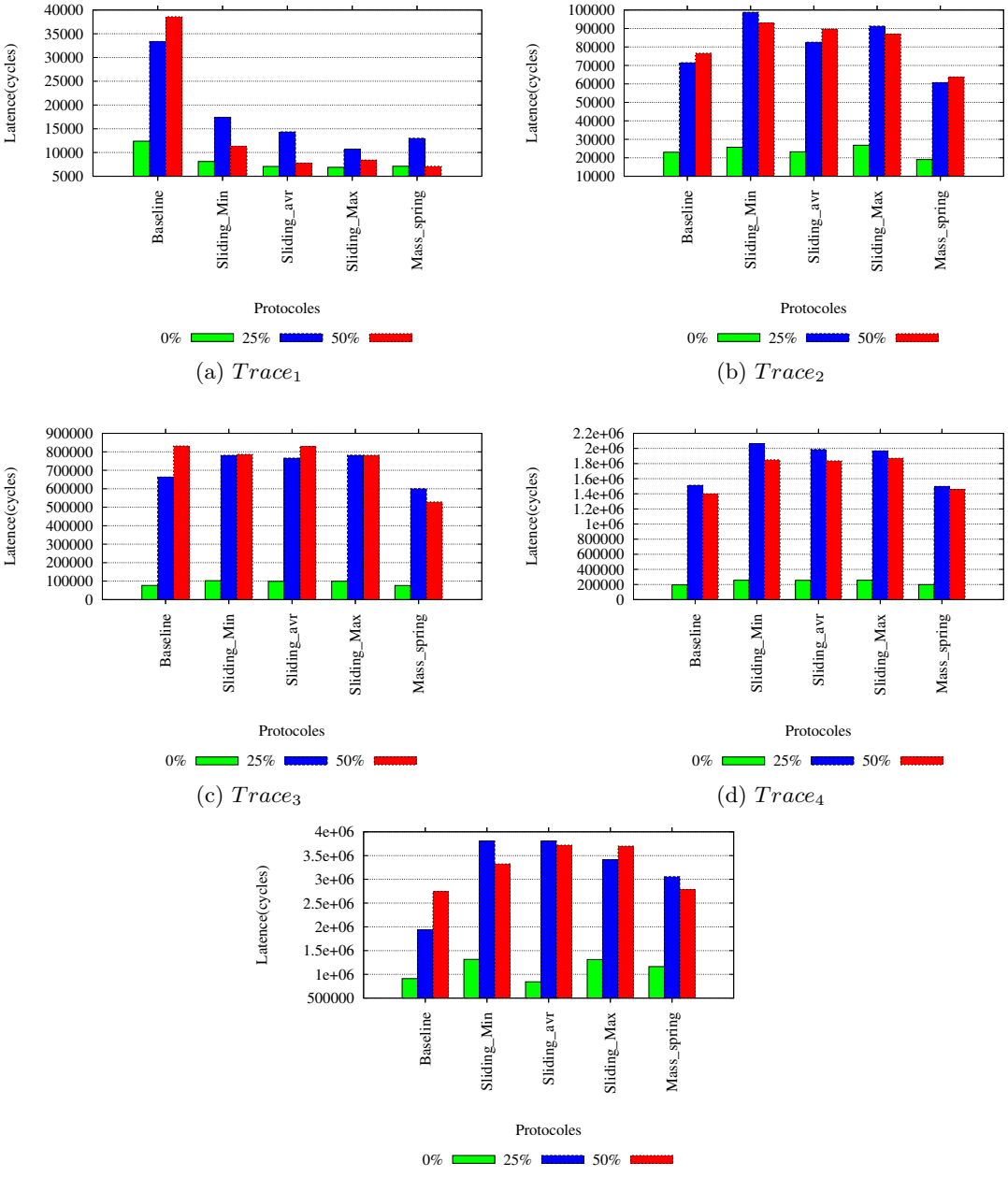
La figure 5.7 montre que pour la $Trace_1$ les protocoles de glissement et de masse-ressort sont plus performants que le protocole *baseline*. Nous observons à partir des résultats que les performances obtenues avec un niveau de parallélisme nul sont bien meilleures que les performances obtenues avec injection de la contention dans le réseau. Ceci est expliqué par le fait que le parallélisme est responsable de la dégradation des performances du *NoC* à cause des conflits entre les messages circulant en même temps sur le réseau.

Pour des traces plus complexes (un plus grand nombre d'accès mémoire) les protocoles de glissement de données ne sont pas spécialement plus efficaces (ceci dépend de la répartition de la charge dans la puce). Cependant, le protocole masse-ressort, caractérisé par son adaptabilité au comportement de l'application et à la répartition de la charge, fait preuve de meilleures performance dans la plupart des scénarios étudiés.

Un dernier point d'analyse est le temps de simulation. Les mesures du temps de simulation ont été effectuées avec un niveau de parallélisme de 50%. Notons bien que le temps de simulation ne dépasse pas 3661 millisecondes, ce qui est largement en dessous d'une simulation au niveau des portes logiques (*en anglais Gate level simulation* [AC14]).

5.5 Discussion

Le modèle temporel proposé dans le cadre de ce travail permet d'abord d'obtenir une évaluation en nombre de cycles des performances des protocoles de cohérence, ce qui n'était pas possible avec le modèle analytique. L'extension de ce modèle en utilisant un simulateur TLM de NoC permet d'augmenter la précision du modèle en prenant en compte l'effet de la contention du réseau sur les performances globales des protocoles. Une telle simulation offre plus de flexibilité à l'utilisateur dans la configuration de la plate-forme cible (topologie, taille de la puce, algorithme de routage, etc). Elle permet également de configurer le niveau de stress du NoC à travers le paramètre du niveau de parallélisme qui détermine le taux de communications injectées en parallèle dans le réseau. Le temps de simulation est de l'ordre de quelques milliers de millisecondes pour des traces d'accès complexes, ce qui reste raisonnable par rapport à d'autres outils de simulation. Cet avantage est d'une grande importance dans le cadre de ces travaux de thèse car il peut s'intégrer facilement dans la processus de compilation proposé. Enfin, l'étude des performances de protocoles présentée est un exemple d'analyse pouvant orienter le choix et le paramétrage des protocoles de cohérence.



(a) $Trace_1$ (b) $Trace_2$

(c) $Trace_3$ (d) $Trace_4$

(e) $Trace_5$

FIGURE 5.7 – Comparaison en les latences d'accès des différentes traces mémoire en utilisant différents protocoles de cohérence. Une latence plus basse correspond à de meilleures performances du protocole.

Chapitre 6

Paramétrage des protocoles de cohérence à l'aide d'une approche génétique

Le chapitre précédent (chapitre 4) décrit toute la chaîne de compilation permettant de prendre la décision sur le choix des protocoles en fonction de l'application et de la cible d'exécution. Ce chapitre décrit la phase principale de cette chaîne de décision qui est la brique de configuration des protocoles de gestion de cache. Le choix des paramètres des protocoles configurables influence les performances de ces derniers.

Ce chapitre est organisé comme suit : la section 6.1 présente les problèmes d'optimisation et les méthodes qui les traitent. La section 6.1.2 discute de l'état de l'art des méthodes d'optimisation multi-objectifs, et la section 6.2 décrit le problème de configuration des protocole de gestion des données et l'approche d'optimisation utilisée pour le paramétrage de ces protocoles. Finalement une étude de cas d'application sur le protocole de glissement de données est présentée dans la section 6.3.

6.1 Problème de paramétrage des protocoles de gestion des données

Dans la chaîne de décision définie précédemment (chapitre 4), la brique de configuration des protocoles de gestion assignés à l'ensemble des accès mémoire est un élément clé de décision permettant d'adapter le comportement de ces protocoles au contexte applicatif.

Nous présentons dans cette section le problème de configuration des protocoles de gestion des données. Un protocole peut avoir un ou plusieurs paramètres. Chaque paramètre est défini sur une plage de valeurs permettant de configurer le protocole selon différents critères liés aux performances du système. Le protocole de glissement de données est un exemple de protocole paramétrable avec le rayon de propagation comme principal paramètre car il définit le degré de liberté accordé à une donnée sur la puce.

Un autre exemple de protocole paramétrable est le protocole *CoCCA* [MLC+12] basé sur le préchargement de motifs d'accès où le choix de la longueur de ces motifs est déterminant pour les performances du protocole.

6.1.1 Définition des problèmes d'optimisation

L'optimisation est un axe de recherche important dans le domaine des mathématiques appliquées. Elle a été utilisée dans la résolution de nombreux problèmes d'ordonnancement tels que les tournées de véhicule et la planification dans le domaine de la logistique ou encore le dimensionnement et la répartition des réseaux dans le domaine informatique et dans les télécommunications.

Les algorithmes d'optimisation ont comme objectif de trouver les meilleurs paramètres d'un système tout en respectant les différentes conditions et contraintes définies par le système.

La première étape dans la résolution d'un problème d'optimisation est la modélisation de ce problème. Cette modélisation permet de positionner le problème étudié en fonction des types de problèmes d'optimisation connus. Ensuite, la détermination de la fonction objectif qui permet de définir le lien entre les paramètres du système et ses contraintes. La fonction objectif peut avoir une forme complexe, non-linéaire ou non-différenciable.

Le but du processus d'optimisation est de trouver un optimum global de la fonction objectif. Les algorithmes d'optimisation ont comme objectif, en plus de trouver l'optimum global, de réduire le coût de calcul de cette solution et de garantir la robustesse de fonctionnement et la facilité d'application sur différents problèmes d'optimisation.

Mathématiquement parlant, un problème d'optimisation se présente sous la forme suivante :

$$\begin{cases} \texttt{minimiser ou maximiser} & f(\vec{x}) & (\texttt{fonction objectif}) \\ \qquad\qquad\qquad\quad avec & \vec{g}(\vec{x}) & < 0 \\ \qquad\qquad\qquad\quad et & \vec{h}(\vec{x}) & = 0 \end{cases}$$

Les vecteurs $\vec{g}(\vec{x})$ et $\vec{h}(\vec{x})$ représentent respectivement m contraintes d'inégalité et p contraintes d'égalité. L'ensemble des contraintes délimite l'espace de recherche de la solution optimale.

La fonction objectif f est le nom donné à la fonction de coût (critère d'optimisation). C'est la fonction que l'algorithme d'optimisation cherche à optimiser (trouver un optimum global). Le vecteur $\vec{x}$ regroupe les variables de décision. La variation de ce vecteur fait varier la fonction objectif et permet de chercher un optimum.

Les problèmes d'optimisation sont classés selon les précédentes caractéristiques :

1. Nombre de variables de décision
   - une seule variable
   - plusieurs variables
2. Type de variables de décision

- Nombre réel continu
- Nombre entier ou discret
- Permutation sur un ensemble fini de nombres (combinatoire)

3. Type de la fonction objectif
   - Fonction linéaire
   - Fonction quadratique
   - Fonction non linéaire
4. Formulation du problème
   - Avec contraintes
   - Sans contraintes

La formulation précédente est relative à un problème dans lequel un optimum est recherché pour une seule fonction objectif f. Mais en pratique, il est question de plusieurs objectifs pour un seul problème d'optimisation. Par exemple, nous pouvons imaginer vouloir obtenir de bonnes performances d'exécution à basse consommation énergétique. Dans ce cas, on parle d'optimisation multiobjectif (ou optimisation multicritères). La fonction $f(\vec{x})$ devient un vecteur de plusieurs fonctions objectifs $\vec{f}(\vec{x})$.

Le but d'une optimisation multiobjectif dans la résolution d'un problème est d'optimiser au mieux les différents objectifs. Les différents critères d'optimisation sont souvent contradictoires. Si les objectifs ne sont pas contradictoires, le problème est ramené à un problème mono-objectif tout en gardant l'équivalence entre les deux problèmes. Deux objectifs sont dits contradictoires lorsque toute solution optimisée par rapport à un objectif ne l'est pas forcément pour l'autre. Contrairement à une optimisation mono-objectif, l'optimisation multiobjectifs aboutit à de multiples solutions du fait de la contradiction entre les différents objectifs. En effet, si on prend l'exemple de dimensionnement de cache dans une architecture sur puce, l'amélioration des performances mémoire de manière intuitive en augmentant la taille des caches augmente le coût d'accès aux données.

L'espace de recherche est composé d'un grand nombre de solutions possibles. Un concept qui permet de délimiter l'espace des solutions est de chercher un *compromis*. Les solutions définis par le *compromis* optimisent un certain nombre d'objectifs tout en dégradant les performances d'autres objectifs, mais permettent d'atteindre un objectif globale .

6.1.2 Algorithmes d'optimisation

Un algorithme d'optimisation est une méthode de recherche d'un optimum global dans l'espace des solutions possibles du problème traité. Les caractéristiques souhaitables d'un algorithme d'optimisation incluent sa capacité à chercher la solution optimale avec un petit nombre de paramètres de contrôle et un faible coût de calcul, en plus de sa robustesse et son adaptation à d'autres types de problèmes.

