Étude de deux solutions pour le support matériel de la programmation parallèle dans les multiprocesseurs intégrés : vol de travail et mémoires transactionnelles

Quentin L. Meunier

UNIVERSITE DE GRENOBLE

N° attribué par la bibliothèque
978-2-84813-159-7

THÈSE

pour obtenir le grade de

DOCTEUR de L'UNIVERSITÉ DE GRENOBLE
délivré par l'Institut Polytechnique de Grenoble

Spécialité : Informatique

préparée au laboratoire **TIMA**

dans le cadre de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

présentée et soutenue publiquement

par

Quentin MEUNIER

Le 29 Octobre 2010

Titre

Étude de deux solutions pour le support matériel de la programmation parallèle dans les multiprocesseurs intégrés : vol de travail et mémoires transactionnelles

Directeur de thèse : Frédéric PÉTROT

JURY

M.	Philippe CLAUSS	Président
M.	Daniel ETIEMBLE	Rapporteur
M.	André SEZNEC	Rapporteur
M.	Alain GREINER	Examinateur
M.	Jean-Louis ROCH	Examinateur
M.	Frédéric PÉTROT	Directeur de Thèse

Remerciements

Je tiens en premier lieu à remercier mon directeur de thèse Frédéric Pétrot, qui m'a laissé une grande liberté tout au long de ma thèse, mais qui a su être présent aux moments importants. Il m'a témoigné à plusieurs reprises de sa confiance en mon travail, et m'a apporté son soutien lorsque cela était nécessaire.

Je souhaite remercier Olivier Muller, pour son oreille attentive quand j'en ai eu besoin et ses conseils avisés lorsque j'avais un problème à exposer. Sa bonne humeur et ses compétences en font un collègue de choix, avec qui j'ai partagé deux très bonnes dernières années de thèse.

Je tiens à remercier Pierre de Massas qui m'a apporté son soutien moral face aux nombreux bugs que j'ai rencontrés, ainsi qu'une expertise très enrichissante dans le domaine de la simulation et du debug, en particulier au début de ma thèse. Pierre a aussi été toujours à l'écoute de mes soucis ou dilemmes relatifs au matériel, et m'a apporté des conseils de choix.

Je tiens aussi à remercier mes deux rapporteurs, messieurs Daniel Etiemble et André Seznec pour les retours constructifs qu'ils ont apportés sur mon travail.

Je tiens à remercier les membres de l'équipe, en particulier Patrice Gerin, Xavier Guérin, Nicolas Fournel, Damien Hedde, Yan Xu, Florentine "duboiflo" Dubois, Adrien Prost-Boucle et Alexandre Chagoya.

Je tiens enfin à remercier ma petite soeur Tamara pour son soutien durant ma thèse.

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

Vue de loin, l'histoire de l'informatique est une belle histoire : l'informatique a produit des systèmes de plus en plus complexes et paradoxalement de plus en plus accessibles. Sa réussite est telle qu'aujourd'hui, après des décennies de recherche, développement, remise en cause, on peut utiliser un système informatique sans avoir aucune idée de son fonctionnement interne. Ce domaine s'est d'ailleurs progressivement scindé en sous-domaines, chacun de ces sous-domaines devenant à terme une science à part entière : architecture, réseau, système, algorithmique, base de données, interface homme-machine, génie logiciel et bien d'autres. Contrairement à ce que l'on pourrait croire, la particularité de ces sciences est qu'elles sont fondamentalement expérimentales, et *in fine* empiriques, si bien que toutes ont connu leur lot d'échecs (y compris celles censées être des domaines critiques, comme la vérification des programmes).

Cela ne veut pas dire que les procédés de conception ne sont pas bons, mais simplement que concevoir des systèmes informatiques au-delà d'une certaine complexité est une véritable gageure. Pour relever ces défis et pouvoir créer des systèmes toujours plus complexes, l'histoire de l'informatique nous a montré qu'il fallait automatiser le maximum d'étapes et ne laisser à l'utilisateur d'un niveau donné que l'interface la plus simple possible et ne possédant que les degrés de liberté nécessaires à ce qu'il doit réaliser.

Cette thèse vise, dans le cadre des machines multiprocesseurs intégrées, à étudier des interfaces permettant précisément de simplifier le travail du programmeur d'applications parallèles, en lui fournissant un support matériel et des interfaces matériel/logiciel adaptées. Un des buts est de minimiser le surcout induit par cette exigence de simplicité.

Évolution des architectures matérielles

L'intégration VLSI (*Very Large Scale Integration*) continue de suivre la loi de Moore, ce qui a permis d'atteindre sur cette décennie l'intégration de centaines de millions à quelques milliards de transistors sur un substrat de silicium. Cette diminution de la taille s'est accompagnée de plusieurs autres avantages : réduction de la consommation, augmentation en fréquence et intégration possible de systèmes complets sur une seule puce, appelée dans ce cas *SoC* (System-on-Chip). Les difficultés et barrières rencontrées par les concepteurs matériels sont devenues la complexité trop grande des systèmes conçus, notamment en ce qui concerne le sous-système composé du processeur et du cache. À titre d'exemple, les errata pour processeurs disponibles publiquement reportent pour la majorité entre 10 et 100 bogues, par exemple 123 pour le pentium™4 [Int06]. Il n'est donc pas étonnant que des constructeurs comme Intel se soient détournés des architectures basées sur un seul cœur devenues trop complexes et n'offrant plus les performances espérées, comme celle du

Pentium™4, pour repartir d'architectures plus simples (Pentium™3) mais en y intégrant plusieurs cœurs (Core™2).

Cette évolution des architectures amène donc à penser autrement l'écriture des programmes qui doivent alors intégrer une nouvelle dimension : celle du parallélisme. En effet, si l'écrasante majorité de l'informatique a été séquentielle pendant près de 50 ans, cela est en train de changer.

La complexité trop grande des architectures mono-cœur n'est d'ailleurs pas la seule raison qui a amorcé le développement des architectures multicœur : d'une part, la consommation croît fortement avec la détection dynamique de l'ILP (*Instruction Level Parallelism*) ce qui limite l'applicabilité de cette approche, et d'autre part la limite en fréquence des 4GHz sera difficile à dépasser.

Types d'architectures multiprocesseurs

Il existe deux types d'architectures multiprocesseurs : les architectures symétriques (SMP) et les architectures hétérogènes.

Les architectures hétérogènes sont généralement composées d'un processeur généraliste (GPP) qui réalise les tâches de contrôle, et d'un ou de plusieurs co-processeurs, spécialisés dans le calcul intensif d'une tâche (ex : Processeur de traitement de signal – communément appelé DSP – ou VLIW). Ces architectures ont pour avantage d'être plus ciblées vis-à-vis de l'application, et proposent donc des performances élevées pour l'application en question.

À l'inverse, les architectures SMP sont composées d'unités de calcul identiques et partageant un espace mémoire commun. La figure 1.1 montre un exemple d'architecture SMP avec le processeur ARM Cortex™-A9 MPCore, pouvant contenir jusqu'à 4 cœurs.

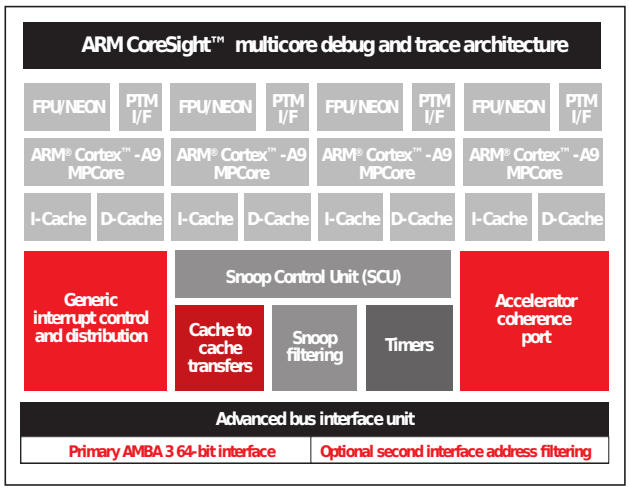
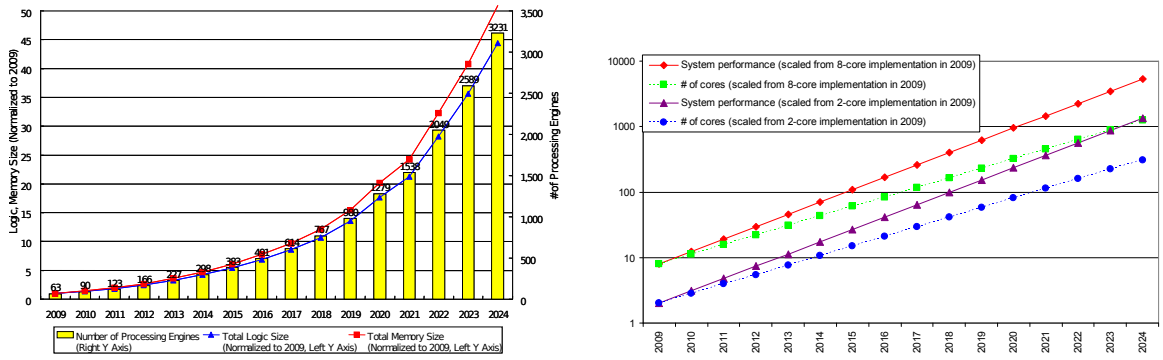


FIG. 1.1 – ARM-Cortex™-A9 MPCore

L'avantage de ces architectures est qu'elles sont plus généralistes, donc portables et plus faciles à exploiter dans la mesure où les mêmes chaines d'outils et systèmes d'exploitation sont utilisés sur tous les processeurs. Par ailleurs, bien que ces architectures ne soient pas optimisées pour un type de logiciel en particulier, elles permettent d'atteindre des bonnes performances comparativement aux architectures mono-cœur, et ont l'avantage d'être génériques. En ce sens, nous pensons que ce sont ces architectures qui seront principalement utilisées dans le futur.

Cette tendance a d'ailleurs déjà commencé : si dans les années passées, la plupart des systèmes embarqués étaient constitués d'un GPP et d'un DSP, il y a une propension à aller vers des architectures SMP contenant une ferme de processeurs homogènes basse consommation (petits GPP ou VLIW dual-issue) et à réaliser les fonctions de calcul intensif en logiciel [DTP+05, WGH+07]. En effet, le fait de réaliser de plus en plus de fonctions par logiciel plutôt que d'utiliser des composants matériels dédiés a un cout en terme de performance, mais ce dernier est compensé par la flexibilité offerte, aussi bien durant la conception du système que pendant sa durée de vie (correction d'erreurs, évolutions des fonctionnalités).

Les prévisions stratégiques du rapport ITRS confirment cette tendance et montrent que ce nombre devrait continuer d'augmenter fortement dans le futur [ITR09]. Les figures 1.2(b) et 1.2(a) qui en sont tirées représentent l'estimation de cette tendance pour deux types de SoCs : ceux dédiés aux réseaux et ceux du domaine grand public.



(a) (b)

FIG. 1.2 – Prévisions ITRS sur nombre de cœurs dans les SoCs grand public (a) et réseaux (b)

Ainsi, les spécifications pour les applications utilisées dans les systèmes électroniques intégrés tendent à être écrites en grande partie comme des programmes parallèles communiquant par l'intermédiaire de mémoire partagée. Ces programmes requièrent en général une cohérence matérielle entre les caches, ce qui est très difficile à assurer dans le cas de systèmes hétérogènes.