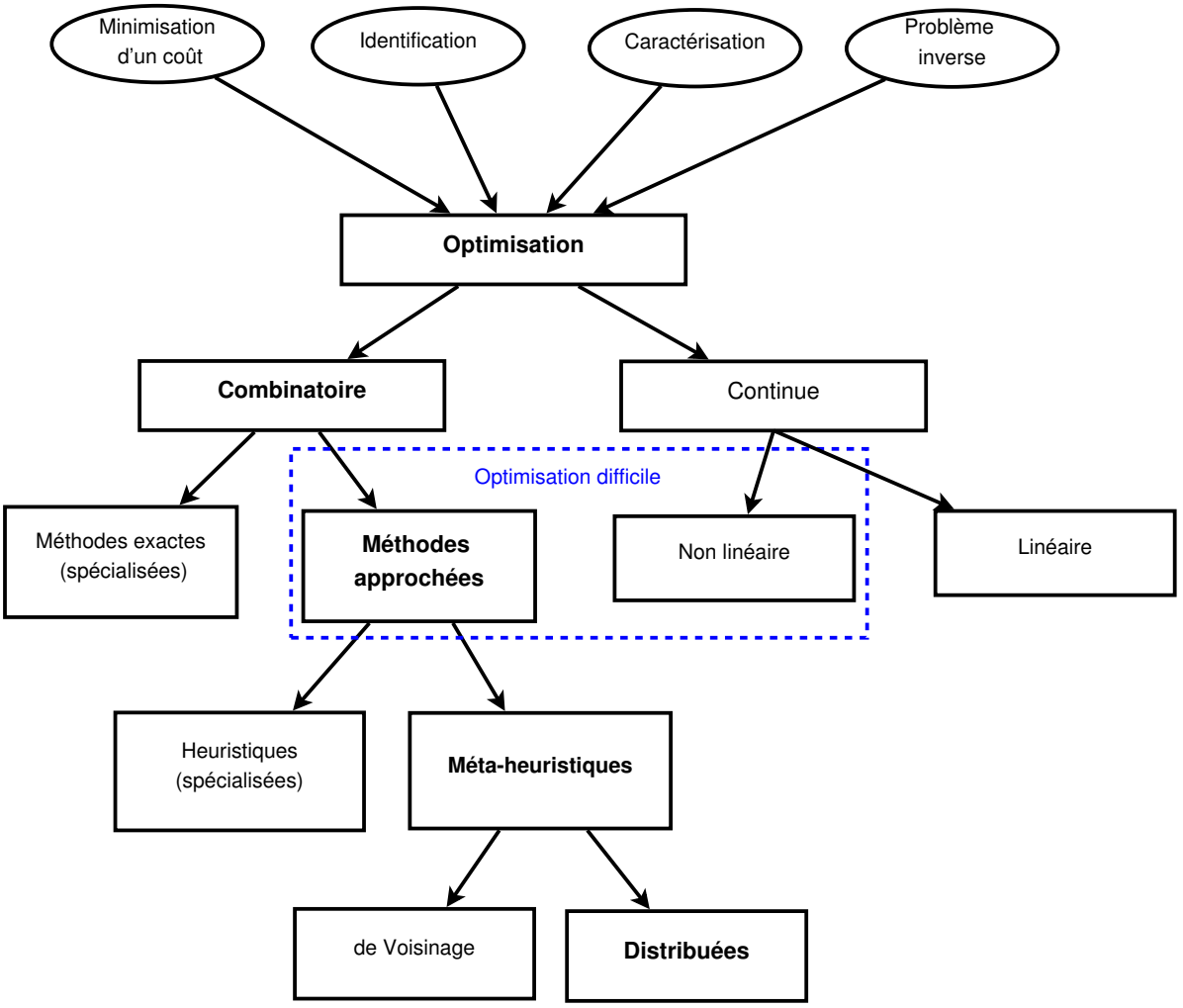


FIGURE 6.1 – Classification des méthodes d'optimisation mono-objectif [CS13]. Les classes mises en gras correspondent au positionnement de notre problème.

Il existe différentes méthodes d'optimisation. La figure 6.1 présente une classification de ces méthodes selon les différents types de problèmes.

La recherche d'une solution optimale à l'aide de méthodes mathématiques exactes ne peut être appliquée sur tous les cas à cause de la complexité computationnelle liée à ces problèmes. C'est le cas des problèmes *difficiles*. Deux sortes de problèmes sont considérés comme *difficiles* :

- Des problèmes d'optimisation combinatoire (ou discrète), pour lesquels on ne connaît pas d'algorithme exact rapide.
- Des problèmes d'optimisation à variables continues pour lesquels on ne connaît pas d'algorithme qui permet de trouver un optimum global à coup sûr en un nombre

fini de calculs.

Pour le domaine ds l'optimisation linéaire, les méthodes classiques de résolution des problèmes difficiles sont appelées méthodes *d'optimisation globale*. Ces techniques sont souvent inefficaces si la fonction objectif ne possède pas une propriété structurelle particulière telle que la convexité [BS01].

Pour des problèmes combinatoires difficiles, il existe plusieurs méthodes approchées. Ces méthodes sont souvent classées en deux catégories :

Méthodes heuristiques : elles sont développées pour résoudre un type particulier de problème. L'efficacité d'une heuristique est généralement limitée au type de problème spécifique pour lequel elle a été développée, par exemple l'algorithme de Welsh et Powell [GM75] pour la résolution des problèmes de coloration de graphe, et l'algorithme de type *First Fit* [ACG+12] dédié au problème de rangement (en anglais *Bin Packing*).

Méthodes méta-heuristiques : elles sont inspirées de processus naturels (ex. le recuit simulé inspiré du recuit d'un métal et les algorithmes génétiques). Elles peuvent s'adapter à plusieurs types de problème et donnent de bons résultats. Parmi les méta-heuristiques, on peut différencier les méta-heuristiques *de voisinage*, qui font progresser une seule solution à la fois (ex. recuit simulé, recherche tabou) et les méta-heuristiques *distribuées* qui manipulent toute une population à la fois (ex. algorithmes génétiques). En pratique, il s'ajoute au modèle des problèmes mono-objectif *difficiles* des complications telles que la considération simultanée de plusieurs fonctions objectifs (i.e. optimisation multiobjectifs). L'optimisation multiobjectif revient à chercher un ensemble de solutions respectant au mieux le compromis de l'ensemble des fonctions objectifs . Parmi les méthodes les plus utilisées dans la résolution de ce type de problèmes, nous nous intéressons aux algorithmes évolutionnistes. Les travaux de thèse de Van Veldhuizen [VV99] proposent une classification des méthodes d'optimisation évolutionniste selon les différentes préférences de compromis :

Préférence à priori : L'utilisateur définit le compromis qu'il souhaite obtenir. Cette catégorie regroupe la plupart des méthodes par agrégation où les fonctions objectifs sont fusionnées en une seule.

Préférence progressive : Le choix du compromis est affiné au fur et à mesure du déroulement de l'optimisation. C'est le cas des méthodes interactives.

Préférence à posteriori : L'utilisateur effectue un choix parmi les solutions obtenues par le processus d'optimisation. Les résultats de ces méthodes sont fournis sous forme d'une surface de solutions.

La section suivante présente une approche évolutionniste pour l'optimisation du processus de paramétrage des protocoles de gestion des données. La difficulté de résolution d'un tel problème est causée par le grand nombre de configurations possibles pour un même protocole et par la complexité computationnelle du processus d'évaluation.

6.2 Résolution du problème de configuration des protocoles de gestion des données

Nous présentons dans cette section la modélisation proposée du problème de configuration des protocoles de cohérence de donnés. Nous proposons une technique génétique pour le paramétrage des protocoles de cohérence permettant de réduire le temps de d'exécution du moteur de configuration. Nous montrons dans cette section à travers une étude de cas appliquée au protocole de glissement de données que l'utilisation de méthode génétique permet d'obtenir une bonne qualité des solutions tout en réduisant drastiquement le coût temporel par rapport à une recherche exhaustive.

6.2.1 Modélisation du problème de configuration des protocoles

Deux instances d'un protocole caractérisées par des paramètres différents induisent des comportements distincts ce qui conduit potentiellement à des performances divergentes. La prise de décision sur la configuration des protocoles de gestion des données est un problème d'optimisation où l'ensemble des solutions est constitué de toutes les instances de protocoles possibles. Une instance de protocole correspond à la combinaison d'un protocole et les valeurs de paramètres lui étant associées. Il est donc intuitif que le choix d'une solution optimale dépend principalement de la qualité du choix de ses paramètres.

Il s'agit bien d'un problème d'*optimisation discrète* (ou *combinatoire*). Le choix d'une solution revient à définir une combinaison de paramètres pour l'ensemble des protocoles choisis. Ce choix est effectué selon le profil de performance lié au contexte applicatif étudié (ex. consommation d'énergie, temps réel). Afin de répondre aux besoins utilisateurs, nous avons défini plusieurs métriques de performances comme fonctions objectifs pour le choix et l'évaluation des différentes instances de protocoles.

En plus du coût lié à l'évaluation des solutions (i.e. la valeur de la fonction objectif), il s'ajoute à la complexité de notre problème d'optimisation d'autres facteurs comme :

- La taille du problème définie par la granularité adoptée dans l'affectation des protocoles. La granularité la plus petite est d'affecter un protocole pour chaque accès à une donnée partagée et la plus grande correspond à l'affectation d'un seul protocole à toute l'application.
- Le nombre des paramètres par protocole.
- L'espace de définition de chaque paramètre de protocole.

Il est donc très coûteux de faire une recherche exacte dans l'espace des solutions possibles afin de choisir la solution optimale globale garantissant les meilleures performances.

La méthode la plus adaptée à notre problème de configuration de protocoles est l'utilisation de méta-heuristiques distribuées telles les algorithmes génétiques car elles permettent d'explorer au mieux l'espace des solutions possibles.

6.2.2 Utilisation d'une approche génétique pour optimiser le processus de paramétrage des protocoles

Il existe de nombreuses heuristiques et métaheuristiques de résolution des problèmes d'optimisation multiobjectif tels que le *recuit simulé*, la *recherche tabou* ou encore les technique bio-inspirées tels que les *réseaux de neurones* et les *algorithmes évolutionnaires*. Ces derniers sont basés sur la théorie d'évolution de Darwin (i.e. ce sont les populations les plus adaptées à leur milieu qui survivent). De précédents travaux [Gol89] ont démontré les différents avantages d'utilisation des *algorithmes évolutionnaires* :

- Ils permettent de trouver un ensemble de solutions optimales et non pas une seule [Deb01].
- L'espace des solutions est plus largement exploré avec moins d'appels à la fonction d'évaluation comparé à d'autres méthodes de recherche locales (ex. recuit simulé, recherche tabou) [AGKL03].
- La résolution du problème d'optimisation est indépendante de la solution initiale et ne nécessite pas une optimisation locale (i.e. recherche dans le voisinage) [AGKL03].

Plusieurs variantes des *algorithmes évolutionnaires* ont été développées afin de résoudre les problèmes d'optimisation multiobjectif [CVVL02, Deb01]. Parmi les algorithmes les plus utilisés dans la littérature nous trouvons les versions récentes des algorithmes : *Non-dominated Sorting Genetic Algorithm* (NSGA-II) proposé par Deb et al. [DPAM02], *Strenght Pareto Evolutionnary Algorithm* (SPEA-II) développé par Zitzler et al. [ZLT$^+$01] et *Pareto Archived Evolution Strategy* (PAES) présenté dans les travaux de Knowles et al. [KC99].

Dans la majorité des problèmes d'optimisation multiobjectifs, le temps d'évaluation d'une solution peut prendre plusieurs minutes voire plusieurs heures dans des cas complexes. La complexité du processus d'évaluation rend le coût de résolution prohibitif ce qui mène à la dégradation de la qualité de la solution ou bien un temps de résolution très grand.

Il est donc très important de prendre en compte cette contrainte dans le choix de notre système de décision qui est caractérisé par un coût élevé d'évaluation de chaque instance de protocole. Nous utilisons dans notre brique de paramétrage un algorithme génétique rapide appelé *Fast Pareto Genetic Algorithm*(FastPGA) proposé par Eskandari et al. [EGL07]. Il permet de trouver une solution *Pareto* optimale dans un temps acceptable (section 6.3).

Nous décrivons dans le paragraphe 6.2.3 l'algorithme *FastPGA*, ensuite une étude de cas dans la section 6.3 à travers une application sur le choix des paramètres du protocole de glissement de données pour différentes applications.

6.2.3 Algorithme d'optimisation multiobjectif FastPGA

L'algorithme d'optimisation *FastPGA* est basé sur une nouvelle stratégie de classement des solutions. Chaque solution est représentée par un vecteur $x = (x_1, .., x_2)$ où la

variable de décision x_i est un nombre réel compris entre une valeur inférieure a_i et une valeur supérieure b_i. La taille du vecteur x est égale au nombre de variables de décision du problème étudié.

En pratique, l'algorithme du *FastPGA* est caractérisé par l'introduction de deux nouveaux opérateurs de recherche afin de réduire le temps d'exécution :

- Opérateur de sélection élitiste : assurant la propagation du *front de Pareto* optimal (Définition 20).
- Opérateur de régulation : permettant d'adapter dynamiquement la taille de la population à chaque itération de l'algorithme génétique.

L'algorithme 2 décrit les différentes étapes du processus d'optimisation *FastPGA*.

Algorithme 2 Pseudo-code de l'algorithme *FastPGA*

Require: Taille de la population et fonctions objectifs.
1: $t := 0$
2: Créer une population initiale aléatoire P_t
3: Évaluer(P_t)
4: **while** critère d'arrêt non atteint **do**
5: $t := t + 1$
6: ▷ sélectionner des paires de solutions pour la reproduction
7: $\mathrm{P'}_t$:=sélectionner(P_{t-1})
8: O_t := croisement($\mathrm{P'}_t$)
9: O_t := mutation(O_t)
10: évaluer(O_t)
11: ▷ former une nouvelle population composée de la population courante et la population issue du croisement et de mutation
12: $\mathrm{CP}_t := \mathrm{P}_{t-1} \cup \mathrm{O}_t$
13: classement(CP_t)
14: ▷ réguler la nouvelle population en fonction du nombre des solutions non dominées
15: régularisation(CP_t)
16: ▷ générer une nouvelle population en éliminant les solutions dominées
17: P_t := générer(CP_t)
18: **end while**

La première étape de l'algorithme consiste à paramétrer la population initiale avec des valeurs aléatoires de l'espace des solutions. Cette solution aléatoire est ensuite évaluée selon les fonctions objectifs définies au début de l'algorithme.