Problème général et objectifs de la thèse

Les travaux de cette thèse se placent dans le cadre des problématiques liées à la programmation parallèle. Si les concepts abordés ne se limitent pas systématiquement au cas des architectures homogènes, notre étude se fera avec cette hypothèse d'homogénéité.

Les systèmes sur puce posent de nombreux défis et problématiques, en particulier lorsqu'ils intègrent plusieurs cœurs (on parle alors de *MPSoC*). L'objectif de simplicité à cout acceptable que l'on se fixe est difficilement atteignable dans ce contexte avec les solutions matérielles actuelles. Les problématiques liées au parallélisme étant multiples, nous ciblerons deux points en particuliers. Le premier concerne la manière de décrire le parallélisme du point de vue de l'application. Les paradigmes de programmation parallèle usuels sont les threads et le passage de message. Une proposition relativement peu exploitée aujourd'hui est le vol de travail, que nous étudierons du point de son interface matériel/logiciel. Dans un premier temps, nous porterons donc notre attention sur une bib-

liothèque basée sur le concept de vol de travail, nommée AWS, qui permet une description du parallélisme.

Dans un second temps, nous nous concentrerons sur le paradigme de programmation avec les systèmes de mémoire transactionnelle. Ce paradigme de programmation permet d'ajouter un niveau d'abstraction à la synchronisation entre les threads, rendant plus facile la conception des programmes parallèles. Il requiert néanmoins pour espérer être efficace un support matériel non négligeable.

Plan de la thèse

Le deuxième chapitre présente les problématiques auxquelles nous nous attaquerons dans cette thèse.

Le chapitre 3 est un état de l'art sur le vol de travail. Il présente les notions de base de ce domaine, ainsi que les principaux travaux existants qui s'y rapportent. Le chapitre 4 présente notre contribution en matière de vol de travail dans le cadre des systèmes embarqués. Y sont aussi présentés les expérimentations et résultats afférents.

Le chapitre 5 dresse un constat des travaux ayant été effectués dans le domaine des systèmes de mémoire transactionnelle (TM). Le chapitre 6 décrit la conception et l'évaluation de deux systèmes TM matériels basés sur des protocoles de cohérence de cache différents.

Le chapitre 7 présente plusieurs variantes de systèmes TM et les compare d'un point de vue performances. La question de la robustesse de ces systèmes face à des cas pathologiques y est aussi abordée.

Les limitations de ce travail ainsi que les axes de recherche en vue de travaux futurs sont présentés dans le chapitre 8.

Enfin, le dernier chapitre conclut en répondant aux questions posées dans la problématique, et ce en se basant sur les contributions apportées. Il résume l'ensemble des résultats et des apports de ce travail de thèse.

Chapitre 2

Problématique

Nous présentons dans ce chapitre le contexte général de notre étude, centrée sur les systèmes multiprocesseurs. Ces derniers doivent faire face à plusieurs défis, un des plus importants étant celui de la programmation parallèle. Nous nous intéresserons aux contraintes, garanties et cout des différentes techniques de programmation parallèle aux différents niveaux hiérarchiques d'un système, et ce par l'intermédiaire de deux concepts : le vol de travail et les mémoires transactionnelles.

2.1 Évolution des architectures matérielles

Comme évoqué dans l'introduction, les architectures matérielles connaissent actuellement une évolution vers le multicœur, i.e. le fait d'avoir plusieurs cœurs sur une même puce.

On distingue deux types d'architectures multicœur : les systèmes-sur-puce multiprocesseurs (MPSoCs) et les puces multiprocesseurs (CMP). Les MPSoCs intègrent à la fois des processeurs, des unités de calcul spécialisées et des périphériques, et de ce fait sont utilisés principalement dans le domaine de l'embarqué. Les CMPs sont à l'inverse principalement utilisés dans les serveurs et les machines personnelles de bureau, car ils intègrent sur la même puce des processeurs généralistes et puissants, ainsi que typiquement deux niveaux de cache.

2.1.1 Différences entre les MPSoCs et les CMPs

On pourrait par ailleurs croire que les architectures MPSoC suivent la même voie que les CMPs car elles tendent à être de plus en plus génériques - du fait du cout des masques, du besoin d'évolution rapide des applications et du support pour de multiples applications. Cependant, ce n'est pas réellement le cas pour plusieurs raisons.

2.1.1.1 Consommation d'énergie

Premièrement, la consommation d'énergie pour les CMPs est au moins un ordre de grandeur au-dessus de celles des MPSoCs - souvent sur batteries et sans ventilateur. Pour satisfaire les besoins en performance malgré les contraintes de consommation, l'idée est d'utiliser beaucoup de processeurs ayant un ratio performance/Watt élevé, de les alimenter à basse tension, et de les faire tourner à une fréquence adaptée à ce voltage. Les processeurs RISC simples ou les VLIW à 2 ou 4 instructions simultanées sont en général de bons compromis.

2.1.1.2 Classes d'applications

Les classes d'applications ou les cas d'utilisation pour les MPSoCs sont connus au moment de la conception. En conséquence, ces derniers ne visent pas une généricité totale, et des choix ou optimisations spécifiques à la classe d'applications peuvent aussi être faits au moment de la conception.

2.1.1.3 Distribution de la mémoire

Troisièmement, le temps de mise sur le marché et le cout sont des problèmes importants pour les systèmes destinés à des marchés de masse. Même si la plupart des architectures embarquées n'ont aujourd'hui pas un espace d'adressage partagée, les codes dits "legacy" rendent attractif le support du modèle de programmation à mémoire partagée. En prenant cela pour hypothèse, le choix en général retenu dans les MPSoCs est d'avoir une mémoire adressable plutôt que des caches L2 dans la puce. De manière à pouvoir assurer assez de bande passante mémoire à basse fréquence, une technique couramment employée est de distribuer physiquement la mémoire sur la puce, en supposant que le moyen d'interconnexion ne soit pas un goulot d'étranglement. Cela signifie que la mémoire n'est pas centralisée mais que plusieurs bancs mémoire sont présents et se partagent l'espace adressable. On parle alors de mémoire logiquement partagée et physiquement distribuée. L'idée d'avoir des NoCs, introduite en 1999 [GG00], permet d'éviter ce problème de congestion des accès au cout de nécessiter un mécanisme de cohérence des caches basé sur un répertoire.

2.2 Cache et cohérence en milieu multiprocesseur

Dans le cadre de nos travaux, nous utiliserons de la mémoire cache. Cette section survole les contraintes relatives à l'utilisation des caches dans un contexte multiprocesseur.

2.2.1 Les caches, composants indispensables des architectures

Bien que les caches occupent une proportion élevée de la surface disponible dans une puce, leur efficacité est telle qu'ils demeurent le moyen principal de réduire le temps d'accès à la mémoire, y compris dans les systèmes embarqués. Néanmoins, avec l'arrivée des architectures multiprocesseurs, l'utilisation des caches a introduit une contrainte supplémentaire en matière de cohérence des données puisqu'ils impliquent de potentielles réplications multiples d'une donnée mémoire.

2.2.2 Cohérence des données

La cohérence des données dans un système multiprocesseur avec des mémoire cache peut être classée selon deux grandes catégories : gestion logicielle ou gestion matérielle. Avec une gestion logicielle, le programmeur a pour responsabilité d'invalider lui-même les lignes de caches lorsque cela est nécessaire afin de garantir la cohérence. Avec une gestion matérielle, la cohérence est garantie de manière transparente du point de l'utilisateur puisqu'elle est délaissée au système matériel sous-jacent.

Bien sûr, le cout de la gestion matérielle de la cohérence n'est pas nul et impacte la complexité et la surface des contrôleurs des caches et des mémoires. Néanmoins, en dépit de ce surcout, c'est généralement cette solution qui est retenue pour des raisons de simplicité d'un point de vue programmation.

On distingue deux familles de protocoles de cohérence : les protocoles à *écriture simultanée* (*write-through*), pour lesquels toutes les écritures sont propagées en mémoire et les protocoles à *écriture différée* (*write-back*), pour lesquels les écritures sont locales aux caches et recopiées en mémoire uniquement lorsque cela est nécessaire.

2.2.3 Systèmes embarqués et protocole de cohérence

Dans les CMPs, les protocoles de cohérence de cache les plus répandus sont les protocoles à écriture différée. En effet, le grand nombre d'écritures vers la mémoire des protocoles à écriture simultanée induit une consommation élevée de la bande passante du bus, de quoi résulte une baisse des performances.

Dans les MPSoCs ancienne génération utilisant un bus, cela était aussi généralement le cas, car en plus de la bande passante, le grand nombre d'écritures vers la mémoire des protocoles à écriture simultanée impacte négativement la consommation du système.

Néanmoins, avec l'introduction des réseaux-sur-puce (NoCs), le problème de bande passante n'est plus aussi crucial qu'avant. Par ailleurs, les protocoles à écriture simultanée s'avèrent être plus légers : ils économisent quelques bits par ligne de cache et par ligne mémoire, et simplifient grandement les machines d'états finies des caches et mémoires. Cela résulte du fait que les situations de compétition entre les requêtes de données et les requêtes d'invalidation sont plus simples à traiter en écriture simultanée qu'en écriture différée.

Comme chacun des types de protocole présente des avantages, nous considérerons dans nos analyses matérielles alternativement l'une et l'autre de ces solutions.

2.3 La programmation parallèle, une nécessité

Les changements dans les architectures impliquent des changements dans la façon de programmer. L'avènement des architectures multicœur implique des modifications dans la façon d'écrire les programmes puisqu'ils doivent alors être écrits de manière parallèle. En particulier, il n'y a aucun moyen automatique de rendre parallèle un algorithme séquentiel dans le cas général.

2.3.1 Problème général

Le problème le plus général relatif à la programmation parallèle et auquel nous tenterons d'apporter une contribution est le suivant : comment avoir des algorithmes parallèles performants et corrects ?

D'un point de vue global, on peut distinguer cinq couches qui vont décider de l'efficacité et de la correction d'un algorithme parallèle :

1. L'algorithme ; dans nos travaux, nous ne nous intéresserons à cet aspect que de manière marginale.
2. Les bibliothèques ou constructions utilisées pour l'écriture du programme. En effet, disposer d'un bon algorithme n'est pas toujours suffisant pour avoir une exécution efficace : la création des tâches, leur assignation sur les cœurs et leur synchronisation sont des éléments pouvant avoir de grandes répercussions sur les performances globales, mais aussi sur la capacité à écrire des programmes corrects ou erronés.
3. Le modèle de programmation ; il définit les abstractions utilisées pour le parallélisme. Ses conséquences impactent principalement la correction des programmes. Modèles de programmation et bibliothèques sont intimement liés puisqu'une bibliothèque va

fournir à un utilisateur un ou plusieurs modèles de programmation, et elle-même utiliser un modèle de programmation pour son écriture, en général différent.

4. Les primitives fournies par le matériel : elles peuvent avoir des conséquences sur la concurrence des accès aux données, et donc sur l'efficacité, ainsi que sur le modèle de programmation utilisé dans la couche supérieure.
5. Les ressources matérielles de calcul : il s'agit principalement du nombre de cœurs et de leur efficacité.

Dans la suite, nous allons nous intéresser plus en détail à chacun de ces points, et en particulier au niveau des implications qui découlent du problème global défini au-dessus.

2.3.2 Parallélisme intrinsèque d'un algorithme, loi d'Amdahl

Chaque algorithme possède un parallélisme intrinsèque, qui est le degré maximal de parallélisme que l'on peut en tirer. Ce parallélisme intrinsèque limite l'accélération (*speedup*) que l'on peut obtenir en le parallélisant. La loi d'Amdahl dit simplement que si p est la proportion du temps parallélisable d'un algorithme, l'accélération maximale S que l'on peut avoir avec N processeurs est de :

$$S = \frac{1}{(1-p) + \frac{p}{N}}$$

De plus, cette loi, qui donne globalement une tendance, ne prend en compte ni le cout de transfert des données, ni le cout de la synchronisation, qui grandit à mesure que le nombre de processeurs augmente.

Ainsi, la qualité d'un algorithme parallèle se mesure par la proportion du temps passé dans des sections parallèles lorsque ce dernier est exécuté sur un seul processeur. C'est pourquoi pour résoudre certains problèmes de manière parallèle, il peut être plus efficace d'utiliser un algorithme différent de celui utilisé pour le séquentiel.

En conséquence, pour qu'un algorithme soit performant, il faut tenter de maximiser la concurrence des accès aux données partagées et de minimiser le nombre et la taille des sections critiques. Or, cela va généralement rendre le programme plus complexe, et avoir des conséquences sur sa correction.

2.3.3 Ressources matérielles de calcul

De manière assez évidente, on peut espérer que l'exécution d'un algorithme parallèle sera meilleure à mesure que l'on augmente le nombre cœurs (ou processeurs), et que ceux-ci sont performants. Néanmoins, jouer sur ces paramètres n'est pas toujours possible pour des raisons de cout [HM08].

Au niveau de la correction des algorithmes, il n'y a pas grand chose sur lequel on puisse influer à ce niveau.

Néanmoins, l'exploitation de ces ressources matérielles constitue l'articulation critique de notre problème. De fait, seuls peu de programmes écrits arrivent à faire une utilisation efficace des ressources, et jusqu'à récemment, le parallélisme était en général limité à de larges charges de travail s'exécutant en parallèle sur plusieurs machines. Par opposition, les programmes parallèles ciblant les architectures multicœur font face à des contraintes très différentes, notamment en termes de latence et de bande passante. Aussi, nous allons porter notre attention sur les 3 points intermédiaires : les bibliothèques utilisées pour l'écriture des programmes, les modèles de programmation et les primitives fournies par le matériel.

2.4 Librairies et constructions de haut-niveau pour l'écriture des programmes parallèles

Souvent, les programmes parallèles sont écrits en utilisant des bibliothèques (OpenMP, MPI) qui proposent une interface de haut-niveau pour la description des tâches ou du travail. L'avantage d'utiliser de telles bibliothèques par rapport à une synchronisation faite à la main est qu'elles facilitent l'obtention d'une synchronisation correcte entre les tâches.

Les questions que l'on se pose alors à ce niveau sont les suivantes : l'utilisation de ces constructions de haut-niveau permet-elle d'avoir une exploitation efficace des ressources matérielles ? Quelles en sont par ailleurs les contraintes au niveau de la programmation ?

2.4.1 Description du travail et communication entre tâches

Se pose donc tout d'abord la question du moyen employé pour décrire le travail à effectuer. La plupart des algorithmes utilisés actuellement dans les MPSoCs sont essentiellement implémentés sous une des formes suivantes : (a) sous la forme de tâches séquentielles gros grain communiquant par l'intermédiaire de fifos sans-pertes, (b) de représentations par flots de données synchrones pour lesquels des ordonnancements optimaux peuvent être statiquement calculés, ou (c) des réseaux de processus de Kahn qui ont la propriété d'avoir une sortie indépendante de l'ordonnancement tout en étant moins contraints que les formalismes précédents. Cette tendance a été adoptée par plusieurs acteurs de l'industrie de l'électronique de manière à pouvoir bénéficier des propriétés de ces modèles [Dur06]. D'autres approches sont moins restrictives, et utilisent des sous-ensembles de bibliothèques de programmation parallèle pour lesquelles les propriétés et le modèle de programmation sont moins claires : MPI [AKS06], versions légères de Corba [PPB02], OpenMP [OKK03, BvLR+08], ou même de simples threads communiquant par mémoire partagée.

2.4.2 Le vol de travail adaptatif

Un autre paradigme de programmation parallèle consiste à avoir un ordonnancement basé sur le vol de travail [ABP01, BR02]. De tels algorithmes reposent sur le principe que chaque processeur exécute sa propre tâche jusqu'à devenir inactif, puis essaie de voler une fraction du travail restant sur un processeur actif choisi au hasard.

Les algorithmes par vol de travail exhibent un bon comportement en pratique quand la charge de travail ne peut pas être bien estimée *a priori* [MKH91, FLR98, Pap98, TRM+08], ce qui est le cas pour les algorithmes de type encodage ou compression, souvent utilisés dans les systèmes embarqués destinés au grand public (*consumer electronics*). Ces algorithmes sont aussi adaptés aux applications s'exécutant sur les plateformes embarquées sur lesquelles tous les processeurs ne sont pas cadencés à la même vitesse, créant ainsi une charge de travail déséquilibrée.

Comme il se peut que cette approche soit adaptée aux architectures intégrées massivement multiprocesseurs développées dans l'industrie, la question que l'on se pose est la suivante : ce modèle de programmation pourrait-il convenir à certaines applications MPSoCs typiques ? Quelles sont les contraintes sur l'algorithme à paralléliser et quelles en sont les garanties ?

Par ailleurs, si l'on souhaite avoir des performances efficaces, quelles doivent être les caractéristiques d'une architecture matérielle spécifique à ce type d'algorithme ?

2.5 Modèles de programmation et interface avec le matériel

Les bibliothèques ou interfaces permettant de décrire le parallélisme reposent sur des primitives ou mécanismes de bas niveau (ex : verrous, ou *locks*). Il s'agit souvent de systèmes en étroite relation avec le matériel, c'est pourquoi nous présentons ces deux aspects dans une section commune.

Un des défis principaux de la programmation parallèle consiste en l'exploitation effective des ressources de calcul disponibles. Or, il s'avère qu'écrire des programmes parallèles est quelque chose d'intrinsèquement difficile - notre apprentissage de la programmation reste essentiellement séquentiel -, et pour un programmeur donné, il existe une différence importante entre l'écriture d'un algorithme séquentiel performant et l'écriture d'un algorithme parallèle performant.

Ainsi, s'il ne suffit pas d'avoir des ressources de calcul pour augmenter la vitesse d'exécution d'un algorithme, il n'est pas non plus suffisant d'avoir une bonne description parallèle de l'algorithme : il faut qu'il y ait une adéquation entre l'implémentation de haut niveau, le modèle de programmation utilisé et les mécanismes de bas niveau mis à disposition par le système.

Les défis que posent ces aspects sont donc : avec quelles abstractions et quels mécanismes bas niveau peut-on exploiter efficacement les ressources matérielles d'un système ? De la même manière, peut-on favoriser l'écriture de programmes corrects en jouant sur le modèle de programmation ? Comment simplifier le modèle de programmation tout en gardant une concurrence élevée dans l'accès aux données ?

2.5.1 Threads et verrous, éléments indispensables d'abstraction et de synchronisation

Cet avènement du paradigme de programmation parallèle à mémoire partagée a donc fait ressortir le besoin d'abstractions pour pouvoir exploiter le parallélisme. Actuellement, l'abstraction la plus utilisée est celle des threads : ils permettent d'écrire un programme sans se soucier du nombre de cœurs sur lequel ce dernier va réellement s'exécuter, et peuvent accéder à des données partagées. Les programmes multithreadés sont synchronisés au moyen de verrous, basiques ou plus évolués (sémaphores, moniteurs), qui présentent deux avantages principaux : une facilité d'implantation en matériel et une sémantique simple. De fait, ils sont le moyen principalement utilisé pour la synchronisation entre threads et l'implantation des sections critiques.

2.5.2 Problèmes posés par les verrous

Si les verrous jouent un rôle majeur dans la synchronisation des programmes parallèles, l'expérience a montré que les programmes à base de verrous étaient difficiles à implémenter, déboguer et maintenir. Si la sémantique individuelle du verrou est simple, les différents entrelacements pouvant résulter d'une exécution ne sont pas toujours faciles à appréhender. De plus, ils doivent en général faire face à un obstacle supplémentaire en termes de performances : celui du choix de la granularité des sections critiques. En effet, faire des sections critiques larges réduit pour beaucoup les performances en augmentant de manière considérable la congestion ; mais à l'inverse, faire des sections critiques petites rend les problèmes de synchronisation encore plus délicats, et expose le programme à un risque d'erreurs plus élevé, ainsi qu'aux étreintes mortelles (*deadlocks*).

2.5.3 La remise en cause des verrous

Pour les raisons évoquées, des alternatives aux verrous ont été recherchées, dans le but de pouvoir fournir au programmeur un modèle de programmation garantissant certaines propriétés - notamment l'absence d'étreintes mortelles - et des performances meilleures, en essayant de maximiser la concurrence des accès aux données. D'autres abstractions toujours basées sur les threads ont ainsi vu le jour, et concernent la synchronisation entre ces threads.

2.5.3.1 Le modèle de programmation lock-free

La première alternative aux verrous est le modèle de programmation *lock-free*. Ce dernier utilise des primitives non bloquantes et tente de corriger les problèmes liés à l'utilisation des verrous, mais souffre pour sa part d'autres défauts majeurs : (1) une complexité conceptuelle importante : les algorithmes lock-free sont souvent difficiles à écrire et contre-intuitifs (2) des performances faibles dans le cas général : si certains algorithmes spécifiques relatifs à des types de données particuliers sont plus efficaces que les implémentations à base de verrous, la grande majorité des algorithmes lock-free sont faits à partir de transformations systématiques, desquelles résultent des performances bien en deçà des équivalents à base de verrous (3) enfin, un support matériel minimal est requis.

2.5.3.2 CAS et CAS-n

Les primitives principalement utilisées par le modèle de programmation lock-free reposent toutes sur des variantes de l'instruction Compare-and-Swap (CAS). La sémantique de cette instruction est illustrée figure 2.1(a).

```
bool CAS(T *address, T
old_val, T new_val){
   <atomic>
   if (*address == old_val){
      *address = new_val;
      return true;
   }
   else {
      return false;
   }
   <end atomic>
}
```

(a) Sémantique de la primitive CAS en pseudo-code

```
bool CAS2(T *addr1, T *addr2, T old1, T
old2, T new1, T new2){
   <atomic>
   if (*addr1 == old1 && *addr2 == old2 ){
      *addr1 = new1;
      *addr2 = new2;
      return true;
   }
   else {
      return false;
   }
   <end atomic>
}
```

(b) Sémantique de la primitive CAS-2 en pseudo-code

FIG. 2.1 – Sémantique des primitives CAS et CAS-2 en pseudo-code

Principalement deux types d'améliorations existent pour cette instruction : la meilleure consiste à pouvoir opérer un CAS sur un nombre de mots supérieur à 2. On parle alors de CAS-n. La sémantique du CAS-n pour $n = 2$ est illustrée figure 2.1(b). Cette instruction est couramment nommée CAS-2 ou DCAS (pour Double CAS). Le second type d'amélioration,

plus faible, consiste à pouvoir effectuer des CAS sur des mots de plus grande taille ; ces instructions peuvent être vues comme des CAS-n avec la restrictions que les mots sur lesquels ils opèrent doivent être contigus. Si l'instruction CAS opérant sur 2 mots est relativement répandue dans les jeux d'instructions des processeurs, l'instruction CAS-2 n'a vu le jour que sur très peu d'architectures. L'instruction CAS-n pour $n > 2$ n'existe dans aucune architecture comme étant directement implantée en matériel.