Dans chaque itération, une nouvelle population (CP_t) est constituée de l'union des solutions issues des opérations de croisement et de mutation O_t et de la population initiale (P_{t-1}). Les individus (éventuelles solutions) de la population composée sont classés en deux catégories (appelées *rangs*) selon leur *dominance*. La notion de *dominance* est utilisée dans l'optimisation multi-objectif pour éliminer les solutions les moins

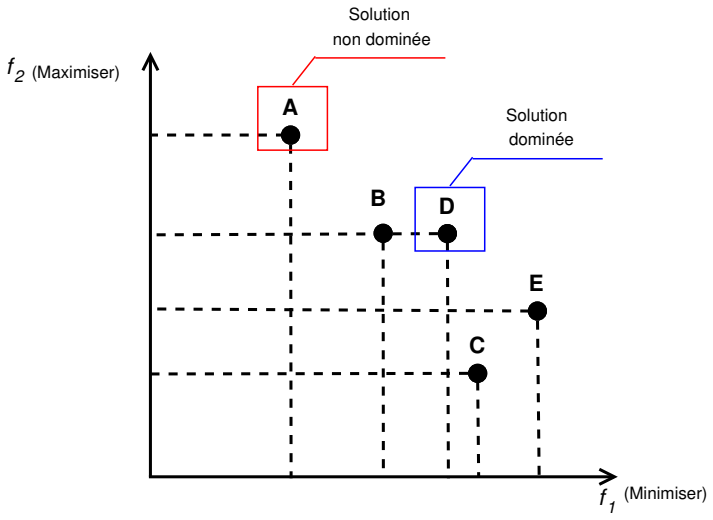


FIGURE 6.2 – Représentation des solutions dans le plan (f_1, f_2)

optimales. Pour qu'une solution soit intéressante, il faut qu'il existe une relation de *dominance* entre cette solution et les autres solutions. Cette relation est définie comme suit :

Définition 19 (Dominance)**.** On dit que la solution $\vec{x}$ domine la solution $\vec{y}$ si :

- $\vec{x}$ est au moins aussi bon que $\vec{y}$ par rapport à tous les objectifs, et,
- $\vec{x}$ est strictement meilleur que $\vec{y}$ dans au moins un objectif

Notons $\vec{x} > \vec{y}$ la relation de dominance entre les deux vecteurs.

La *dominance* permet de définir le concept de solutions optimales dans le sens de *Pareto* (ou solutions *non dominées*).

Définition 20 (Solution optimale de Pareto)**.** Une solution $\vec{x}$ est optimale globalement au sens de *Pareto* s'il n'existe pas de solution $\vec{y}$ tel que $\vec{y}$ domine $\vec{x}$.

Soient A, B, C, D et E des solutions d'un problème d'optimisation à deux fonctions objectifs f_1 et f_2. Supposons que la solution optimale doit minimiser f_1 et maximiser f_2. La figure 6.2 représente les solutions dans le plan (f_1, f_2). La solution A est une solution optimale (i.e. dominante) car aucune autre solution ne la domine. La solution D est une solution dominée par la solution B et la solution A.

Le classement des solutions en deux rangs a comme objectif de définir l'adéquation de chaque solution avec les objectifs définis. Ceci permet de préserver les meilleures solutions (premier rang) de la population précédente dans les futures générations.

Toutes les solutions *non dominées* sont identifiées comme solutions du premier rang. Le degré d'adaptation des solutions *non dominées* aux objectifs définis est calculé en

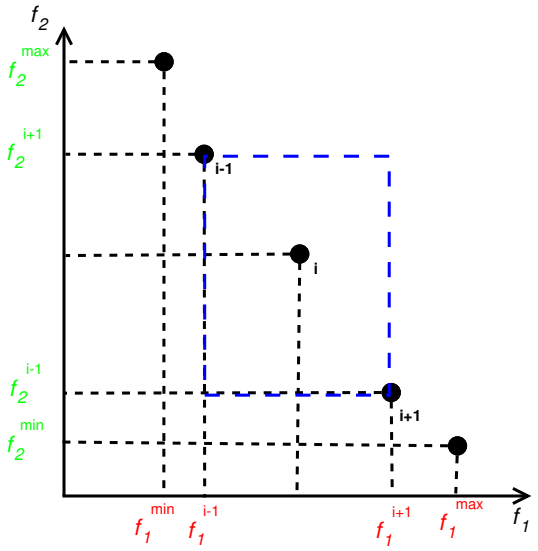


FIGURE 6.3 – La distance d'encombrement pour une solution i pour un objectif est égale à la différence entre les fonctions objectifs des solutions adjacentes $i+1$ et $i-1$.

comparant les solutions entre elles et en affectant une valeur d'adaptation (i.e. en anglais *fitness*) à chaque solution. La valeur d'adaptation d'une solution utilise la notion de *distance d'encombrement* (en anglais *crowding distance*) proposée par Deb. et al. [DPAM02]. Cette technique permet de maintenir la diversité au sein de la population à un coût réduit.

La *distance d'encombrement* affectée à une solution correspond à son positionnement par rapport aux autres solutions (figure 6.3). Deux types de distance sont définis : *distance d'encombrement* et la *distance d'encombrement globale* qui est calculée à partir de la première. La définition de la *distance d'encombrement* normalisée est la suivante :

Définition 21 (Distance d'encombrement normalisée)**.** Soit P l'ensemble des solutions étudiées, N la taille de cet ensemble et f la fonction objectif considérée.
Si I est la population triée dans un ordre croissant par rapport à la fonction objectif f, alors :

- Les solutions sur les frontières prennent une distance infinie. :
  $I[1]_{distance}$=$I[N]_{distance}$ =inf
- Pour toute autre solution intermédiaire :

$$I[i]_{distance} = \frac{f(I[i+1]) - f(I[i-1])}{f_{max} - f_{min}} \tag{6.1}$$

La valeur d'adaptation d'une solution est égale à la *distance d'encombrement globale* correspondant à cette solution. En pratique, le calcul de la *distance d'encombrement globale* nécessite de trier dans l'ordre croissant toutes les solutions selon chacune des

fonctions objectifs et ensuite de calculer la *distance d'encombrement individuelle* pour toutes les fonctions objectifs. La distance globale est donc définie comme suit :

Définition 22 (distance d'encombrement globale (en anglais *crowding Distance*))**.** La *distance d'encombrement* globale des solutions intermédiaires est la somme de toutes les distances individuelles calculées pour chaque objectif.

Notons que plus la valeur de *fitness* est grande, plus la solution est éloignée des autres solutions non dominées voisines. Les solutions avec une grande valeur de *fitness* sont donc meilleures pour les prochaines itérations de l'algorithme génétique.

Toutes les solutions dominées sont considérées dans le deuxième rang. Une nouvelle fonction de *fitness* est utilisée pour celles-ci. Chaque solution dominée est comparée avec toutes les autres, et sa valeur de *fitness* correspond au nombre des solutions qui la dominent. L'idée ici est basée sur la notion de *force* utilisée dans *SPEA* [ZT99] avec une généralisation prenant en compte en plus des solutions dominantes, celles dominées par la solution étudiée. La *force* d'une solution dominée est calculée comme suit :

Définition 23 (Force)**.** Soient CP_t la population composée et x_i une solution dominée dans cette population.

$$F(x_i) = \sum_{x_i > x_j} S(x_j) - \sum_{x_j > x_i} S(x_j), \forall x_j \in CP_t \wedge i \neq j \tag{6.2}$$

Avec :

$$S(x_i) = \mid \{x_j | \forall x_j \in CP_t \wedge x_i > x_j \wedge i \neq j\} \mid \tag{6.3}$$

Après le calcul des valeurs de *fitness* de toutes les solutions, celles-ci sont comparées deux à deux. Trois scénarios de comparaison se présentent : (i) Si les deux solutions appartiennent à des rangs différents et dans ce cas la solution ayant le meilleur rang est privilégiée, (ii) Si les deux solutions appartiennent au même rang mais avec des valeurs de *fitness* différentes, la solution avec la plus grande valeur est préservée, (iii) Si les deux solutions qui ont le même rang ont la même valeur de *fitness*, une solution est choisie de façon aléatoire.

Toutes les solutions de la population précédente sont incluses dans la population composée afin d'assurer le transfert des solutions non dominées dans les futures générations (i.e. stratégie élitiste). Le nombre des solutions non dominées augmente généralement d'une itération à autre. Nous maintenons à chaque itération les M meilleures solutions pour former la population suivante. Afin de mieux s'adapter au problème étudié, la taille de la population est choisie dans notre cas selon le nombre des paramètres à configurer et la taille de la trace mémoire en entrée (voir section suivante 6.3).

6.2.4 Définition des opérateurs utilisés

L'algorithme génétique *FastPGA* (section 6.2.3) est constitué de plusieurs étapes. Cette section décrit les opérateurs utilisés dans chaque étape de l'algorithme. Les étapes élémentaires du *FastPGA* sont :

1. Initialisation de toutes les variables de décision.
2. Définition de la population initiale avec des valeurs aléatoires.
3. Sélection des paires de solutions (i.e. parents) de la population précédente en utilisant l'opérateur *Tournoi binaire*.
4. Croisement et mutation des solutions sélectionnées afin de constituer la nouvelle génération.
5. Évaluation des nouvelles solutions selon les fonctions objectifs définies.
6. Combinaison des solutions générées avec la population précédente pour former la population composée.
7. Classement de l'ensemble des solutions en se basant sur leur valeur de *fitness*.
8. Régulation de la taille de la population en fonction du nombre de solutions *non-dominées* et création de la nouvelle génération en éliminant les solutions dominées inférieures.
9. Terminaison de l'algorithme si la condition d'arrêt est atteinte.

Nous décrivons dans ce qui suit avec plus de détails les opérations utilisés pour chaque étape :

Sélection par tournoi binaire

L'étape de sélection a comme objectif de définir à chaque génération un ensemble de paires de solutions *parents* qui serviront, à la prochaine étape, à créér de nouvelles solutions *enfants*. Les paires de solutions sont sélectionnées parmi les meilleurs individus de la population (i.e. stratégie élitiste). Nous utilisons dans cette étude la *sélection par tournoi binaire*, un cas particulier de la *sélection par tournoi* (figure 6.4). Elle est composée de plusieurs étapes de sélection appelée *tournoi*. Un tournoi consiste à tirer aléatoirement I individus dans la population, puis à choisir le meilleur individu du groupe en fonction de sa valeur de *fitness* et de la probabilité de sélection donnée. Ce processus est répété k fois jusqu'à obtention du nombre d'individus requis. Dans le cas d'un *tournoi binaire* la taille des groupes sélectionnés dans chaque tournoi est fixée à 2 ($I = 2$). Pour une sélection déterministe, la probabilité de conserver un individu à chaque tournoi doit être égale à 1 (i.e. l'individu avec le meilleur fitness est choisi à chaque fois). La pression de sélection est ajustée grâce au nombre de tournois. Une grande valeur de ce paramètre permet de diminuer la chance de survie des individus faibles.

Cette méthode de sélection garantit donc l'élimination des individus faibles d'une population (quand la probabilité de sélection s'approche de 1). Les individus sélectionnés (parents) sont ensuite recombinés pour générer de nouveaux individus (enfants). Le paragraphe suivant décrit la technique de croisement utilisée dans l'algorithme *FastPGA*.

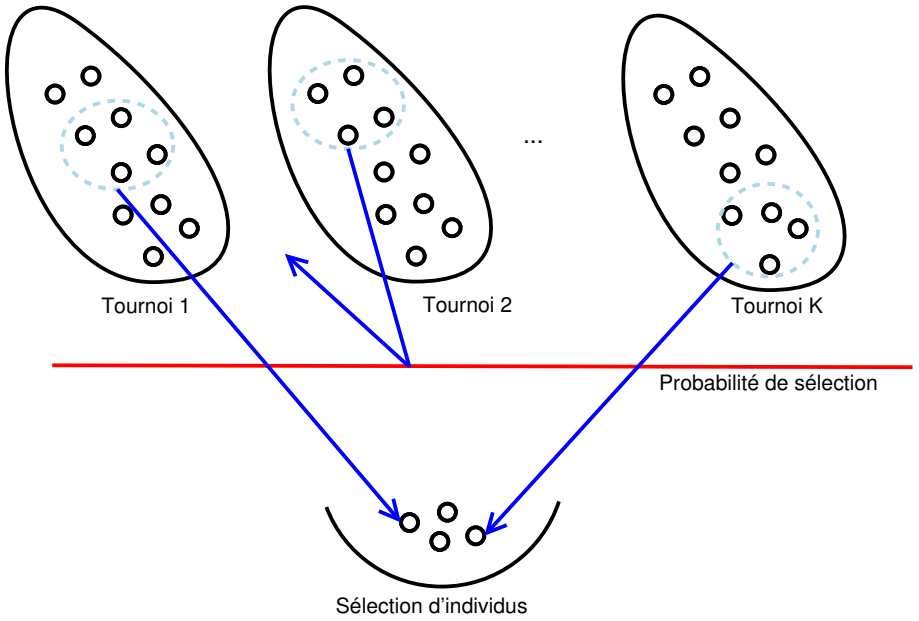


FIGURE 6.4 – Sélection par tournoi. À chaque tournoi un groupe d'individus est choisi aléatoirement. Seule le meilleur individu est retenu à l'issue de chaque tournoi. La sélection finale est composée d'au plus k individus selon la valeur de probabilité définie.

Croisement binaire simulé

Pour mieux explorer l'espace des solutions, il est indispensable de combiner/transformer les individus sélectionnés (étape précédente). La première phase de transformation de ces individus est de les recombiner par paires (i.e. *croisement*). Il existe différentes techniques de croisement dans la littérature. Nous utilisons dans ce travail le mécanisme de *croisement binaire simulé* [DA94] (Simulated Binary Crossover (*SBX*)).

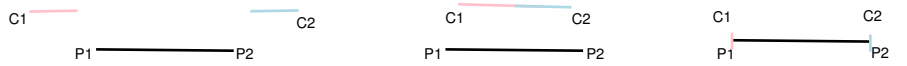
La technique de croisement *SBX* consiste à générer à partir de deux parents p_1 et p_2 deux enfants c_1 et c_2. La relation qui décrit cette reproduction est définie comme suit :

$$\begin{cases} c_1 = 0.5[(1+\beta) * p_1 + (1-\beta) * p_2] \\ c_2 = 0.5[(1-\beta) * p_1 + (1+\beta) * p_2] \end{cases} \tag{6.4}$$

où β est un facteur de dispersion des enfants par rapport aux parents définie par la formule :

$$\beta = \begin{cases} (2u)^{(\frac{1}{\eta+1})} & \text{si u} < 0.5 \\ (\frac{1}{2*(1-u)})^{(\frac{1}{\eta+1})} & \text{sinon} \end{cases} \tag{6.5}$$

où u est une valeur aléatoire uniformément définie sur l'intervalle $[0, 1]$ et η un paramètre réel non-négatif caractérisant la forme de la distribution des enfants par rapport aux parents. Une grande valeur de η augmente la probabilité d'avoir des enfants



(a) Croisement avec expansion (b) Croisement avec contraction (c) Croisement stationnaire

FIGURE 6.5 – Les 3 modes de croisement selon la valeur du facteur de dispersion β. La figure 6.5a correspond à un croisement avec une dispersion supérieure à 1, ce qui signifie que les enfants sont loin des parents. La figure 6.5b est un croisement avec contraction où les enfants sont proches des parents. Enfin, lorsque les enfants sont identiques aux parents, il s'agit d'un croisement stationnaire.

proches des parents. La valeur de dispersion garantit donc de mieux parcourir l'espace des solutions. On distingue trois modes de croisement selon la valeur de dispersion :

- Croisement avec expansion ($\beta > 1$) : Les solutions enfants sont loin des solutions parents (figure 6.5a).
- Croisement avec contraction ($\beta < 1$) : Les solutions enfants sont très proches des solutions parents (figure 6.5b).
- Croisement stationnaire ($\beta = 1$) : Les enfants sont identiques aux parents (figure 6.5c).