Malheureusement pour ce modèle de programmation, la primitive CAS seule, même opérant sur deux mots, reste légère pour beaucoup d'algorithmes. Selon [GCPB99], la primitive DCAS est nécessaire pour que ce modèle de programmation puisse avoir un nombre plus élevé d'implémentations de structures de données efficaces. Or, cela est loin d'être le cas, et pour qu'elle se généralise, il aurait fallu que ce modèle de programmation ne soit pas si difficile à utiliser.

2.5.4 Le modèle de programmation transactionnel

Une autre alternative aux verrous est le modèle de programmation transactionnel, introduit en 1977 par Lomet [Lom77]. Ce modèle définit une transaction comme étant un ensemble d'instructions réalisées de manière atomique du point de vue des autres tâches s'exécutant de manière concurrente. Cela offre une abstraction de haut niveau pour écrire des programmes parallèles, puisqu'un programmeur peut ainsi raisonner sur la correction de son code à l'intérieur d'une transaction sans se soucier des interactions possibles avec les autres transactions.

Issue du domaine des bases de données, la sémantique des transactions mène à quatre propriétés, connues sous le nom de "propriétés ACID" : Atomicité, Consistance, Isolation et Durabilité.

Le modèle de programmation transactionnel restreint la notion de transaction à deux seulement de ces propriétés : atomicité et isolation. L'atomicité garantit que les changements d'états du programme résultant de l'exécution du code contenu dans une transaction sont indivisible du point de vue des autres threads. En d'autres termes, si une transaction modifie plusieurs variables à travers une série d'affectations, un autre thread ne peut observer que l'état immédiatement avant ou immédiatement après la transaction, mais pas un état intermédiaire. L'isolation garantit que les tâches s'exécutant de manière concurrente ne peuvent pas affecter le résultat d'une transaction allant à son terme (dans le cas où ces dernières ne sont pas en conflit). Ainsi, une transaction doit produire le même résultat que si aucune autre tâche ne s'exécutait en même temps.

2.6 Les systèmes de Mémoire Transactionnelle

Le modèle de programmation transactionnel est actuellement sujet à d'actives recherches du fait de son haut niveau d'abstraction. En effet, il décharge le programmeur du problème d'une synchronisation correcte entre tâches en le transférant au système sous-jacent, ce qui signifie que le programmeur n'a plus à se soucier des étreintes mortelles possibles, et de la granularité des sections critiques.

De tels systèmes portent le nom de "Systèmes de Mémoire Transactionnelle (TM)". L'implantation d'un tel système a pour la première fois été évoquée en 1993 [HM93].

Les systèmes TM peuvent être implémentés en logiciel [FH07, HLMS03, MSS05, ST97] ou implantés en matériel [MBM+06a, HWC+04, RHL05].

Une autre catégorie de systèmes TM dite hybride se situe à cheval entre le logiciel et le matériel [MTC+07, VHC+08, BNZ08]. Parmi les systèmes hybrides, on distingue entre

autres les systèmes logiciels avec accélération matérielle pour certaines actions.

A ce stade, la question que l'on se pose est la suivante : le modèle de programmation transactionnel apporte-t-il suffisamment d'avantages et de garanties, et est-il implantable à un cout tel qu'il soit envisageable d'utiliser un système TM sur un système embarqué ?

2.6.1 Les systèmes de Mémoire Transactionnelle logiciels (STM)

Les systèmes TM logiciels se démarquent par le fait qu'ils sont basés sur des primitives lock-free existantes qui ne requièrent donc pas d'autre matériel particulier que celui pour supporter les primitives elles-mêmes, par exemple un simple CAS. Ces systèmes sont en général implémentés sous la forme d'une bibliothèque basée sur ces primitives simples, et offrent à l'utilisateur des primitives plus évoluées, allant du CAS-n à de vraies transactions.

Le principal avantage des systèmes STM est qu'ils sont donc d'ores et déjà utilisables sur les machines actuelles.

En revanche, ces systèmes souffrent d'une utilisation souvent complexe : les sections critiques remplacées par des transactions doivent contenir le détail des accès pour la transaction. Un exemple de parallélisation d'un programme via le système WSTM (présenté dans [FH07]) est donné figure 2.2(a), et illustre notamment le fait que les transformations à faire demandent à l'utilisateur de spécifier les lectures et les écritures pouvant entrer en conflit avec une autre transaction. Cela est d'autant plus contraignant que dans ce cas, une parallélisation avec un verrou serait très simple à écrire.

```
typedef struct _node { int key; struct _node* nxt; } node;
typedef struct { node* head; } list;

void insert(list * l, int k){
   node *n := new node(k);

   node *prv := l->head;
   node *cur := prv->nxt;
   while (curr->key < k){
      prv := cur;
      cur := cur->nxt;
   }
   n->nxt := cur;
   prv->nxt := n;

}
```

(a) Algorithme séquentiel

```
void insert(list * l, int k){
   node *n := new node(k);
   wstm_transaction *tx;
   do {
      tx = WSTMStartTransaction();
      node *prv := WSTMRead(tx,&(l->head));
      node *cur := WSTMRead(tx,&(prv->nxt));
      while (cur->key < k){
         prv := cur;
         cur := WSTMRead(tx, &(cur->nxt));
      }
      n->nxt := cur;
      WSTMWrite(tx, &(prv->nxt), n);
   } while (!WSTMCommitTransaction(tx));
}
```

(b) Algorithme parallèle pour un système STM

FIG. 2.2 – Exemple d'insertion dans une liste chainée triée : algorithme séquentiel et parallèle pour un système STM

Par ailleurs, ces systèmes sont affectés par de gros problèmes de performances comparés aux autres formes de parallélisation, y compris celles utilisant des verrous [CBM+08], ce qui constitue un frein important à leur essor.

2.6.2 Les systèmes TM matériels (HTM)

À l'inverse des systèmes STM, les systèmes entièrement HTM sont basés sur le fait qu'aucune instrumentation logicielle spécifique n'est requise, mais que le système est entièrement implanté au niveau matériel. Bien souvent, ces systèmes proposent une interface restreinte contenant principalement deux primitives qui indiquent respectivement le début et la fin d'une transaction.

Les nombreux systèmes HTM existants, tous au niveau conceptuels, se différencient les uns des autres sur un grand nombre de points. Entre autres, on peut noter des différences dans les hypothèses au niveau du matériel requis, au niveau du système d'exploitation (en particulier au niveau de la virtualisation des transaction par l'intermédiaire du système), de l'efficacité visée, de la généricité et de la portabilité, et des limitations concernant la taille des transactions.

Cependant, dans les nombreuses implantations matérielles des systèmes HTM, la supposition d'un protocole de cohérence de cache à écriture différée est toujours faite de manière explicite ou implicite, et il n'y a eu aucune étude sur l'influence du protocole de cohérence de cache sur les performances des mémoires transactionnelles.

Cela est dû au fait que la plupart des systèmes TM ne sont pas dédiés au SoCs. Aussi, dans notre tentative de réponse à la question d'embarquer un système TM sur un SoC, nous nous poserons la question suivante : cette hypothèse est-elle justifiée ? Est-il possible d'écrire un système TM basé sur un protocole de cohérence à écriture simultanée ? Auquel cas quelles en sont les performances par rapport à un système basé sur un protocole à écriture différée ? Par ailleurs, quelles sont les difficultés liées à l'implantation un système TM autour d'un NoC ?

Comme expliqué précédemment, ce travail est motivé par le fait nous visons le domaine des architectures intégrées multiprocesseurs utilisant des NoCs, et que dans de telles architectures, un protocole de cohérence à écriture simultanée requiert moins de logique et est moins complexe qu'un protocole à écriture différée pour des performances similaires [dMP08]. Cependant, l'écriture simultanée peut mener à plus de trafic mémoire et est *a priori* moins adapté à un protocole transactionnel du fait de la nature exclusive d'une ligne dans une transaction.

D'un point de vue interne, les systèmes TM matériels se distinguent les uns des autres sur les moyens utilisés pour garantir les propriétés d'atomicité et d'isolation. Nous essaierons donc de répondre aux questions suivantes : quelles sont les politiques qui affichent les meilleures performances ? Quels en sont les couts d'implantation matérielle ?

Enfin, les systèmes TM sont sujets à des comportements dits pathologiques. Ces pathologies affectent les performances, et dans le pire des cas peuvent mener à une étreinte active (*livelock*). Ainsi, nous essaierons de voir quelles sont les conséquences du choix des politiques sur les pathologies : certaines politiques sont-elles plus sensibles aux pathologies ? D'autres peuvent-elles garantir une absence d'étreinte active ?

2.7 Contexte d'étude

Pour traiter les différentes questions évoquées dans cette problématique, nous allons utiliser des modèles de plateformes présentant des caractéristiques propres au cadre de l'étude.

Ainsi, nos travaux se placent dans le contexte des architectures multiprocesseurs visant le domaine des MPSoCs. Elles présentent donc les aspects suivants :

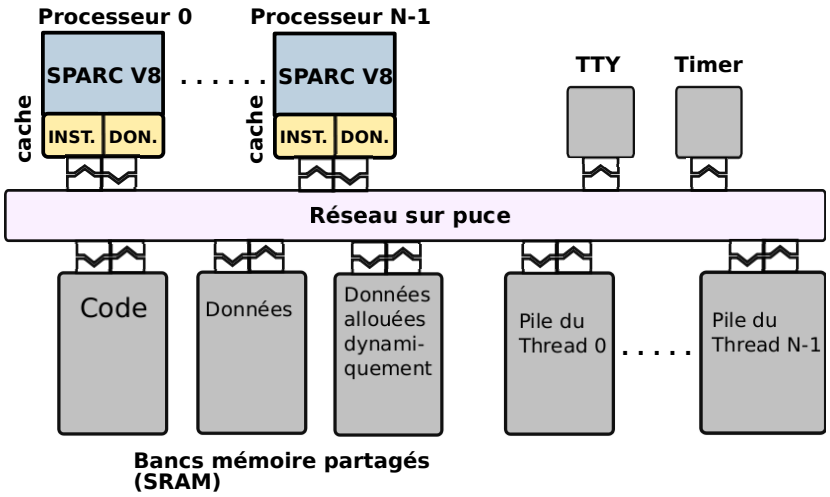


FIG. 2.3 – Exemple d'architecture qui sera utilisée dans les expérimentations

- **Homogènes et symétriques.** Les processeurs utilisés dans nos plateformes sont simples, génériques et identiques. Il s'agira dans notre cas de processeurs SPARC.
- **Caches.** Les plateformes possèdent des caches de niveau L1, mais pas de cache de niveau L2. La cohérence est maintenue grâce à un protocole de cohérence implanté en matériel. Dans certaines expérimentations, les caches de données seront néanmoins désactivés et remplacés par des mémoires locales.
- **Distribution de la mémoire.** La mémoire sera souvent distribuée physiquement, de manière uniforme. Néanmoins, dans certains cas, elle sera centralisée pour permettre l'évaluation de l'impact de la congestion sur les performances.
- **Accès aux bancs mémoire.** Tous les bancs mémoire sont accessibles par tous les processeurs, qui voient donc le même espace d'adressage.
- **Réseau d'interconnexion.** Les réseaux d'interconnexion utilisés sont de type Réseau-sur-puce (NoC), permettant à différentes requêtes de transiter de manière simultanée sur la puce.
- **Hiérarchie.** Souvent, la hiérarchie utilisée sera à plat, i.e. tous les processeurs et mémoires seront connectés au même composant d'interconnexion. Certaines expérimentations utiliseront cependant deux niveaux de hiérarchie : un premier niveau avec un réseau local, un processeur, une mémoire et quelques autres périphériques, et un deuxième niveau reliant tous ces nœuds.

Une architecture typique qui sera utilisée est représentée figure 2.3.

2.8 Conclusion

Nous venons de soulever plusieurs problèmes relatifs à la programmation parallèle. Les deux problèmes principaux sont : comment faire pour avoir des programmes parallèles performants ? Et comment faciliter l'écriture de programmes parallèles corrects ?

Ces problèmes généraux vont être abordés de deux points de vue différents : du point de vue de l'interface utilisée avec la mise en œuvre et la modification d'une bibliothèque de vol de travail et l'analyse d'une architecture adaptée à son support ; et du point de vue des constructions élémentaires de synchronisation, par le biais de systèmes de mémoire transactionnelle.

Les problèmes que nous allons attaquer dans cette étude sont donc les suivants :

- Le vol de travail est-il un moyen d'écriture de programmes parallèles adapté pour le domaine de l'embarqué ?
- Quelles sont les caractéristiques matérielles permettant d'avoir une exécution efficace d'un algorithme basé sur le vol de travail ?
- Est-il envisageable de concevoir un système HTM basé sur un protocole de cohérence à écriture simultanée dans une architecture à base de NoCs ? Pour quelles performances ?
- Quelles politiques des systèmes HTM donnent les meilleurs résultats ? À quels couts matériels ?
- Quelles sont les garanties que l'on peut obtenir en termes de progression du système et d'absence de pathologies pour ces différentes politiques ?

Chapitre 3

État de l'art sur le support matériel au vol de travail

Ce chapitre présente les notions de base du vol de travail et décrit les principaux travaux existants du domaine. Ce paradigme de programmation parallèle est apparu dans le but d'exploiter au mieux les ressources matérielles disponibles, utilisant une abstraction de nœud de calcul. Il pose néanmoins une contrainte sur la manière d'écrire le programme : dans sa version classique, le vol de travail requiert une décomposition de l'application parallèle en tâches – usuellement des fonctions réentrantes –, dont les dépendances peuvent être vues comme un graphe orienté sans cycle. Cette restriction apporte cependant un avantage, puisque les synchronisations entre tâches sont en partie gérées par le système logiciel, ce qui limite l'écriture de programmes incorrects.

Dans ce contexte, nous cherchons à étudier l'intérêt de la mise en œuvre de mécanismes matériels en vue d'en améliorer les performances.

3.1 Le paradigme du vol de travail

Le vol de travail est un paradigme d'ordonnancement pour les calculs parallèles. Il s'agit d'un ordonnanceur décentralisé [ALHH08] : à chaque fois qu'un processeur n'a plus de travail à faire, il en vole à un processeur choisi au hasard. Depuis l'implémentation de référence du vol de travail Cilk [FLR98], l'intérêt pour le vol de travail sur les architectures multicœur n'a cessé de croitre.

3.1.1 Intérêt pour le vol de travail

En Juillet 2007, Intel a lancé un projet open source nommé TBB (pour *Threading Building Blocks*) consistant à créer un ensemble de primitives C++ de haut-niveau à la manière de la bibliothèque de templates standards (STL), mais pour les algorithmes parallèles et avec des containers *thread-safe*, i.e. garantissant la cohérence des données même dans un environnement multithreadé. Créée en Avril 2007 par Frigo et Leiserson, la start-up Cilk Arts fournit depuis mi 2008 une extension parallèle de Cilk en C++ (Cilk++) pour permettre la programmation portable des machines multicœur.

3.1.2 Implémentations du vol de travail à base de deque

Blumofe et Leiserson [BL94] ont proposé un algorithme pour le vol de travail basé sur des files à deux accès (deques). Chaque processeur possède une deque de tâches qu'il utilise

comme une pile pour ses propres tâches (LIFO) : il empile les tâches qu'il crée ou débloque en bas de la deque, et lorsqu'il complète une tâche, il dépile une nouvelle tâche du bas de la deque si elle n'est pas vide. Dans le cas contraire, le processeur passe en attente et devient un voleur : il envoie une requête de vol à un processeur choisi au hasard – appelé victime –, jusqu'à trouver une victime avec une deque non vide. Il prend alors la tâche située en haut de la deque de la victime.

3.1.3 Performances du vol de travail

Des bornes sur les performances du vol de travail peuvent être prouvées. Arora, Plaxton et Blumofe [ABP01] ont montré que pour n'importe quel programme parallèle utilisant Cilk, le temps T_p sur p processeurs identiques vérifie avec une grande probabilité (w.h.p) : $T_p \leq O(\frac{T_{seq}}{p} + T_\infty)$ où T_{seq} est le temps d'exécution séquentiel (qui correspond au travail à faire), et T_∞ à la profondeur maximale (i.e. le temps d'exécution sur un nombre de processeurs non borné). Avec de légères variantes, des bornes similaires peuvent être atteintes lorsque le nombre de processeurs varie durant l'exécution [ABP01] et pour des processeurs avec des vitesses d'exécution différentes [BR02]. En particulier, le nombre de requêtes de vol est $O(pT_\infty)$ w.h.p. ; en conséquence, il est petit dans le cas où $T_\infty \ll T_{seq}$.

Une grande partie des travaux se restreint au cas où $T_\infty \ll T_{seq}$, qui correspond à la plupart des applications embarquées de traitement de flux. Puisque le nombre de requêtes de vol $O(p.T_\infty)$ est petit par rapport au travail total, le *Work First Principle (Principe du Travail D'abord)* est généralement utilisé, et consiste à mettre le surcout de l'ordonnancement au niveau des requêtes de vol. Cela permet ainsi d'optimiser l'exécution séquentielle de l'algorithme parallèle.

Dans [TRM+08], l'implémentation du vol de travail sans deque consiste à supprimer le surcout relatif à la gestion de la deque en retardant la création des tâches : une tâche n'est créée qu'après qu'une requête de vol se produise sur le processeur victime. Dans ce cas, l'opération nommée "extraction parallèle" [TRM+08] crée une nouvelle tâche qui est assignée au processeur voleur.

De manière similaire, dans le système *Tascell* [HYUY09], un processeur actif ne crée une tâche que lorsque cela est requis par un autre processeur en attente. Cette stratégie est appelée *équilibrage de charge basé sur le backtrack* : la victime effectue une extraction parallèle en revenant sur ses pas et en rétablissant son état le plus ancien qui puisse créer une nouvelle tâche.

Frigo, Leiserson et Randall [FLR98] ont appliqué le *Work First Principle* pour implémenter le vol de travail de manière efficace dans le compilateur Cilk : cela implique d'optimiser les synchronisations des opérations sur la deque, ce qui est fait par l'intermédiaire d'un protocole appelé THE.

3.2 Problèmes et contraintes relatives au vol de travail

Le vol de travail basé sur les deques s'emploie assez naturellement avec le parallélisme de fonctions. Par exemple, le mot-clé *spawn* en Cilk est utilisé pour indiquer une fonction qui peut être appelée de manière asynchrone. Cependant, le parallélisme de données est plus difficile à exploiter avec ce type de vol de travail. La solution généralement utilisée est de faire une découpe récursive des données. Nous illustrons ce problème avec deux exemples de la bibliothèque TBB d'Intel.

Dans TBB, la fonction `parallel_for`, qui applique une fonction sans effet de bord à toute une plage d'éléments, est implémentée avec une découpe récursive jusqu'à un certain seuil.

Le problème est que beaucoup de tâches ainsi créées ne seront jamais volées. Ces tâches ont un cout de création et de maintien, et même si le *Work First Principle* est appliqué, cette découpe récursive crée un surcout comparé à l'exécution séquentielle pure d'une boucle for. La solution à ce problème dans TBB est d'utiliser un `auto_partitionneur` qui fait une découpe intelligente en fonction du nombre de processeurs et des évènements de vol.

Le second exemple est la fonction `parallel_scan`, qui calcule le préfixe des éléments d'une plage. Après avoir coupé la plage des éléments à traiter en deux moitiés et calculé le préfixe sur chaque moitié, elle doit faire une opération supplémentaire pour calculer le résultat final : multiplier tous les éléments de la seconde moitié par préfixe du dernier élément de la première moitié. Nicolau et Wang [WNySS96] ont montré qu'une borne inférieure pour le temps de calcul des préfixes sur p processeurs est $\frac{2n}{p+1}$ où n est le nombre d'éléments de la plage. Un algorithme naïf basé sur une découpe récursive calculerait le résultat en $\frac{2n}{p}$ opérations, ce qui n'est pas optimal. Ceci est la conséquence d'opérations additionnelles dues à la parallélisation. TBB adresse ce problème en évitant ce surcout lorsque la seconde moitié n'a pas été volée.

Ces exemples montrent que le problème de la découpe des données est crucial vis-à-vis des performances d'un système de vol de travail.

Enfin, le vol de travail basé sur les deques doit faire face au problème de la granularité : si les tâches sont à grain fin, i.e. que les fonctions parallèles sont courtes, le surcout lié à la gestion et l'ordonnancement entre tâche est élevé ; mais si les tâches sont à gros grain, cela limite potentiellement le parallélisme.

3.2.1 Une solution : le vol de travail adaptatif

Une solution pour conserver un parallélisme entre données efficace et adresser le problème de la granularité est d'utiliser le vol de travail adaptatif. Ce dernier est basé sur le couplage de deux algorithmes : un algorithme séquentiel exécuté par un processeur, qui consiste à faire des *extractions séquentielles* d'une partie du travail total et à exécuter ce travail localement ; un algorithme parallèle permettant de remettre en cause le calcul séquentiel avec une extraction parallèle du travail.

Une implémentation logicielle efficace d'un schéma de vol de travail adaptatif suit le *Work First Principle* qui est la combinaison de deux résultats : premièrement, l'algorithme parallèle exécuté séquentiellement (comme dans Cilk) est remplacé par un algorithme séquentiel optimisé [DGK+05] ; deuxièmement, l'introduction d'une boucle pour contrôler le surcout dû à la parallélisation [DGG+07]. En suivant ce principe, une implémentation sans deque est prouvée optimale, à la fois en théorie et en pratique, pour une large gamme d'algorithmes de la STL [TRM+08]. Une autre implémentation, à laquelle nous allons nous intéresser par la suite a été prouvée optimale pour les calculs sur des flux de données [BRT08].

3.3 AWS : une bibliothèque de vol de travail adaptatif

La bibliothèque de vol de travail constitue une part importante de l'écriture d'un algorithme parallèle. Cette section vise à présenter la bibliothèque AWS (pour Adaptive Work Stealing), qui est une implémentation du vol de travail adaptatif, écrite en vue de tourner sur des systèmes embarqués. Nous nous intéressons à cette bibliothèque dans le but de tirer parti des avantages du vol de travail dans notre contexte architectural.

3.3.1 Philosophie d'AWS

Une contrainte majeure dans les systèmes embarqués et les MPSoCs est que l'espace mémoire doit être statiquement borné. En conséquence, cette spécification du vol de travail n'alloue pas de mémoire supplémentaire durant l'exécution. Par ailleurs, elle ne requiert pas de gestion de la concurrence entre les threads puisque à chaque instant, seul un calcul est en cours sur chaque unité d'exécution qui n'est pas en attente.