Mutation

Après le croisement, l'opérateur de mutation modifie aléatoirement les valeurs des individus enfants avec une faible probabilité. Cette modification permet de changer les caractéristiques des individus tout en gardant le même format, ce qui peut engendrer des résultats d'évaluation totalement différents. Le rôle de cette étape de l'algorithme est d'introduire du bruit afin de maintenir la diversité de la population entre les différents individus parcourus. Elle permet de garantir de mieux explorer l'espace de recherche et d'éviter la convergence vers des optimums locaux. La valeur de la probabilité de mutation doit rester faible pour ne pas perdre de l'information utile et détériorer la qualité de la population. Une grande valeur pour cette probabilité reviendrait à faire une recherche aléatoire dans l'espace des solutions. L'opérateur de mutation utilisé dans cette étude est de type *polynomiale binaire*. La mutation *polynomiale binaire*, comme le croisement *SBX*, utilise une distribution de probabilité polynomiale (généralisation de la loi binomiale). La création d'une valeur mutée consiste à générer de façon aléatoire une valeur réelle positive u dans l'intervalle $[0, 1]$. Soit η un nombre réel positif (voir croisement *SBX*), l'équation 6.6 permet de calculer le facteur de perturbation δ correspondant à u :

$$\delta = \begin{cases} (2u)^{\frac{1}{\eta+1}} - 1, & \text{si u} < 0.5 \\ 1 - [2(1-u)]^{\frac{1}{\eta+1}} & \text{sinon} \end{cases} \tag{6.6}$$

Une grande valeur de η produit des mutants ressemblants au parents. La valeur mutée est calculée avec la probabilité P définie en fonction du facteur de perturbation par l'équation :

$$P(\delta) = 0.5(n+1)(1-|\delta|)^n \quad \text{avec -1} < \delta < 1 \tag{6.7}$$

La valeur mutée est calculée comme suit :

$$c = p + \delta \triangle_{max} \tag{6.8}$$

où $\triangle_{max}$ est une valeur fixe limitant la perturbation maximale autorisée. c et p représentent respectivement la valeur mutante et celle mutée.

Les autres étapes de l'algorithme génétique sont moins complexes et ne nécessitent pas pas plus d'explications.

La section suivante est une étude de cas du protocole de glissement de données pour le choix du rayon de glissement associé à chaque instance du protocole.

6.3 Étude de cas : paramétrage du protocole de glissement de données

Le protocole de glissement de données (section 2.4) est caractérisé par le rayon de migration des données qui limite la distance (en nombre de pas réseau) entre les cœurs propriétaires et les cœurs coopératifs accueillants les données. Le choix de ce rayon est important pour mieux adapter le comportement du protocole aux besoins de l'application. Nous étudions dans cette section la problématique du choix du rayon de glissement en utilisant l'algorithme évolutionniste *FastPGA*.

6.3.1 Fonction objectifs

Les fonctions objectifs définies dans cette étude sont : le nombre d'accès réussi au cache (i.e. *Cache Hit*) et la distance cumulative entre les cœurs propriétaires et les cœurs accueillant les données. Ces deux métriques permettent d'évaluer les performances du protocole de gestion des données en fonction du rayon de glissement associé à chaque donnée.

La valeur du rayon de glissement est limitée par la taille de la puce (voir figure 6.6). Elle est comprise entre une valeur minimale égale à 1 pour une coopération sur le voisinage proche et la distance de *Manhattan* entre les deux cœurs les plus éloignés de la puce : $2 * \sqrt{Taille\_Puce} - 2$, ce qui correspond à une coopération étendue sur toute la puce.

L'étude expérimentale consiste à analyser, en premier lieu, les performances du processus de décision proposé en terme du temps d'exécution. La suite de l'étude porte sur la qualité des solutions obtenues à l'aide de ce processus d'optimisation.

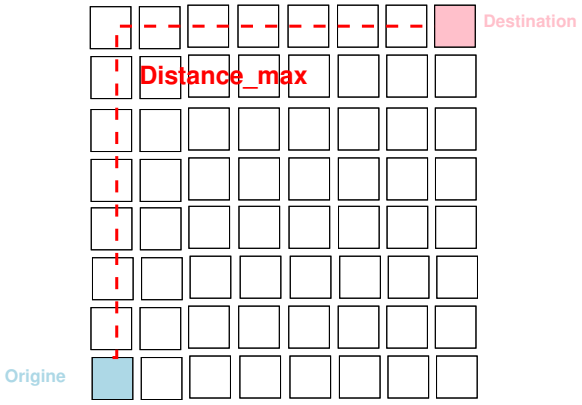


FIGURE 6.6 – Le rayon maximal est égal à la distance de *Manhattan* maximale entre les deux cœurs les plus éloignés de la puce dans le contexte d'un *NoC* maille 2D.

6.3.2 Algorithme génétique versus méthode exhaustive

Nous étudions dans cette section le cas de la configuration du protocole de glissement de données pour différentes traces mémoire.

L'intérêt de l'utilisation d'un algorithme génétique pour la configuration des protocoles de cohérence est principalement de réduire le temps d'exécution qui est exponentiel dans le cas d'une recherche exhaustive. Ce temps dépend de la taille de l'espace des solutions qui est défini en fonction de la trace d'accès mémoire et de l'intervalle de définition de chaque paramètre du protocole de cohérence étudié. Dans notre étude nous considérons que pour chaque accès à une donnée est affectée une instance de protocole de glissement de données. Ce dernier peut être caractérisé par la valeur du rayon de glissement affectée à chaque instance de protocole. Nous comparons en premier lieu le coût d'une recherche exhaustive d'une solution optimale au coût d'une recherche optimisée par l'algorithme génétique utilisé. Une recherche exhaustive revient à évaluer tout l'espace des solutions en vue de n'en garder que la meilleure. Une solution est une combinaison de valeurs des paramètres affectée à l'ensemble de la trace mémoire. Rappelons qu'un appel de la fonction d'évaluation revient à simuler le trafic réseau à l'aide du *CacheValidator* (section 3.1.1) ce qui est très coûteux. Lorsque le nombre d'accès mémoire -et donc le nombre d'instances de protocoles- est très important, le temps dépensé dans l'exploration de la totalité de l'espace des solutions devient très grand. Nous avons utilisé un serveur *NUMA* de *Dell* constitué de quatre processeurs *AMD Opteron 6172 cadencé à 2.1 GHz*, chacun doté de 12 cœurs et de *(64 Go de RAM)* . Nous utilisons quatre cœurs dans notre expérimentation. L'algorithme exhaustif consomme beaucoup de mémoire ce qui limite le nombre d'accès à 20 accès afin de rester dans des temps d'expérimentation raisonnables.

Les objectifs d'optimisation consiste à :

- réduire le mouvement des données sur la puce, ce qui revient à minimiser la distance cumulative de migration des données

Taille de la trace mémoire	Recherche exhaustive (s)	algorithme génétique (s)
6	24	53
10	240	55
15	10312	54
20	63976	54

TABLE 6.1 – Comparaison entre les coûts temporels des deux algorithmes de recherche exhaustif et génétique en fonction de la taille des traces mémoires.

- augmenter les performances des caches, ce qui revient à minimiser le nombre de défauts de cache

Le tableau 6.1 présente les temps de recherche en utilisant la méthode exhaustive et la méthode génétique en fonction des tailles des traces mémoire utilisées.

Nous constatons que pour la recherche exhaustive le coût temporel augmente avec le nombre d'accès aux données (de manière exponentielle), ce qui est expliqué par l'augmentation de la taille de l'espace des solutions à évaluer (nombre d'appels au *CacheValidator*) d'un coté et et de la taille de chaque solution de l'autre.

Cette augmentation est moindre dans le cas de la recherche génétique, car nous avons fixé le nombre d'itérations de l'algorithme génétique à 10, ce qui laisse le nombre d'appels de la fonction d'évaluation fixe. Dans ce cas, nous ne voyons pas vraiment l'impact de la variation de la taille des traces mémoires sur le temps de recherche.

La comparaison entre les coûts temporels de l'algorithme exhaustif et de l'algorithme génétique montre que le premier est beaucoup plus coûteux. Ceci est expliqué par le fait que la solution exhaustive nécessite de parcourir la totalité des configurations possibles ce qui génère un grand nombre d'appels de la fonction d'évaluation. Pour une trace mémoire constituée de n accès, et un rayon de glissement défini sur un intervalle $[1, m]$, la taille de l'espace des solutions est égale à m^n. Dans notre cas, le rayon de glissement est limité par la taille de la puce $2*\sqrt{Taille\_Puce}-2$. Or, pour des raisons de simplicité et pour éviter de saturer la mémoire nous limitons le rayon de glissement à deux valeurs (1 et 2).

Cependant, le nombre des solutions évaluées par l'algorithme génétique est égale au produit de la taille de la population par itération et le nombre d'itérations globales(équation 6.9.

$$Nombre\_solutions = Taille\_population * Nombre\_itérations \tag{6.9}$$

Pour ces expérimentations, la taille de la population et le nombre d'itérations sont fixés à 10, ce qui donne un espace de 100 solutions à explorer. Ceci explique la stabilité du coût temporel de la solution génétique. Pour la trace mémoire à 6 accès, il est logique que le temps de recherche exhaustive soit inférieur à celui obtenu par la recherche génétique vu que la taille de l'espace des solutions possibles explorées par la méthode exhaustive (= 36) est inférieure au nombre de solutions explorées par l'algorithme génétique (= 100).

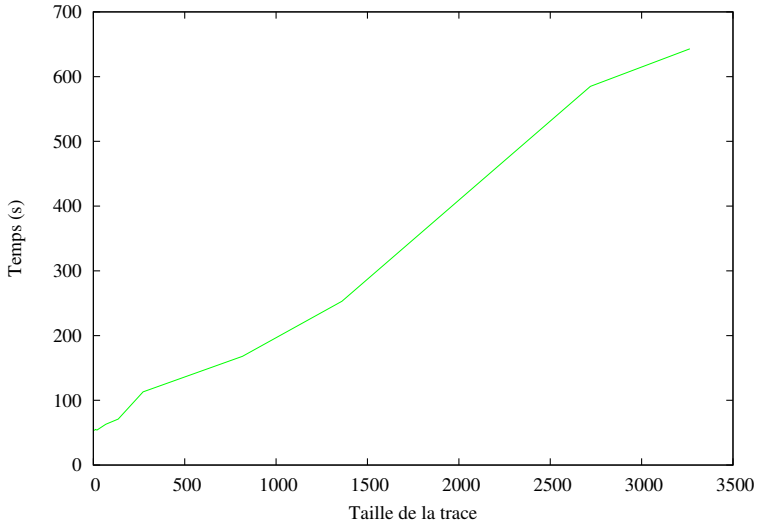


FIGURE 6.7 – La variation des temps de recherche d'une solution à l'aide de l'algorithme génétique proposé en fonction de la charge mémoire.

La figure 6.7 montre la variation du temps de recherche en utilisant l'algorithme génétique en fonction de la taille de la trace mémoire à configurer pour un espace de solutions fixe (100 solutions). La courbe montre que le temps de recherche est proportionnel à la taille de la trace mémoire démontrant ainsi l'impact de la charge mémoire sur l'efficacité du processus de décision. Nous rappelons que le coût temporel est fortement lié au temps de simulation de la trace dans l'outil d'évaluation (le *CacheValidator* dans ce cas).

L'utilisation d'un algorithme génétique permet de parcourir un plus grand nombre de solutions en minimisant le temps de recherche à travers un nombre d'itérations fixe. Un plus grand nombre d'itérations permet donc une plus large exploration de l'espace des solutions possibles et éventuellement une amélioration de la qualité des solutions obtenues. La taille de la population permet également d'améliorer les performances de l'algorithme génétique en augmentant le nombre de solutions parcourues à chaque itération.

La qualité des solutions dépend de leur optimisation des fonctions objectifs définies. Par exemple, une bonne qualité de solution dans le cas d'un protocole de glissement de données correspond au meilleur compromis entre la distance de migration et le nombre de défauts de caches correspondants.

6.3.3 Configuration de l'algorithme génétique FastPGA

Le processus de décision est basé sur l'algorithme génétique *FastPGA*. Il permet de rechercher une solution de *Pareto* dans l'espace des solutions à l'aide d'une méthode génétique itérative. Les méthodes génétiques nécessitent généralement une configuration efficace des opérateurs de l'algorithme (croisement, mutation). La condition d'arrêt est

également importante car elle définit le degré d'exploration de l'espace des solutions. La condition d'arrêt utilisée dans notre étude est le nombre d'itérations.

Pour améliorer la qualité de la solution, il faut garantir la diversification des solutions parcourues. Les probabilités de croisement et de mutation permettent de contrôler le degré de modification des individus de chaque génération. Les valeurs de ces probabilités jouent un rôle très important dans la qualité des solutions obtenues. Nous étudions dans cette section la variation de la probabilité de croisement et son impact sur les solutions générées par l'algorithme *FastPGA*. Le *facteur de diversité* (définition 24) est une métrique qui nous permet de mesurer la diversité entre les individus générés à chaque itération de l'algorithme génétique.

Définition 24 (Facteur de diversité)**.** Le *facteur de diversité* est défini comme le nombre de solutions différentes dans une génération d'individus. Deux solutions sont différentes si leurs valeurs de fonctions objectifs ne sont pas identiques.

Nous définissons la probabilité dynamique en fonction du *facteur de diversité* dans la génération courante.

$$Probabilité = \frac{P_0 * (\text{Taille\_population - 1 - Diversité})}{\text{Taille\_population - 1}} \tag{6.10}$$

P_0 est la valeur de la probabilité initiale.