En effet, plutôt que de gérer sur chaque unité un ensemble de tâches prêtes, AWS suit l'idée de l'algorithme de vol de travail sans deque proposé dans [TRM+08] qui est basé sur la création paresseuse des tâches. Un processeur actif j effectue le calcul de la tâche qui lui est assignée ; quand une requête de vol venant du processeur i arrive sur j, une nouvelle tâche est créée et correspond à une partie du travail restant sur j prête à être calculée. Cette tâche est alors assignée au processeur i qui peut continuer son exécution.

Dans la suite, l'opération qui construit la description d'une tâche publique est appelée `extract_par()` et celle qui construit la description d'une tâche privée est appelée `extract_seq()`. Ces opérations sont implémentées au niveau de l'application. Ainsi, la création effective des tâches est déléguée à l'application ; AWS ne gère que les requêtes de vol sur les processeurs en attente et l'exécution locale du travail sur les processeurs actifs.

Ces deux opérations sont indépendantes : un vol qui se termine avec succès mène à un `extract_par()` sur la victime, ce qui modifie le travail restant. Cette synchronisation entre les requêtes de vol et l'exécution locale est gérée au niveau de la bibliothèque de vol de travail. Le travail local sur un processeur actif est effectué sous la forme d'un bloc d'opérations, et est défini au niveau de l'application : la fonction `extract_seq()` est appelée de manière itérative sur le travail local jusqu'à sa complétion.

3.3.2 Avantages et contraintes du modèle de programmation lié à AWS

Cette section discute de l'impact des caractéristiques et des choix fait dans AWS sur plusieurs points. Premièrement, un des avantages d'AWS est de laisser à l'utilisateur la possibilité de descendre à grain fin tout en n'impliquant qu'un surcout très faible. En effet, la quantité de travail volée et la taille des tâches extraites s'adapte à la quantité de travail restante.

Par ailleurs, l'ordonnancement des tâches prêtes est connu pour avoir un impact sur les performances. Ainsi, la délégation au niveau applicatif des opérations `extract_par()` et `extract_seq()` supprime ce surcout au niveau de l'application. L'inconvénient que cela pose est qu'il faut arriver à écrire son programme sous la forme de ces deux fonctions, plus celle pour le traitement des éléments.

Ensuite, il est à noter que le couplage de deux algorithmes, un séquentiel - pour les opérations `extract_seq()` - et un parallèle - pour les opérations `extract_par()` - n'empêche pas le parallélisme récursif : il est possible d'imaginer la gestion d'une collection de tâches prêtes à être volées à l'intérieur du travail local. Bien sûr, cela délaisse alors les contraintes mémoire au niveau applicatif. Dans le même ordre d'idée, il est imaginable d'utiliser une bibliothèque de plus haut niveau au-dessus d'AWS pour gérer des synchronisations supplémentaires.

L'inconvénient principal de l'approche proposée est que pour une application complexe, toutes les synchronisations inter-tâches (comme dans le cas de l'exemple précédent avec le parallélisme récursif) doivent être gérées dans le code de l'application. En effet, AWS ne gère pas la dépendance entre tâches de par son modèle, et n'est donc pas très adapté à toutes les classes d'applications. Cela n'est cependant pas nécessairement un problème, puisque les applications de traitement de données ne requièrent pas de telles synchronisations.

3.3.3 Implémentation d'AWS

Comme nous avons dans nos travaux utilisé et modifié la bibliothèque AWS, il est nécessaire de décrire son implémentation. Néanmoins, comme nous sommes repartis d'une bibliothèque existante, nous la présentons dans cette section. Nous rappellons que notre objectif ici est d'étudier l'intérêt d'un support matériel ad-hoc en vue d'améliorer les performances de l'implémentation.

3.3.3.1 Comportement général

Basé sur les choix de conception précédents, ce paragraphe présente la façon dont la bibliothèque d'AWS est construite, ainsi que son API.

Le comportement général est le suivant. Au démarrage, chaque processeur est actif et commence un calcul, déterminé par son identifiant processeur. Quand un processeur va en attente, il devient un voleur. Il sélectionne une victime de manière cyclique jusqu'à trouver du travail à voler.

3.3.3.2 Structures utilisées pour le travail

Chaque processeur gère deux structures `work_t` qui représentent une partie du travail total. À chaque instant, toute donnée ne peut être contenue que dans au plus une structure `work_t` du système. La première structure `work_t` est publique et visible de tous les autres processeurs. Elle est initialisée avec une certaine quantité de travail, généralement la même pour tous les processeurs. La seconde structure `work_t` est une structure privée qui n'est visible que pour le processeur qui la possède et qui contient le travail à faire localement.

3.3.3.3 Cœur de l'algorithme

Le cœur de l'algorithme de vol de travail adaptatif (fonction `loop_core_adaptive`) est présenté figure 3.1. Après une requête de vol se soldant par un succès, un processeur obtient du travail w et exécute une boucle locale : la `local-loop`. Dans cette boucle, le processeur exécute une fonction appelée `extract_seq()` qui extrait une petite partie du travail du `work_t` public vers le `work_t` privé l. Le processeur traite ensuite ce travail l avec une fonction appelée `local_run()` : ce traitement est fait de manière séquentielle et ne peut être préempté. Quand la structure `work_t` publique w est vide, le processeur devient un voleur et sort de la `local-loop`. Il scanne alors toutes les structures `work_t` publiques jusqu'à en trouver une non-vide et exécute alors la fonction `extract_par()` qui extrait une partie du travail de la victime dans sa propre structure publique w. Le processeur entre ensuite à nouveau dans la `local-loop`. Quand toutes les structures `work_t` publiques sont vides, la totalité du travail a été traitée.

3.3.3.4 Nature des structures `work_t`

Les structures `work_t` ne contiennent en général pas le travail lui-même, mais plutôt une description du travail, tels que des indexes définissant le début et la fin d'une zone à traiter.

Pour le cas où les données d'entrées sont un tableau, la figure 3.2 illustre la structure d'un nœud, qui contient deux structures `work_t` pointant respectivement vers le premier et le dernier élément du travail en cours de traitement pour ce nœud, et le premier et dernier élément du travail restant, qui lui peut être volé.

```
1   /* node_mutex: verrou protégeant le travail partagé
2      (i.e. pouvant être volé) du noeud courant */
3   lock(node_mutex);
4   has_local_work = true;
5   has_global_work = true;
6   while (has_global_work) {     /* steal-loop */
7     while (has_local_work) {    /* local-loop */
8       status = extract_seq(); /* extrait du travail local l de w */
9       if (status == STATUS_OK) {
10        unlock(node_mutex);
11        local_run();          /* traite localement l */
12        lock(node_mutex);
13      } else
14        has_local_work = false;
15    }                           /* fin de la local-loop */
16    /* try steal */
17    unlock(node_mutex);
18    status = steal();           /* récupère du travail partagé : w */
19    lock(node_mutex);
20
21    if (status == STATUS_OK)
22      has_local_work = true;
23    else
24      has_global_work = false;
25  }                             /* fin de la steal-loop */
26  aws_unlock(node_mutex);
```

FIG. 3.1 – Cœur de l'algorithme AWS (fonction `loop_core_adaptive`)

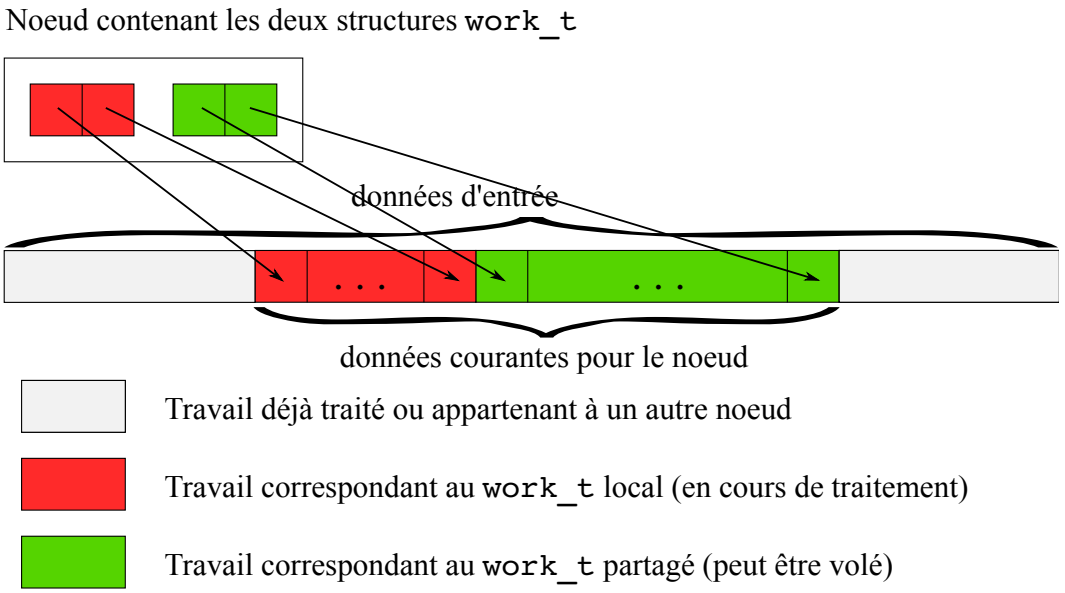


FIG. 3.2 – Exemple des données contenues dans un nœud

3.3.3.5 Taille des parties extraites par les fonctions `extract_par()` et `extract_seq()`

Pour limiter le surcout lié à l'ordonnancement, il est montré que la fonction `extract_seq()` doit extraire log du travail restant, et la fonction `extract_par()` doit extraire une fraction, généralement la moitié, du travail restant. Ainsi, par une analyse théorique de la taille des parties extraites par les fonctions `extract_seq()` et `extract_par()`, l'implémentation garantit un temps asymptotique optimal pour les calculs sur des flux, tout en ne dépendant pas du nombre et de la nature des processeurs [BRT08].

Dans l'implémentation, les données d'entrées sont initialement pré-allouées entre les processeurs. Après une opération de vol réussie, la fonction `extract_par()` extrait la moitié du travail restant. La fonction `extract_seq()` extrait une quantité de travail égale à $\alpha \log_2$ du travail restant.

Dans le cas restreint à une application très régulière, la parallélisation statique (PAR) en parties de tailles égales fournira des performances optimales. Cela est le cas pour certains codes de traitement d'images ou de signaux numériques [Fea91].

3.4 Adaptation des architectures avec les programmes parallèles

Si les architectures dédiées sont le moyen d'obtenir les performances idéales pour une application donnée, elles ne sont pas toujours souhaitables pour des raisons de généricité et d'évolutivité. Néanmoins, tout en restant générique, il est possible de faire des choix de conception qui impactent les performances en fonction du modèle de programmation des applications. Ces choix portent sur l'exploration pour le logiciel et au moment propice, de composants existant déjà dans l'architecture de base, ou devant y être ajoutés. Nous relatons dans cette section les travaux en lien avec les propriétés matérielles des systèmes utilisant le vol de travail ou un concept proche.

3.4.1 Vol de travail et architecture matérielle

Plusieurs travaux analysent les performances du vol de travail sur les machines SMP, distribuées ou intégrées (multicœur), en particulier dans le contexte de Cilk [FLR98]. En considérant les CMPs, [CGK+07] se concentre sur le nombre de défauts de cache en comparant les performances de deux implémentations du vol de travail : une traditionnelle basée sur les deques et une sans deque. Les résultats, basés sur une analyse théorique du nombre de défauts de cache, montrent que les performances varient en fonction du type de l'application, mais sont globalement équivalentes. De plus, ces travaux étant faits sur des machines SMP, le passage à l'échelle du nombre de cœurs s'en trouve limité.

Les expérimentations sur l'utilisation de l'environnement Capsule qui propose un support à l'exécution pour la découpe récursive du travail sont présentés dans [CLP+08]. Ces résultats sont obtenus sur une plateforme de quatre cœurs prédéfinie et les auteurs n'explorent pas d'implémentation alternative sur le placement des données et la synchronisation.

Dans [BRPS06], une technique d'ordonnancement basée sur le vol de travail est appliquée à une application de décodage MPEG-4. L'application est exécutée au-dessus d'une architecture abstraite modélisée en SystemC au niveau Transactionnel (TLM) [Ghe05]. Ce travail bénéficie de la vitesse de simulation rapide des modèles TLM, ce qui permet de valider fonctionnellement l'implémentation de la stratégie d'AWS. Néanmoins, il ne permet pas d'étudier des propriétés architecturales de la plateforme du fait du niveau d'abstraction et de l'absence d'informations de timing dans les modèles.

[LAS+07] étudie le problème de la hiérarchie mémoire dans les systèmes MPSoC, et propose une méthodologie pour comparer les différentes hiérarchies, pour une large gamme d'applications. Différents critères sont utilisés pour évaluer les résultats (temps, énergie, latence et bande passante), mais aucun des algorithmes utilisés pour les benchmarks n'utilise le vol de travail.

Il y a enfin beaucoup de travail sur le partitionnement des données [CRM07] et les propriétés architecturales pour les applications parallèles, mais aucune étude du support logiciel/matériel pour le vol de travail adaptatif n'a encore été conduite.

3.4.2 Conclusion

Si le modèle de programmation apporté par AWS offre des avantages, le gain apporté par l'équilibrage dynamique des charges arrive-t-il à compenser le cout lié au support du vol de travail ? Est-il possible de rendre l'exécution de programmes utilisant cette bibliothèque encore plus efficace en utilisant une architecture générique et adaptée ? Quelles sont alors les critères que doit remplir cette architecture ?

Par ailleurs, est-il possible d'améliorer le cœur de l'algorithme AWS, par exemple en utilisant le modèle de programmation lock-free, ou en ayant un surcout faible même à grain très fin ?

Chapitre 4

Performances du vol de travail et étude de propriétés architecturales

Ce chapitre est découpé en deux parties : dans un premier temps, nous allons effectuer une analyse des performances d'AWS en fonction de certaines propriétés architecturales ; dans un second temps, nous allons nous intéresser au cœur de cet algorithme et proposer des implémentations alternatives, dans le but de diminuer le surcout lié à l'équilibrage du travail.

4.1 Objectifs

Notre but dans la première partie de ce chapitre est d'effectuer une analyse de l'effet de quelques caractéristiques architecturales simples sur les performances de la bibliothèque AWS implémentant le vol de travail adaptatif. Les modifications architecturales que nous visons sont l'utilisation de caches cohérents, de mémoires locales et de DMAs, et la distribution des verrous. Nous déterminons aussi comment ces modifications doivent être prises en compte dans le logiciel. Nous mesurons de plus le surcout du cœur d'AWS comparé à une exécution parallèle statique – appelée PAR dans la suite – pour des configurations matérielles équivalentes, afin de valider dans quelle mesure l'approche d'AWS est viable d'un point de vue performances. Basé sur le pré-partitionnement du travail d'entrée en parties de tailles égales, l'algorithme PAR représente une borne inférieure sur le temps d'exécution pour les applications dont la charge est connue à l'avance : en effet, supporter le vol de travail à l'exécution a un cout, qui doit bien être différencié du gain de temps possible dû à l'équilibrage dynamique de la charge de travail. Ces différents points sont abordés via la simulation d'algorithmes AWS sur différentes plateformes.

Dans ce but, nous nous proposons de :

- Mesurer le temps d'exécution pour un petit programme synthétique (appelé dans la suite micro-kernel) parallélisé avec AWS sur différentes architectures, afin d'évaluer comment des choix de conception usuels peuvent impacter les performances et dans quelle mesure ;
- Mesurer l'accélération sur ces différentes architectures, des parallélisations PAR et AWS par rapport au temps d'exécution séquentiel ;
- Inférer de ces mesures le surcout d'un algorithme AWS comparé à l'algorithme PAR correspondant ;
- Valider l'approche sur deux applications à fort taux de calcul : une avec une charge de travail uniforme, et l'autre avec une charge non-uniforme.

4.2 Architecture MPSoC et nature du micro-kernel choisi

4.2.1 Matériel

Étant donné la taille de l'espace de conception, nous définissons un modèle d'architecture pour lequel nous fixons les paramètres que nous estimons être non significatifs ou ne pas interagir de manière suffisante avec notre contexte d'étude. Comme on peut le voir figure 4.1, la plateforme est une interconnexion de sous-systèmes. Chacun de ces sous-systèmes partage un espace d'adressage commun et peut accéder les modules mémoire locaux ou partagés, un timer ainsi qu'un module de verrous matériels qui permet de prendre un verrou en lisant simplement une adresse, i.e. une IP matérielle qui implémente le *test-and-set* pour une plage d'adresses [SM01].

Le processeur choisi est un Sparc V8 avec 4 fenêtres de registres. Le processeur accède des caches d'instructions et de données disjoints et à correspondance directe.

Il n'y a pas de chargement anticipé (*prefetch*) de données automatique, à l'exception du chargement anticipé pour remplir une ligne de cache, comme dans la plupart des solutions intégrées, dans la mesure où charger de manière spéculative des données et des instructions n'est pas considéré efficace du point de vue de la consommation d'énergie. La configuration de l'architecture est présentée table 4.1, toutes les architectures utilisées dans cette étude dérivant de celle-ci.

TAB. 4.1 – Caractéristiques des plateformes simulées

Nombre de processeurs	$p = \{1, ..., 16\}$
Nombre de bancs mémoire	$p + 3$
Modèle du processeur	SPARC-V8 avec FPU
Taille du cache données	$16Ko$
Taille des lignes du cache données	8 mots (32 octets)
Taille du cache instructions	$16Ko$
Taille des lignes du cache instruction	8 mots
Associativité du cache	Correspondance directe
Taille du tampon d'écritures	8 mots
Contrôleur DMA	2 interfaces initiateur pour envoyer 1 lecture et 1 écriture par cycle (au maximum)
Topologie du NoC	Maillage 2D
Latence du NoC global	$\sqrt{2n}$ cycles pour n interfaces
Latence des NoCs locaux	1 cycle
Déf. d'un cycle sur les graphes	200 cycles simulés

De manière à éviter une congestion élevée sur un seul banc mémoire, ou des latences élevées comme cela se produirait sur une architecture *dance-hall*, la plateforme sur laquelle nous basons notre étude a deux niveaux de hiérarchie. Chaque processeur est relié à un réseau d'interconnexion local, sur lequel sont aussi connectés ses périphériques et une mémoire locale. Ces réseaux locaux sont connectés *via* un pont *(bridge)* sur un réseau d'interconnexion global auquel sont aussi connectées les mémoires partagées.

Le réseau d'interconnexion utilisé est un NoC basé sur le travail de [SGMP08] car l'ex-

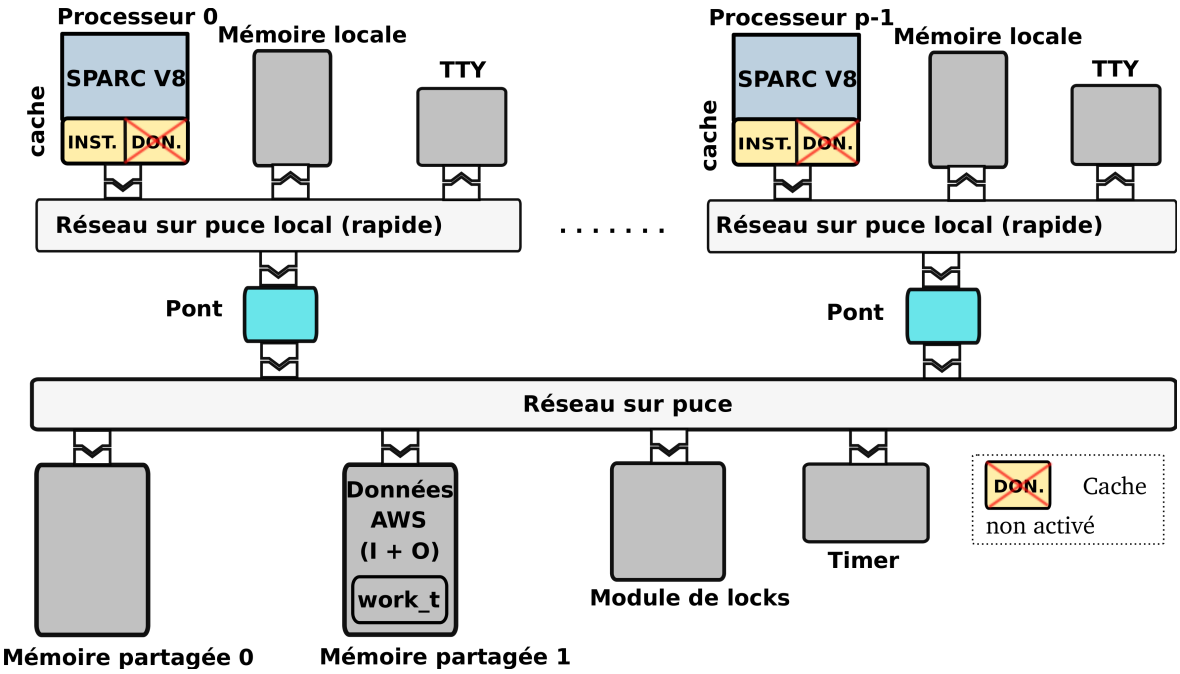


FIG. 4.1 – Vue schématique de l'architecture de base

tensibilité des bus est très limitée, et la complexité des crossbars devient trop importante pour le nombre de processeurs visé.

La topologie utilisée est un maillage 2D puisque cette dernière a un bon ratio temps de traversée/complexité, et a de bonnes propriétés pour le passage sur silicium.

Le composant de mesure du temps utilisé compte un cycle pour 200 cycles de simulations. Aussi, les valeurs présentés en cycles sur les graphes ne sont pas à prendre comme des cycles de simulations, mais doivent d'abord être multipliées par 200 pour cela.

Il aurait été intéressant d'utiliser une mémoire de type *scratch-pad* au lieu d'une mémoire reliée au réseau local (i.e. une mémoire connectée directement au processeur par l'intermédiaire d'une interface dédiée [PDN97]), mais cela n'a pas été exploré car étant une limitation des modèles disponibles dans l'environnement de simulation. Cependant, du fait de la latence du crossbar local, les comportements temporels de ces deux solutions auraient été relativement proches.

L'espace d'adresses vu par les processeurs est partitionné en un ensemble de segments. Un ou plusieurs segments peuvent être associés à un périphérique ou une mémoire, en respectant les contraintes suivantes : les segments associés à un périphérique ne peuvent pas être cachés, et les segments d'une même mémoire doivent avoir le même attribut de cachabilité (caché ou non).