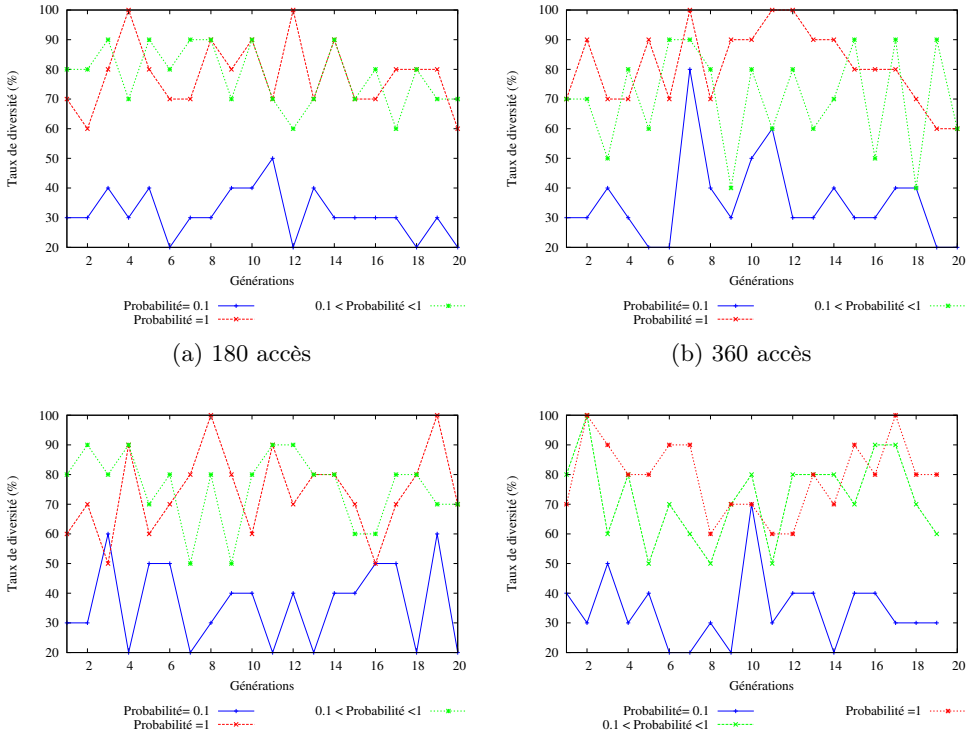
Lorsque la diversité est grande la probabilité diminue et vice versa. L'objectif de l'utilisation d'une probabilité dynamique est d'adapter la valeur de la probabilité de croisement à l'évolution de la population.

La probabilité de mutation est fixée à 0.05. La figure 6.8 montre la variation du facteur de diversité sur 10 générations pour deux valeurs de probabilité statique 0.1 et 1 et une probabilité dynamique variant entre ces deux valeurs.

Le taux de diversité calculé pour les différentes valeurs de probabilité de croisement reflète le recouvrement de l'espace des solutions. Nous constatons dans la figure 6.8a, figure 6.8b et la figure 6.8c que la diversité obtenue par la méthode à probabilité dynamique est comprise entre 40% et 90%, tandis que la probabilité 1 donne une grande diversité entre 50% et 100% et la probabilité faible 0.1 donne une diversité inférieure à 40%.

La comparaison entre les solutions obtenues avec les différentes valeurs de probabilité montre qu'un taux élevé de croisement donne une meilleure qualité de solutions (6.9). Une probabilité de croisement égale à 1 permet d'obtenir une meilleure qualité de solution, mais elle peut engendrer la perte d'information utile. Ce risque ne se représente pas souvent dans notre cas d'étude mais il est existant. Il est donc plus recommandé dans l'état de l'art d'utiliser une probabilité inférieure à 1. Nous proposons dans ces travaux une configuration dynamique de la valeur de probabilité de croisement. L'idée consiste à adapter dynamiquement la valeur de la probabilité en fonction du taux de diversité de la population.

La variation dynamique de la probabilité a pour objectif de garantir une bonne exploration de l'espace des solutions en s'adaptant dynamiquement à l'évolution de la population.



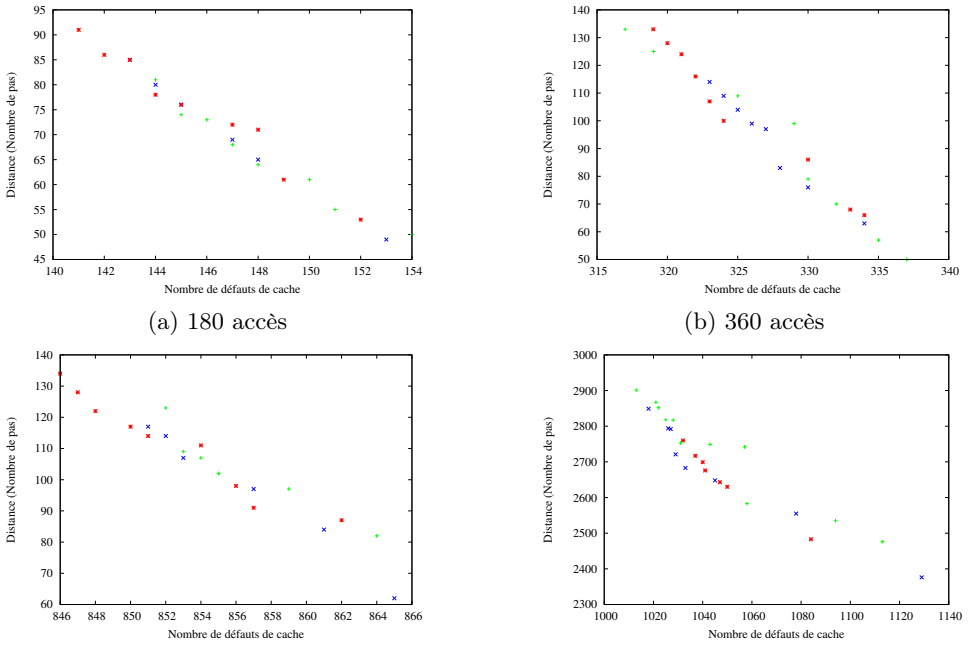
(a) 180 accès

(b) 360 accès

(c) 900 accès

(d) 2080 accès

FIGURE 6.8 – La variation du taux de diversité sur 20 générations pour différentes valeurs de probabilités de croisement. La probabilité maximale et celle dynamique génèrent une meilleure diversité entre les individus explorés.



(a) 180 accès

(b) 360 accès

(c) 900 accès

(d) 2080 accès

FIGURE 6.9 – Comparaison des performances entre les solutions obtenues avec des valeurs de probabilité de croisement différentes. Les points rouges correspondent à la probabilité 1, les points bleus correspondent à la probabilité variable [0.1, 1] et les point verts correspondent à la probabilité minimale 0.1. Les résultats présentés correspondent à des charges mémoire différentes.

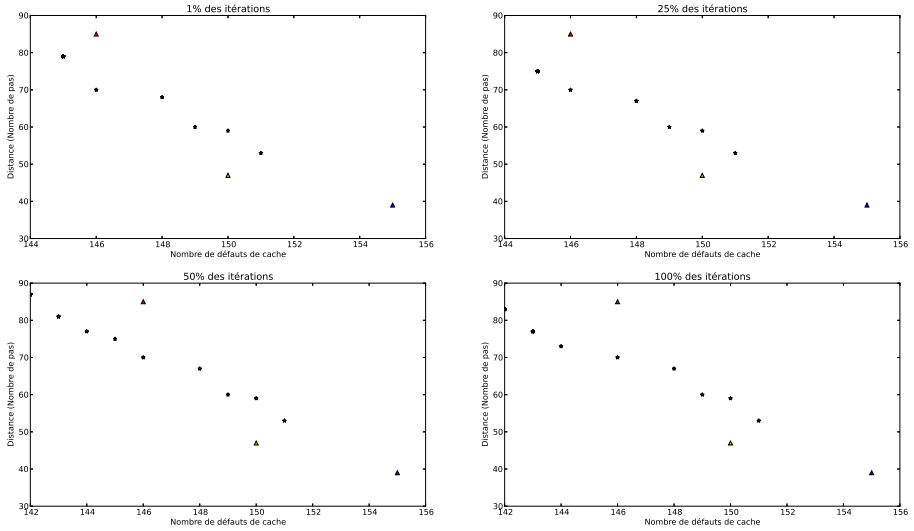


FIGURE 6.10 – Évolution des solutions de *Pareto* en fonction du nombre d'itérations (180 accès). Le point rouge correspond à la solution correspondant au rayon maximal, le point jaune au rayon moyen et le point bleu à la valeur minimale.

Un autre élément de configuration qui peut influencer sur la qualité des solutions obtenues est le nombre d'itérations. Elle permet de limiter le nombre de générations parcourues par le processus génétique.

Les figures 6.10, 6.11 et 6.12 montrent l'évolution des solutions obtenues par la méthode génétique sur les différentes étapes de l'algorithme : à 1%, à 50%, à 80% et à 100% des itérations.

Nous constatons que plus la trace mémoire est grande (plus grand nombre d'accès) plus l'impact du nombre d'itérations est visible. En effet, ce constat est expliqué par le fait que la taille de la trace mémoire augmente drastiquement la taille de l'espace des solutions. L'exploration de ce dernier à l'aide de la méthode génétique proposée nécessite un plus grand nombre d'itérations ce qui permet d'améliorer la qualité des solutions obtenues. De façon générale, l'augmentation du nombre de générations de l'algorithme génétique permet de couvrir un plus grand nombre de solutions possibles. Or, ceci peut avoir un coût de calcul très élevé. La question est donc de choisir un nombre d'itérations bien adapté à la taille de la trace pour obtenir de meilleures performances.

6.3.4 Analyse de la qualité des solutions obtenues

La section précédente montre que le processus d'optimisation à l'aide de l'algorithme génétique *FastPGA* permet de réduire drastiquement le temps de recherche des solutions. Cette section est dédiée à l'étude de la qualité des solutions obtenues (i.e. solution *Pareto*) par la méthode génétique. Les traces mémoire étudiées dans cette expérimenta-

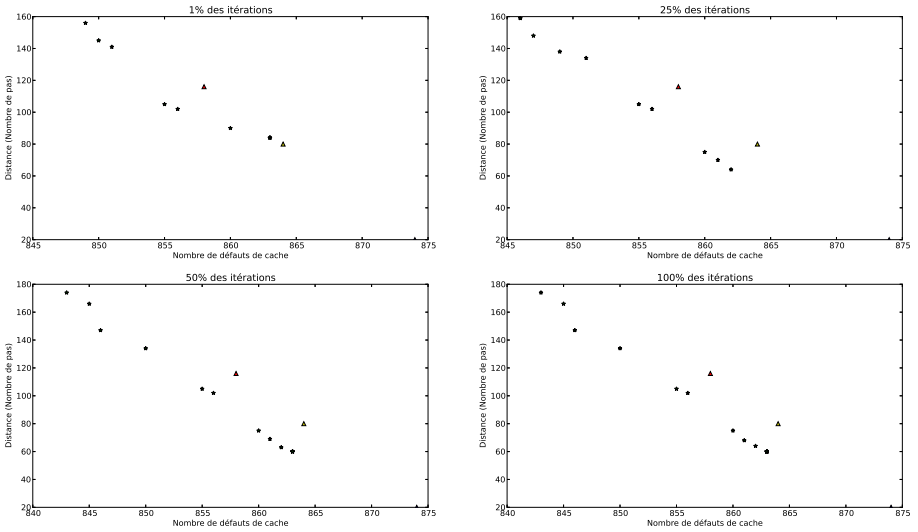


FIGURE 6.11 – Évolution des solutions de *Pareto* en fonction du nombre d'itérations à 1%, 25%, 50%, 100% (360 accès).

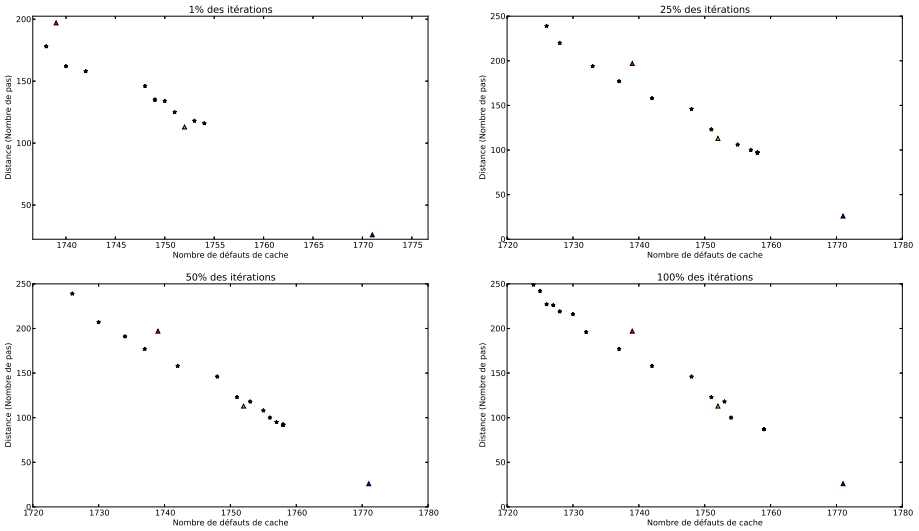


FIGURE 6.12 – Évolution des solutions de *Pareto* en fonction du nombre d'itérations (1800 accès).

Paramètres	Configurations
Taille de la population initiale	10
Nombre d'itérations	100
Probabilité de croisement	[0.6,1]
Probabilité de mutation	0.05
Opérateur de croisement	croisement binaire simulé, η_c=15
Opérateur de mutation	mutation polynomiale η_m=20
Opérateur de sélection	Tournoi binaire

TABLE 6.2 – Configuration de l'algorithme génétique *FastPGA*.

tion sont des traces synthétiques qui représentent différentes charges mémoire avec une répartition aléatoire des accès sur la puce.

Le tableau 6.2 décrit la configuration de l'algorithme génétique utilisée dans cette étude.

La figure 6.13 montre le positionnement en terme de performance (distance cumulative, nombre de défauts de cache) des solutions *Pareto* obtenues par la méthode génétique par rapport à des solutions aléatoires correspondant respectivement à des valeurs de rayons fixes : 1 (rayon min), 7 (rayon moyen), 14 (rayon maximal).

Le temps de recherche des solutions est également un facteur de performance de la méthode génétique proposée. Le tableau 6.3 résume l'ordre de grandeur des temps de recherche en fonction de la taille de la trace mémoire. Les simulations sont effectuées sur une machine composée d'un processeur *intel i3* et cadencée à *2.3GHz*.

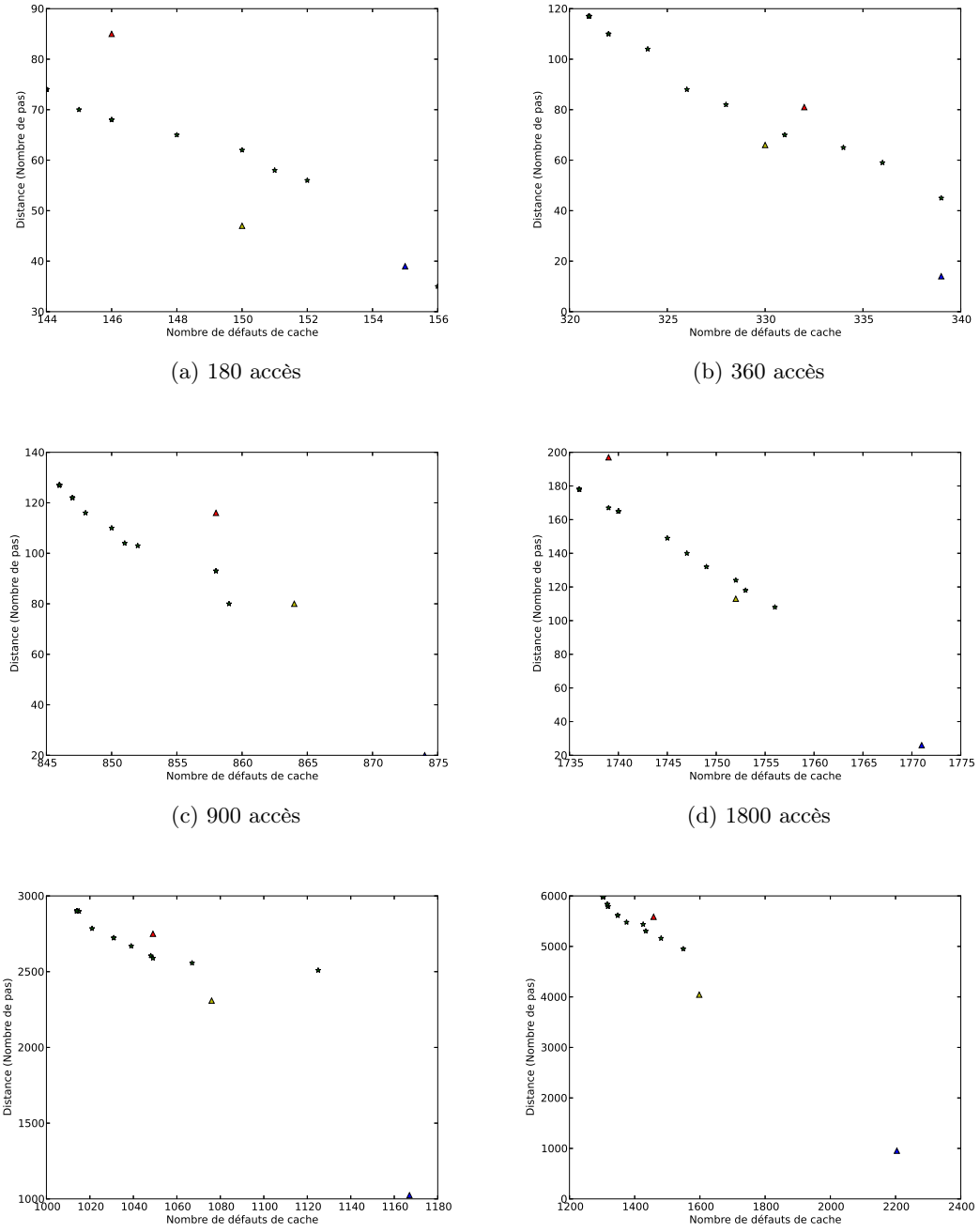
À un nombre d'itérations et d'individus fixe le temps de recherche augmente avec le nombre de données à traiter. Ceci est dû principalement au temps de simulation comme mentionné précédemment.

Nombre d'accès	180	360	900	1800	2080	3264
Temps de recherche (s)	398	675	1544	2965	2931	4324

TABLE 6.3 – Temps de recherche en fonction de la taille de la trace mémoire.

Les solutions obtenues par la méthode génétique représentent un meilleur compromis entre les deux objectifs de performance qui sont la distance et le nombre de défauts de cache. Elles sont caractérisées par une distance cumulative réduite (plus petite que la distance obtenue avec un rayon maximal), ce qui signifie une réduction des temps d'accès mémoire, tout en maintenant un taux de défauts de cache plus bas que la solution à rayon minimal.

Le choix d'une solution parmi celles appartenant au *Pareto* dépend du profile de performance défini par l'utilisateur. Dans notre cas d'étude, un profil d'économie d'énergie nécessite de choisir la solution optimale par rapport à la distance cumulative afin de limiter la propagation des données dans la puce ce qui peut engendrer une hausse de la consommation d'énergie. Une profil de haute performance privilégierait de réduire le nombre de défauts de cache au détriment du coût énergétique.



(a) 180 accès

(b) 360 accès

(c) 900 accès

(d) 1800 accès

(e) 2080 accès

(f) 3264 accès

FIGURE 6.13 – Solutions *Pareto* obtenues avec la méthode génétique versus des solutions aléatoires à : rayon maximal (rouge), rayon Moyen (jaune), rayon minimal (bleu).

6.4 Discussion

Nous proposons dans ce travail une plate-forme de compilation pour le choix et la configuration des protocoles de cohérence. Ce processus de compilation représente de nombreux avantages. Un premier avantage qui est que toutes les prises de décisions sont effectuées hors ligne, pendant la compilation ce qui n'engendre aucun surcôut à l'exécution. Un deuxième avantage toujours lié au choix d'implémentation de la plate-forme est son architecture flexible permettant de modifier les différentes briques indépendamment et aussi d'ajouter de nouveaux étages d'analyse, de décision et d'évaluation. Un deuxième avantage cette approche est sa flexibilité vis-à-vis des protocoles de cohérence pris en compte notamment avec la possibilité d'intégrer de nouveaux protocoles. Le moteur d'optimisation basé sur une approche génétique garantit des temps de configuration raisonnables avec une processus d'évaluation offrant plusieurs métriques de performance. Au final, la plate-forme multi-protocolaire proposée permet de répondre à la question de la gestion de la cohérence tout en prenant en compte le comportement de l'application aussi bien que l'architecture d'exécution tout en offrant à l'utilisateur de faire des choix de profils de performances selon son contexte applicatif.

De nombreuses pistes restent à étudier, notamment au niveau de la caractérisation de l'application et du moteur d'optimisation. Ces pistes seront discutées dans la conclusion.

Conclusion et Perspectives

Les systèmes manycœurs profitent de l'évolution des technologies des transistors, des réseaux de connexion et des mémoires. Ils promettent une grande puissance de calcul à basse consommation énergétique. Ces systèmes constitués de plusieurs centaines d'unités de calcul à basse fréquence sont capables d'effectuer plusieurs milliards d'opérations par seconde. La puce *MPPA-256* de *Kalray* atteint 460 *GIPS*, et 230 *GFLOPS* en simple précision avec une consommation moyenne de 5 watts. Pour comparaison, un processeur *Intel i7 980X* a une performance de 90 $GFLOPS$ pour une consommation de 130 watts.

Cependant, l'utilisation efficace des manycœurs repose grandement sur la qualité de gestion du parallélisme dans ces applications. La gestion de la mémoire est un facteur clé de performance qui peut dégrader la vitesse de traitement des données. Les performances mémoire dans les systèmes manycœurs sont limitées d'abord par la surface qui empêche l'utilisation de grands espaces de stockage sur la puce. Elles sont limitées également par le temps d'accès à la mémoire extérieure décrit comme le mur de la mémoire (en anglais *Memory Wall*).

Du point de vue de la mémoire, l'obtention de hautes performances dépend entre autres de la bonne utilisation de la mémoire sur puce (ex. réduction des défauts de cache) et de l'efficacité de gestion de la cohérence des données face à la concurrence d'accès mémoire. Dans un modèle de programmation par passage de messages par exemple le programmeur est responsable de toutes les communications qui lui permettent d'avoir accès aux données partagées. La complexité et le coût de programmation deviennent de plus en élevés pour des programmes complexes avec beaucoup de partages de ressources.

L'utilisation du paradigme de programmation à mémoire virtuellement partagée où les conflits d'accès aux données sont gérés de façon transparente a comme objectif de simplifier la tâche du programmeur en lui donnant l'impression d'un espace d'adressage unifié à l'ensemble des mémoires sur la puce. Dans un tel paradigme de programmation la gestion des données est dévolue au système : une gestion efficace de la mémoire est sensée prendre en compte le comportement de l'application aussi bien que les caractéristiques de la plate-forme cible.

À notre connaissance, il n'existe pas encore de systèmes basés sur de la mémoire virtuellement partagée qui permet de prendre des décisions sur les protocoles et mécanismes de gestion des données en prenant en compte l'application et la cible d'exécution.

Contributions

Les travaux de cette thèse ont comme objectif de répondre à la question de la gestion de la cohérence pour des architectures massivement parallèles tels que les manycœurs afin de garantir une gestion efficace de l'espace de stockage dans ce genre de systèmes. Nous proposons dans ces travaux une plate-forme de compilation à protocoles multiples qui permet de faire des choix de protocoles de cohérence selon les caractéristiques de l'application et les spécifications de l'architecture cible. La prise de décision et la configuration des protocoles choisis s'effectuent via deux principales phases. Une première phase d'analyse statique qui consiste à caractériser les accès aux données partagées permettant de prendre des décisions sur le choix des protocoles. Une deuxième phase de raffinement et de configuration de ces choix qui permet d'adapter les paramètres des protocoles à la plate-forme d'exécution et à certains choix de performances définis par l'utilisateur. Les travaux effectués dans le cadre cette thèse ont abouti à plusieurs propositions :

Protocoles de glissement de données : les protocoles de glissement de données proposés dans le cadre de ce travail démontrent qu'ils sont bien adaptés à la gestion de la charge mémoire dans les zones stressées de la puce (*points chauds*). Il s'agit de protocoles coopératifs permettant de migrer la charge mémoire des caches surchargées de la puce vers les caches les moins chargés. Le processus de migration des données est paramétré par le rayon de glissement accordé à chaque donnée définissant la distance d'éloignement de son nœud propriétaire. Afin de définir un rayon de glissement variable qui s'adapte dynamiquement à l'état de charge de la puce, nous avons proposé une extension du protocole de glissement de données grâce au modèle physique du masse-ressort. Cependant, l'étude des performances de ces protocoles démontre que leur performance dépend de plusieurs facteurs tels que le comportement de l'application, la manière dont la charge est répartie sur la puce ainsi que certaines propriétés architecturales comme la topologie du *NoC*. Les résultats obtenus dans notre étude comparative des protocoles de cohérence démontrent que les performances de l'application peuvent varier (jusqu'à un grain de $\times 10$) en fonction du protocole associé et de sa configuration (ex. le rayon de glissement). Le choix d'un protocole est donc primordial pour l'amélioration des performances d'un système.

Modèle d'analyse statique pour le choix des protocoles : Le modèle proposé permet de représenter de façon formelle les accès concurrents aux données et de les caractériser par la suite en fonction de différentes métriques telles que la fréquence d'utilisation d'une donnée ou la distance entre accès fréquents. Ce type d'analyse portant sur les accès partagés permet de détecter des schémas particuliers d'accès à la mémoire (ex. producteur-consommateur). La caractérisation préalable des protocoles de cohérence pris en charge par la plate-forme permet d'associer un protocole à chaque accès mémoire.

Processus de raffinement et de configuration : Le processus de configuration permet d'instancier les protocoles de cohérence afin de garantir un niveau de perfor-

mance optimal à l'exécution. À chaque accès à une donnée est associé un protocole de cohérence dont les paramètres sont instanciés selon le type de comportement souhaité. Les valeurs des paramètres de protocoles de cohérence sont bornées par certaines caractéristiques de l'application comme la taille de la puce ou la taille des mémoires caches. La complexité du problème de paramétrage des protocoles augmente exponentiellement avec la taille de l'application (en terme de nombre d'accès à traiter), le nombre de paramètres associés à chaque protocole et leurs intervalles de définition. Il est donc très coûteux d'opter pour une recherche exhaustive. Afin de limiter le temps de recherche des solutions, nous proposons un modèle d'optimisation génétique optimisé permettant de prendre en compte plusieurs objectifs de performance tels que le nombre de défauts de cache et la distance de migration pour l'exemple du protocole de glissement de données. Le modèle génétique d'optimisation effectue une analyse itérative de l'espace des instances possibles de chacun des protocoles choisis. À chaque itération plusieurs solutions sont évaluées et classées selon leur adéquation avec les objectifs de performance envisagés. Enfin, parmi un ensemble de solutions optimisées formant un front de *Pareto*, une solution est sélectionnée en fonction du profil de performance souhaité. L'évaluation des différentes générations de solutions nécessite un processus simple et rapide. Les modèles d'évaluations proposés dans cette thèse permettent d'obtenir des configurations optimisées des protocoles de cohérence dans un temps raisonnable (de l'ordre de quelques minutes pour les plus grandes traces mémoire). Le paragraphe suivant décrit les modèles de performance proposés dans le cadre de ce travail.

Modèles d'évaluation : Le processus de décision nécessite des outils d'évaluation des différentes solutions parcourues par l'algorithme génétique. Nous proposons deux modèles d'évaluation des protocoles. Un premier modèle analytique qui permet d'obtenir des statistiques sur les communications et les échanges de messages générés par le protocole de cohérence sous-jacent. Il permet également de mesurer les performances mémoire tels que le nombre de défauts de cache et la répartition de la charge sur les caches de la puce. Le deuxième modèle temporel permet d'estimer les pénalités temporelles d'accès aux données et l'effet de contention sur les performances du *NoC*. Les modèles proposés offrent la possibilité de faire varier les caractéristiques de la plate-forme d'exécution, notamment pour le réseau *NoC* telles que la taille de la puce et le type de topologie. Ils sont caractérisés par des temps de simulation raisonnables (de l'ordre de quelque minutes) comparés à des modèles de simulation de bas niveau où ce temps peut atteindre plusieurs dizaines voire centaines de minutes, ce qui facilite leur intégration dans une chaîne de compilation.

Le travail qui a été réalisé dans le cadre de cette thèse part de l'étude des protocoles de cohérence de données pour des architectures massivement parallèles, jusqu'à la proposition d'une plate-forme de compilation multi-protocolaire qui permet de prendre des décisions efficaces sur le choix et le paramétrage des protocoles étudiés. Nous avons réussi à valider à chaque niveau de la plate-forme les approches utilisées dans la prise

de décision. L'intérêt de faire de tels choix lors de la phase de compilation repose sur plusieurs avantages : Éviter un surcoût important lors de l'exécution pouvant dégrader drastiquement la vitesse de traitement des accès, offrir la possibilité de configurer une même application pour différentes cibles à moindre coût et proposer une grande flexibilité dans le choix des métriques de performance que l'utilisateur peut définir en fonction de ses besoins applicatifs.

Perspectives

Étude de la granularité de décision et de son impact sur les performances en ligne

Le processus de compilation permet d'associer un protocole de cohérence à chaque accès à une donnée. L'ensemble de ces protocoles est instancié et paramétré selon trois principaux critères de décision qui sont : le comportement de l'application, les propriétés de la plate-forme d'exécution et les choix de performances de l'utilisateur. Une granularité de décision très fine permet d'augmenter le niveau de précision dans le choix des protocoles de cohérence. Cependant, une telle précision peut engendrer plusieurs changements de contextes lors de l'exécution. Cette thèse n'étudie pas le coût du changement de contexte sur une exécution réelle mais nous pouvons imaginer qu'une fréquence élevée de changement dans un contexte multi-protocolaire peut pénaliser les performances du système. Le problème qui se pose est : comment décider du niveau de décision optimal pour une application donnée ? Chaque application est constituée de plusieurs phases de comportement. Chaque phase représente une charge mémoire différente avec un schéma d'accès spécifique. Une telle information peut être déduite de l'analyse statique de l'application en définissant des critères de répartition de l'application qui permettent de conserver un caractère comportemental homogène à l'intérieur de chaque phase. La possibilité de variation du niveau de granularité pour une même application ouvre sur une nouvelle perspective qui est d'étudier l'impact de la fréquence de changement de protocoles de cohérence en ligne sur les performances du système.

Compilation itérative

La configuration des protocoles de cohérence est un processus complexe. Nous proposons dans ce travail une méthode utilisant l'optimisation génétique qui permet d'obtenir un ensemble de solutions appartenant à un front de *Pareto*. Le choix d'une solution est aujourd'hui basé uniquement sur le profil de performance choisi par l'utilisateur. La compilation itérative est un concept souvent utilisé dans l'optimisation des performances des programmes [BKK+98, KKO03]. Le code d'un programme subit plusieurs transformations donnant des solutions différentes qui seront par la suite évaluées en les exécutant sur la cible réelle. À la fin d'un certain nombre d'exécutions une solution est choisie. L'intégration de cette approche itérative dans la chaîne de compilation proposée permettrait d'avoir un retour sur les décisions prises lors de la phase de compilation. Elle peut être utilisée à la fin du processus de décision afin d'arbitrer entre les solutions finales de *Pareto*. Une évaluation lors de l'exécution de ces solutions donnerait

une idée plus précise des performances obtenues par rapport à l'utilisation des modèles d'évaluation de haut niveau.

Prise en charge de nouveaux protocoles de cohérence

La bibliothèque des protocoles intégrée à la chaîne de compilation est extensible et peut prendre en charge de nouveaux protocoles de cohérence. Nous étudions dans le cadre de cette thèse un protocole à quatre états et un nœud gestionnaire pour chaque donnée, le protocole de *glissement de données* à rayon fixe et le protocole de *masse-ressort*. Il serait intéressant de pouvoir intégrer de nouveaux protocoles. Une perspective de ce travail est de mettre en place une interface permettant à l'utilisateur de proposer de nouveaux mécanismes de gestion des données en fonction de ses besoins.

Le problème de la gestion de la cohérence a été étudié pour de nombreuses plate-formes comme les grilles et grappes de calculateurs ce qui a permis de proposer différentes approches de gestion de la cohérence. Ces approches sont pour la plupart spécifiques à ce type d'architectures. Nous proposons dans les travaux de cette thèse une plate-forme de compilation multi-protocolaire permettant de prendre en compte les caractéristiques de la plate-forme d'exécution dans le choix des protocoles. La validation de notre approche porte sur des architectures sur puce de type manycœurs. Elle considère des contraintes comme les performances de la mémoire sur puce (ex. nombre de succès de cache), la surface (ex. la taille de la puce) et la consommation énergétique (ex. répartition de la consommation au niveau de la puce). L'approche multi-protocolaire proposée peut donc être appliquée sur d'autres types d'architectures sur puce avec des contraintes similaires. Les *microserveurs* (concept apparu en 2010) est un exemple d'architectures sur puce à haute performance et basse consommation énergétique. Le déploiement de notre plate-forme dans un tel contexte permettrait d'étudier la propriété d'hétérogénéité ce qui n'a pas été considéré dans ces travaux.

Références

[ABB+13] Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Loïc Cudennec, Vincent David, Philippe Doré, Paul Dubrulle, Benoît Dupont De Dinechin, et al. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. *Procedia Computer Science*, 18 :1624–1633, 2013.

[ABJ05] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. Juxmem : An adaptive supportive platform for data sharing on the grid. In *Scalable Computing : Practice and Experience*, volume 6, pages 45–55, 2005.

[AC14] Tariq Bashir Ahmad and Maciej Ciesielski. Fast time-parallel c-based event-driven rtl simulation. In *Design and Diagnostics of Electronic Circuits & Systems, 17th International Symposium on*, pages 71–76. IEEE, 2014.

[ACG+12] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and approximation : Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 2012.

[AGKL03] Jay April, Fred Glover, James P Kelly, and Manuel Laguna. Practical introduction to simulation optimization. In *Simulation Conference, 2003. Proceedings of the 2003 Winter*, volume 1, pages 71–78. IEEE, 2003.

[Ant01] Gabriel Antoniu. *DSM-PM2 : une plate-forme portable pour l'implémentation de protocoles de cohérence multithreads pour systèmes à mémoire virtuellement partagée*. PhD thesis, Ecole normale supérieure de lyon-ENS LYON, 2001.

[BAB+06] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H Loh, David McCauley, Pat Morrow, Donald W Nelson, Daniel Pantuso, et al. Die stacking (3d) microarchitecture. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 469–479. IEEE, 2006.

[BBHK05] Silvia Breu, David W Binkley, Mark Harman, and Jens Krinke. Extending c global surveyor. In *Beyond Program Slicing*, 2005.

[BCC+10] Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, et al. Analyzing parallel programs with pin. *Computer*, 43(3) :34–41, 2010.

[BCZ90] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin : Distributed shared memory based on type-specific memory coherence. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176, 1990.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo : a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE, 2003.

[BJM11] Rajeev Balasubramonian, Norman P Jouppi, and Naveen Muralimanohar. Multi-core cache hierarchies. *Synthesis Lectures on Computer Architecture*, 6(3) :1–153, 2011.

[BKK+98] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[BKT07] Jeffery A Brown, Rakesh Kumar, and Dean Tullsen. Proximity-aware directory-based coherence for multi-core processor architectures. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 126–134. ACM, 2007.

[Bla93] Matthew Addison Blaze. *Caching in large-scale distributed file systems.* PhD thesis, Princeton University, 1993.

[BME07] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. Programming distributed memory sytems using openmp. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[BPCB10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303. Springer, 2010.

[BR90] Roberto Bisiani and Mosur Ravishankar. Plus : A distributed shared-memory system. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 115–124. IEEE, 1990.

[BS01] Gérard Berthiau and Patrick Siarry. état de l'art des méthodes d'optimisation globale. *RAIRO-Operations Research*, 35(03) :329–365, 2001.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the second International Symposium on Programming*, pages 106–130. Paris, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CCD+15] Hamza Chaker, Loïc Cudennec, Safae Dahmani, Guy Gogniat, and Martha Johanna Sepúlveda. Cycle-based model to evaluate consistency protocols within a multi-protocol compilation tool-chain. In *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*, page 8. ACM, 2015.

[CCF+05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In *Programming Languages and Systems*, pages 21–30. Springer, 2005.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2) :244–263, 1986.

[CF78] Lucien M Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *Computers, IEEE Transactions on*, 100(12) :1112–1118, 1978.

[CGL94] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5) :1512–1542, 1994.

[CGL96] Toni Cortes, Sergi Girona, and Jesús Labarta. Paca : A cooperative file system cache for parallel machines. In *Euro-Par'96 Parallel Processing*, pages 475–486. Springer, 1996.

[Cha89] Lazowska Levy Chase, Amador. The amber system : Parallel programming on a network of multiprocessors. In *ACM Symposium on Operating Systems Principles*, pages 147–158. ACM, 1989.

[Chr14] George Chrysos. Intel® xeon phi coprocessor-the architecture. *Intel Whitepaper*, 2014.

[CLC07] Chi-Yin Chow, Hong Va Leong, and Alvin TS Chan. Grococa : Group-based peer-to-peer cooperative caching in mobile environment. *Selected Areas in Communications, IEEE Journal on*, 25(1) :179–191, 2007.

[CLIF08] "CEA-LIST and INRIA-Futurs". "frama-c : Framework for modular analysis of c", "2008". "http ://www.frama-c.cea.fr".

[COK+03] Joonho Cho, Seungtaek Oh, Jaemyoung Kim, Hyeong Ho Lee, and Joonwon Lee. Neighbor caching in multi-hop wireless ad hoc networks. *Communications Letters, IEEE*, 7(11) :525–527, 2003.

[Cor11] "Intel Corporation". "introducing intel many integrated core architecture", "2011". "www.intel.com/technology/architecture-silicon/mic/index.htm".

[Cou00] Patrick Cousot. Interprétation abstraite. *Technique et science informatique*, 19(1-2) :3, 2000.

[CS06] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA'06*, pages 264–276, 2006.

[CS07] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *International Conference on Supercomputing*, pages 242–252, 2007.

[CS13] Yann Collette and Patrick Siarry. *Multiobjective optimization : principles and case studies*. Springer Science & Business Media, 2013.

[CVVL02] Carlos A Coello Coello, David A Van Veldhuizen, and Gary B Lamont. *Evolutionary algorithms for solving multi-objective problems*, volume 242. Springer, 2002.

[DA94] Kalyanmoy Deb and Ram B Agrawal. Simulated binary crossover for continuous search space. *Complex Systems*, 9(3) :1–15, 1994.

[DBMO11] Radu David, Paul Bogdan, Radu Marculescu, and Umit Ogras. Dynamic power management of voltage-frequency island partitioned networks-on-chip using intel's single-chip cloud computer. In *Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on*, pages 257–258. IEEE, 2011.

[dDAB+13] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, Frederique Jacquet, Simon Jones, Nicolas Morey Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.

[DE98] Leonardo Dagum and Rameshm Enon. Openmp : an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1) :46–55, 1998.

[Deb01] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[DKM08] Nikos Dimokas, Dimitrios Katsaros, and Yannis Manolopoulos. Cooperative caching in wireless multimedia sensor networks. *Mobile Networks and Applications*, 13(3-4) :337–356, 2008.

[DKTM11] Nikos Dimokas, Dimitrios Katsaros, Leandros Tassiulas, and Yannis Manolopoulos. High performance, low complexity cooperative caching for wireless sensor networks. *Wireless Networks*, 17(3) :717–737, 2011.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm : Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2) :182–197, 2002.

[DS07] Haakon Dybdahl and Per Stenström. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 2–12. IEEE, 2007.

[DSB+88] Michel Dubois, Christoph Scheurich, Faye A. Briggs, et al. Synchronization, coherence, and event ordering in multiprocessors. *IEEE computer*, 21(2) :9–21, 1988.

[DSF88] Gary Delp, Adarshpal Sethi, and David Farber. An analysis of memnet an experiment in high-speed shared-memory local networking. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 165–174. ACM, 1988.

[DWAP94] Michael Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching : Using remote client memory to improve file system performance. In *Operating System Design and Integration*, pages 267–280, 1994.

[EGL07] Hamidreza Eskandari, Christopher D Geiger, and Gary B Lamont. Fastpga : A dynamic population sizing approach for solving expensive multiobjective optimization problems. In *Evolutionary Multi-Criterion Optimization*, pages 141–155. Springer, 2007.

[FBC87] Alessandro Forin, Roberto Bisiani, and Franco Correrini. Parallel processing with agora. 1987.

[FCL92] Michael J Franklin, Michael James Carey, and Miron Livny. *Global memory management in client-server DBMS architectures*. University of Wisconsin-Madison, Computer Sciences Department, 1992.

[FP89] Brett Fleisch and Gerald Popek. *Mirage : A coherent distributed shared memory design*, volume 23. ACM, 1989.

[GADM+05] Nicolas Genko, David Atienza, Giovanni De Micheli, Jose Manuel Mendias, Roman Hermida, and Francky Catthoor. A complete network-on-chip emulation framework. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 246–251. IEEE, 2005.

[Gei94] Al Geist. Pvm : parallel virtual machine. *A users' guide and tutorial for networked parallel computing*, 1994.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6) :789–828, 1996.

[GM75] Geoffrey R Grimmett and Colin JH McDiarmid. On colouring random graphs. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 77, pages 313–324. Cambridge Univ Press, 1975.

[Gol89] David E Golberg. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989, 1989.

[GVL10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks : high-level structured parallel programming enablers. *Software : Practice and Experience*, 40(12) :1135–1160, 2010.

[HBH+07] Christoph A Herrmann, Armelle Bonenfant, Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl, and Robert F Pointon. Automatic amortised worst-case execution time analysis. In *WCET*, 2007.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Notices*, volume 38, pages 388–402. ACM, 2003.

[HFSB08] Ralf Huuck, Ansgar Fehnker, Sean Seefried, and Jörg Brauer. Goanna : Syntactic software model checking. In *Automated Technology for Verification and Analysis*, pages 216–221. Springer, 2008.

[HGC02] Enric Herrero, José González, and Ramon Canal. Ivy : a read/write peer-to-peer file system. In *Proceedings of the 5th symposium on Operating systems design and implementation*, pages 31–44, 2002.

[HGC08] Enric Herrero, José González, and Ramon Canal. Distributed cooperative caching. In *Parallel Architectures and Compilation Techniques*, pages 134–143, 2008.

[HGC10a] Enric Herrero, José González, and Ramon Canal. Elastic cooperative caching : an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *ISCA*, pages 419–428, 2010.

[HGC10b] Enric Herrero, José González, and Ramon Canal. Power-efficient spilling techniques for chip multiprocessors. In *Proceedings of the 16th international Euro-Par conference on Parallel processing : Part I*, pages 256–267, 2010.

[HLM06] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *ACM Sigplan Notices*, volume 41, pages 253–262. ACM, 2006.

[HLMSI03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[HM93] Maurice Herlihy and J Eliot B Moss. *Transactional memory : Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[Hol04] Gerard J Holzmann. *The SPIN model checker : Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[HP03] "D. Hovemeyer and W. Pugh". "finding bugs is easy", "2003". "http ://www.cs.umd.edu".

[HP12] John L Hennessy and David A Patterson. *Computer architecture : a quantitative approach*. Elsevier, 2012.

[Jha93] Seong Tae Jhang. A new write-invalidate snooping cache coherence protocol for split transaction bus-based multiprocessor systems. In *TENCON'93. Proceedings. Computer, Communication, Control and Power Engineering*, pages 229–232, 1993.

[JJA08] Raoul Praful Jetley, Paul L Jones, and Paul Anderson. Static analysis of medical device software using codesonar. In *Proceedings of the 2008 workshop on Static analysis*, pages 22–29. ACM, 2008.

[Joh77] Stephen C Johnson. *Lint, a C program checker*. Citeseer, 1977.

[KC99] Joshua Knowles and David Corne. The pareto archived evolution strategy : A new baseline algorithm for pareto multiobjective optimisation. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 1. IEEE, 1999.

[KCDZ94a] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks : Distributed shared memory on standard workstations and operating systems. In *USENIX Winter*, volume 1994, pages 23–36, 1994.

[KCDZ94b] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks : Distributed shared memory on standard workstations and operating systems. In *USENIX Winter*, volume 1994, pages 23–36, 1994.

[Kim12] Kinam Kim. Future silicon technology. In *Solid-State Device Research Conference (ESSDERC), 2012 Proceedings of the European*, pages 1–6. IEEE, 2012.

[KKO03] Peter MW Knijnenburg, Toru Kisuki, and Michael FP O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24(1) :43–67, 2003.

[KMW67] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM (JACM)*, 14(3) :563–590, 1967.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, 1978.

[LGL+90] Daniel Lenoski, Kourosh Gharachorloo, James Laudon, Ahoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of scalable shared-memory multiprocessors : The dash approach. In *Compcon Spring'90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pages 62–67. IEEE, 1990.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4) :321–359, 1989.

[LL97] James Laudon and Daniel Lenoski. The sgi origin : a ccnuma highly scalable server. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 241–251. ACM, 1997.

[LLG+90a] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. *The directory-based cache coherence protocol for the DASH multiprocessor*, volume 18. ACM, 1990.

[LLG+90b] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *International Symposium on Computer Architecture*, pages 148–159, 1990.

[LP92] Zakaria Lahjomri and Thierry Priol. *Koan : a shared virtual memory for the ipsc/2 hypercube*. Springer, 1992.

[LS89] Kai Li and Richard Schaefer. *Shiva : An operating system transforming a hypercube into a shared-memory machine*. Princeton University, Department of Computer Science, 1989.

[LSL+09] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 57–67. IEEE, 2009.

[LWY93] Avraham Leff, Joel L Wolf, and Philip S. Yu. Replication algorithms in a remote caching architecture. *Parallel and Distributed Systems, IEEE Transactions on*, 4(11) :1185–1204, 1993.

[LYW91] Avraham Leff, Philip S. Yu, and Joel L Wolf. Policies for efficient memory utilization in a remote caching architecture. In *Parallel and Distributed Information Systems, 1991., Proceedings of the First International Conference on*, pages 198–207. IEEE, 1991.

[LYW93] Avraham Leff, Philip S Yu, and Joel L Wolf. Performance issues in object replication for a remote caching architecture. *Computer Systems Science and Engineering*, 8(1) :40–51, 1993.

[MBL+01] Shubhendu S Mukherjee, Peter Bannon, Steven Lang, Aaron Spink, and David Webb. The alpha 21364 network architecture. In *Hot Interconnects 9, 2001.*, pages 113–117. IEEE, 2001.

[MH91] Daniel Muntz and Peter Honeyman. Multi-level caching in distributed file systems. Technical report, Center for Information Technology Integration, 1991.

[MLC+12] Jussara Marandola, Stephane Louise, Loïc Cudennec, J Acquaviva, and David A Bader. Enhancing cache coherent architectures with access patterns for embedded manycore systems. In *System on Chip (SoC), 2012 International Symposium on*, pages 1–7. IEEE, 2012.

[MMB+03] Theodore Marescaux, Jean-Yves Mignolet, Andrei Bartic, Will Moffat, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Networks on chip as hardware components of an os for reconfigurable systems. In *Field Programmable Logic and Application*, pages 595–605. Springer, 2003.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming : A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[Ngu15] Tien Tanh Nguyen. Choix de protocole de cohérence des données pour architectures massivement parallèles. Master's thesis, 2015.

[NL91] Bill Nitzberg and Virginia Lo. Distributed shared memory : A survey of issues and algorithms. *Distributed Shared Memory-Concepts and Systems*, pages 42–50, 1991.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[Ope97] Openmp forum. openmp : A proposed industry standard api for shared memory programming. 1997.

[OVP08] Open virtual platforms, 2008.

[Pet62] Carl Adam Petri. Communicating with automata. *Germany : PhD thesis, Technical University Darmstadt*, 1962.

[PH13] David A Patterson and John L Hennessy. *Computer organization and design : the hardware/software interface*. Newnes, 2013.

[Pom04] Franck Pommereau. Snakes is the net algebra kit for editors and simulators. Comete Procope Workshop, 2004.

[Ram11a] Carl Ramey. Tile-gx100 manycore processor : Acceleration interfaces and architecture. In *Proceedings of the 23th Hot Chips Symposium*, 2011.

[Ram11b] Carl Ramey. Tile-gx100 manycore processor : Acceleration interfaces and architecture. In *Proceedings of the 23th Hot Chips Symposium*, 2011.

[RB88] A. Forin" "R. Bisiani. "multilanguage parallel programming of heterogeneous machines". In *"Solid-State Device Research Conference (ESSDERC), 2012 Proceedings of the European"*, pages "930–945". "IEEETC", "1988".

[SBL04] Taeweon Suh, Douglas M Blough, and Hsien-Hsin S Lee. Supporting cache coherence in heterogeneous multiprocessor systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 1150–1155. IEEE, 2004.

[SJKD05] Huaping Shen, Mary Suchitha Joseph, Mohan Kumar, and Sajal K Das. Precinct : A scheme for cooperative caching in mobile peer-to-peer systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 57a–57a. IEEE, 2005.

[SR+99] Xiaowei Shen, Larry Rudolph, et al. Cachet : an adaptive cache coherence protocol for distributed shared-memory systems. In *Proceedings of the 13th international conference on Supercomputing*, pages 135–144. ACM, 1999.

[SSW07] Johanna Sepúlveda, M Strum, and JC Wang. A tlm-based network-on-chip performance evaluation framework. In *Proc. 3rd Symposium on Circuits and Systems, Colombian Chapter*, pages 54–60, 2007.

[ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2) :99–116, 1997.

[Sys04] Open systemc iniative osci, systemc documentation., 2004.

[Tan76] CK Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 749–753. ACM, 1976.

[TBD+87] Marjorie Templeton, David Brill, Son K Dao, Eric Lund, Patricia Ward, Arbee LP Chen, and Robert MacGregor. Mermaid : A front-end to distributed heterogeneous databases. *Proceedings of the IEEE*, 75(5) :695–708, 1987.

[VGRN08] Carlos Villavieja, Isaac Gelado, Alex Ramirez, and Nacho Navarro. Memory management on chip-multiprocessors with on-chip memories. In *Proc. Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008.

[VGS+10] Nicolas Ventroux, Alexandre Guerre, Tanguy Sassolas, L Moutaoukil, Guillaume Blanc, Charly Bechara, and Raphaël David. Sesam : An mpsoc simulation environment for dynamic application processing. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1880–1886. IEEE, 2010.

[VHR+07] Sriram Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, et al. An 80-tile 1.28 tflops network-on-chip in 65nm cmos. In *IEEE International Solid-State Circuits Conference, ISSCC 2007, Digest of Technical Papers, San Francisco, CA, USA*, pages 98–99. IEEE, 2007.

[VV99] David A Van Veldhuizen. Multiobjective evolutionary algorithms : classifications, analyses, and new innovations. Technical report, DTIC Document, 1999.

[WHS07] Pascal T Wolkotte, Philip KF Holzenspies, and Gerard JM Smit. Fast, accurate and detailed noc simulations. In *Networks-on-Chip, 2007. NOCS 2007*, pages 323–332. IEEE, 2007.

[YC06] Liangzhong Yin and Guohong Cao. Supporting cooperative caching in ad hoc networks. *Mobile Computing, IEEE Transactions on*, 5(1) :77–89, 2006.

[ZIUN08] Li Zhao, Ravi Iyer, Mike Upton, and Don Newell. Towards hybrid last level caches for chip-multiprocessors. *SIGARCH Comput. Archit. News*, 36(2) :56–63, May 2008.

[ZLT+01] Eckart Zitzler, Marco Laumanns, Lothar Thiele, Eckart Zitzler, Eckart Zitzler, Lothar Thiele, and Lothar Thiele. Spea2 : Improving the strength pareto evolutionary algorithm, 2001.

[ZSLW98] Songnian Zhou, Michael Stumm, Kai Li, and David Wortman. Heterogeneous distributed shared memory. *Distributed Shared Memory : Concepts and Systems*, 21 :196, 1998.

[ZT99] Eckart Zitzler and Lothar Thiele. Multiobjective evolutionary algorithms : a comparative case study and the strength pareto approach. *evolutionary computation, IEEE transactions on*, 3(4) :257–271, 1999.

Liste des définitions

Table des figures

Résumé

Le développement des systèmes massivement parallèles de type manycœurs permet d'obtenir une très grande puissance de calcul à bas coût énergétique. Cependant, l'exploitation des performances de ces architectures dépend de l'efficacité de programmation des applications. Parmi les différents paradigmes de programmation existants, celui à mémoire partagée est caractérisé par une approche intuitive dans laquelle tous les acteurs disposent d'un accès à un espace d'adressage global. Ce modèle repose sur l'efficacité du système à gérer les accès aux données partagées. Le système définit les règles de gestion des synchronisations et de stockage de données qui sont prises en charge par les protocoles de cohérence. Dans le cadre de cette thèse nous avons montré qu'il n'y a pas un unique protocole adapté aux différents contextes d'application et d'exécution. Nous considérons que le choix d'un protocole adapté doit prendre en compte les caractéristiques de l'application ainsi que des objectifs donnés pour une exécution. Nous nous intéressons dans ces travaux de thèse au choix des protocoles de cohérence en vue d'améliorer les performances du système. Nous proposons une plate-forme de compilation pour le choix et le paramétrage d'une combinaison de protocoles de cohérence pour une même application. Cette plate-forme est constituée de plusieurs briques. La principale brique développée dans cette thèse offre un moteur d'optimisation pour la configuration des protocoles de cohérence. Le moteur d'optimisation, inspiré d'une approche évolutionniste multiobjectifs (i.e. Fast Pareto Genetic Algorithm), permet d'instancier les protocoles de cohérence affectés à une application. L'avantage de cette technique est un coût de configuration faible permettant d'adopter une granularité très fine de gestion de la cohérence, qui peut aller jusqu'à associer un protocole par accès. La prise de décision sur les protocoles adaptés à une application est orientée par le mode de performance choisi par l'utilisateur (par exemple, l'économie d'énergie). Le modèle de décision proposé est basé sur la caractérisation des accès aux données partagées selon différentes métriques (par exemple : la fréquence d'accès, les motifs d'accès à la mémoire, etc). Les travaux de thèse traitent également des techniques de gestion de données dans la mémoire sur puce. Nous proposons deux protocoles basés sur le principe de coopération entre les caches répartis du système : Un protocole de glissement des données ainsi qu'un protocole inspiré du modèle physique du masse-ressort.

Abstract

Manycœurs architectures consist of hundreds to thousands of embedded cores, distributed memories and a dedicated network on a single chip. In this context, and because of the scale of the processor, providing a shared memory system has to rely on efficient hardware and software mechanisms and data consistency protocols. Numerous works explored consistency mechanisms designed for highly parallel architectures. They lead to the conclusion that there won't exist one protocol that fits to all applications and hardware contexts. In order to deal with consistency issues for this kind of architectures, we propose in this work a multi-protocol compilation toolchain, in which shared data of the application can be managed by different protocols. Protocols are chosen and configured at compile time, following the application behavior and the targeted architecture specifications. The application behavior is characterized with a static analysis process that helps to guide the protocols assignment to each data access. The platform offers a protocol library where each protocol is characterized by one or more parameters. The range of possible values of each parameter depends on some constraints mainly related to the targeted platform. The protocols configuration relies on a genetic-based engine that allows to instantiate each protocol with appropriate parameters values according to multiple performance objectives. In order to evaluate the quality of each proposed solution, we use different evaluation models. We first use a traffic analytical model which gives some NoC communication statistics but no timing information. Therefore, we propose two cycle-based evaluation models that provide more accurate performance metrics while taking into account contention effect due to the consistency protocols communications. We also propose a cooperative cache consistency protocol improving the cache miss rate by sliding data to less stressed neighbors. An extension of this protocol is proposed in order to dynamically define the sliding radius assigned to each data migration. This extension is based on the mass-spring physical model. Experimental validation of different contributions uses the sliding based protocols versus a four-state directory-based protocol.