Pour résumer, l'espace de conception matériel qui sera exploré dans notre étude consistera principalement à l'évaluation de la manière dont les DMAs et/ou les caches peuvent améliorer la localité des données et comment le placement physique des verrous peut accélérer la synchronisation.

4.2.2 Système d'exploitation et assignation des tâches

Nous utilisons la configuration *ordonnanceur décentralisé* (DS) d'un noyau léger appelé Mutek [PG03], qui fournit une implémentation des threads POSIX pour les machines multi-

processeurs à mémoire partagée. Contrairement à la configuration SMP dans laquelle tous les processeurs partagent un seul ordonnanceur pour accomplir la sélection des tâches, la configuration DS limite grandement la congestion puisque chaque processeur possède son propre ordonnanceur. Les tâches peuvent être fixées sur un processeur désiré afin d'éviter la migration. Dans ce cas, chaque thread est assigné à un processeur à sa création. Les piles des threads et les données locales sont situées dans les mémoires locales des processeurs.

Toutes les expérimentations présentées dans cette partie sont faites en utilisant cette même configuration du système.

4.2.3 Critères de sélection et choix du micro-kernel

Afin de répondre aux différents problèmes définis, nous avons choisi de faire dans un premier temps nos expérimentations avec un micro-kernel. Le choix de ce micro-kernel a été motivé par trois points.

Premièrement, de manière à évaluer le surcout du vol de travail par rapport aux approches classiques des systèmes embarqués, le micro-kernel doit permettre la calibration d'applications pour lesquelles une parallélisation optimale est connue. Deuxièmement, les applications multimédia demandent beaucoup de ressources en calcul et en communication : le micro-kernel doit donc avoir un grain fin et être représentatif d'une classe d'applications multimédia tels que les filtres numériques ou les transformées. Troisièmement, il doit permettre une analyse théorique sur l'implémentation du vol de travail afin de donner un retour sur les expérimentations.

Nous avons sélectionné un micro-kernel satisfaisant ces contraintes, qui consiste à faire des opérations indépendantes sur les éléments d'un tableau, dont le contenu constitue l'entrée et la sortie du programme. Ce tableau est alloué en mémoire partagée.

En considérant une opération de traitement quasiment nulle sur chaque élément, cela donne au micro-kernel un très haut ratio communication *vs.* calcul, ce qui permet une analyse du surcout du vol de travail en nombre de cycles. De plus, en considérant un nombre de processeurs identiques et un temps de traitement de chaque élément constant, ce surcout peut être comparé au nombre de cycles de la parallélisation standard statique, dénotée PAR : les données d'entrées de taille n sont partagées de manière égale entre les p processeurs, chaque processeur étant donc en charge d'un bloc contigu de taille $\frac{n}{p}$.

4.3 Analyse théorique du temps parallèle pour le micro-kernel

La simplicité du micro-kernel choisi et de son implémentation permet une analyse théorique. Les notations suivantes sont utilisées : comme défini dans le chapitre 3, T_{seq}, T_p et T_∞ dénotent respectivement le temps d'exécution séquentiel, le temps d'exécution parallèle sur p processeurs et le chemin critique, i.e. le temps d'exécution parallèle sur un nombre non borné de processeurs (sans tenir compte des synchronisations finales). On suppose que le temps de calcul τ d'un élément vérifie $\tau_{min} \leq \tau \leq \tau_{max}$. Nous considérons aussi dans cette section que le cache d'instructions peut contenir toute l'application et nous restreignons notre analyse au cache de données.

4.3.1 Analyse théorique pour le PAR

Considérons d'abord le nombre de défauts en cache de données. Soit M_{seq} le nombre de défauts de l'exécution séquentielle, qui correspond au parcours linéaire du tableau. Dans l'exécution PAR, chaque processeur exécute l'algorithme séquentiel sur sa partie

des données. Ainsi, le nombre de défauts de cache M_p^{PAR} par processeur vérifie $M_{seq} \leq p \times M_p^{PAR} \leq M_{seq} + p$. Puisque $p \ll M_{seq}$, le surcout induit par les défauts de cache en parallèle est négligeable.

Le temps d'exécution T_p^{PAR} est donc égal au temps d'exécution de la portion des données prenant le plus de temps pour être calculé. En supposant un temps constant pour le calcul de l'opération d'un élément, on a :

$$T_p^{PAR} \simeq \frac{T_{seq}}{p}. \tag{4.1}$$

Dans le cas général dans lequel le temps de calcul d'un élément peut varier, on a seulement :

$$\frac{T_{seq}}{p} \leq T_p^{PAR} \leq \frac{\tau_{max}}{\tau_{min}} \frac{T_{seq}}{p}. \tag{4.2}$$

4.3.2 Analyse théorique pour AWS

Pour AWS, on note τ_{steal} une borne sur le temps d'une opération de vol (qui réussit ou qui échoue) sur un processeur donné. Ce surcout lié à AWS est relié au nombre total S de vols, qui est proportionnel à T_∞.

Du fait de l'initialisation, de l'extraction de la moitié du travail lors des vols et de l'extraction locale de $\log_2$ du travail restant, on a que : $T_\infty = \mathcal{O}\left(\log_2 T_{seq}\right)$.

De plus, du fait de la recherche cyclique d'un processeur victime, le nombre total d'opérations de vol est $S = \mathcal{O}(p \times T_\infty)$. w.h.p., et dans le pire des cas :

$$S = \mathcal{O}(p^2 \times T_\infty). \tag{4.3}$$

De la même manière que pour le PAR, le nombre M_p^{AWS} de défauts de cache par processeur pour le parcours du tableau est borné au pire des cas par : le nombre M_{seq} de défauts de cache de l'exécution séquentielle, plus au plus deux défauts supplémentaires après chaque opération de vol réussie – un sur le processeur voleur pour charger la nouvelle partie du tableau, et un sur le processeur volé pour mettre à jour son travail local.

Nous avons donc ainsi : $M_{seq} \leq p \times M_p^{AWS} \leq M_{seq} + 2S$.

Il est à noter que dans le cas du micro-kernel choisi, le processeur qui effectue une opération de vol est considéré en attente, et n'a par conséquence pas de donnée utile dans son cache. C'est pourquoi on peut ignorer les défauts de cache avant un vol réussi. Le temps d'exécution finalement attendu est donc de :

$$T_p^{AWS} = \frac{T_{seq}}{p} + \mathcal{O}(S) = \frac{T_{seq}}{p} + \mathcal{O}(p^2 \times T_\infty). \tag{4.4}$$

On remarque entre autre que le surcout lié aux vols (et notamment à la synchronisation finale) est proportionnel au carré du nombre de processeurs.

4.4 Paramètres de l'architecture

Au-delà de l'analyse théorique, les performances effectives pour le PAR et AWS sont fortement relatives à la configuration matérielle. À grain fin, le micro-kernel fait beaucoup d'accès à la mémoire, donc l'usage de caches et DMAs a un impact direct.

Un des moyens pour améliorer les performances est de faire se recouvrir les calculs et les communications. Dans le contexte de notre étude, cela peut consister à utiliser les mémoires locales pour réduire la latence d'accès à la mémoire principale : en utilisant un DMA, il est possible de copier les données de la mémoire principale vers la mémoire locale d'un processeur tandis que ce dernier est en train de faire des calculs. L'utilisation de caches sera aussi étudiée, ainsi que l'utilisation jointe de caches et de DMAs.

Par ailleurs, dans AWS, des opérations de synchronisation supplémentaires sont nécessaires du fait des appels à `extract_par()` et des vols, nécessitant entre autres des prises de verrous. En l'occurrence, accéder un verrou lors de chaque opération `extract_seq()` peut se révéler peu efficace à grain fin. Puisque la plupart des accès sont locaux, on peut espérer que distribuer les verrous et les structures sur les réseaux d'interconnexion locaux résulte en une réduction du temps de latence moyen.

4.4.1 Utilisation de DMAs

Afin d'explorer l'usage de DMAs, l'architecture de base est modifiée par l'ajout d'une unité de DMA sur chaque réseau local (figure 4.2). De cette manière, les données en entrée pour un processeur peuvent être accédées dans la mémoire locale au lieu de la mémoire partagée. L'allocation dans les mémoires locales est rendue possible grâce à un appel système spécifique.

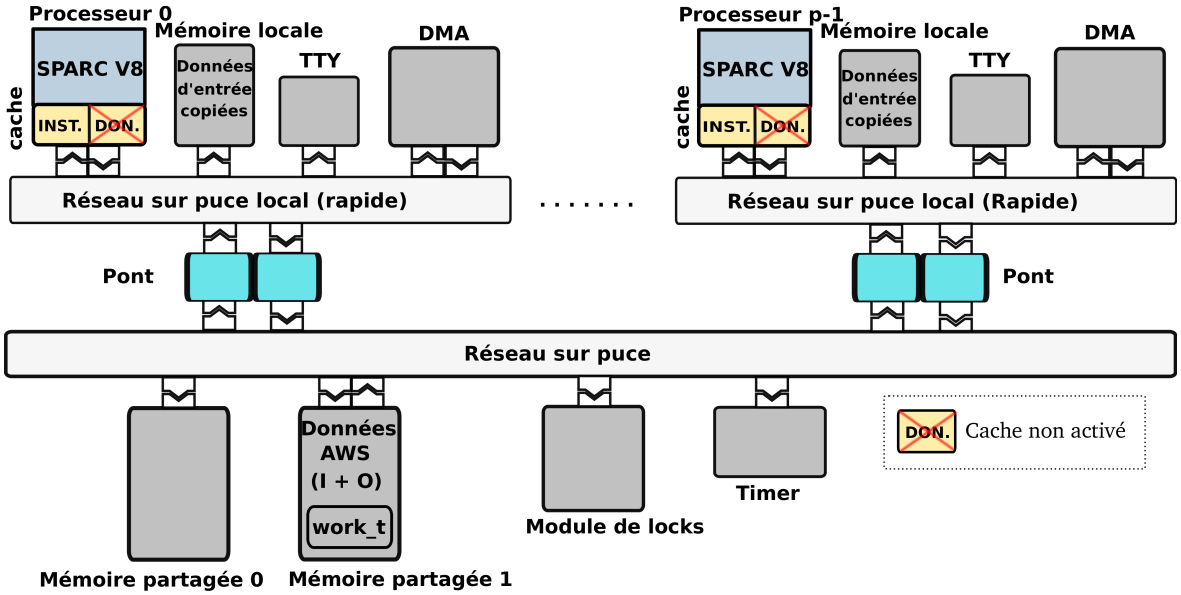


FIG. 4.2 – Architecture avec les DMAs

Le premier problème qui se pose lors de l'utilisation de DMAs est la synchronisation, c'est-à-dire la garantie que la copie est terminée – ou du moins, que les données accédées localement à un instant donné ont été recopiées et sont à jour, même si la copie totale n'est pas finie. Cela peut être fait par *polling* ou par interruption. Le polling apparait plus attractif car il permet de commencer le calcul des données avant la fin de la copie. Notre implémentation du polling utilise un registre du DMA qui contient le nombre d'éléments recopiés et calcule à partir de la valeur de ce registre l'index maximal pour lequel le traitement peut être effectué. Le cout de ces requêtes est négligeable, du fait que la vitesse de recopie est largement supérieure à celle du traitement des éléments (même dans le cas de notre micro-kernel).

La seconde question relative à cette implémentation est de savoir quand effectuer la copie de la mémoire partagée vers la mémoire locale. Plusieurs possibilités ont été envisagées : au début de la fonction `local_run()`, dans la fonction `extract_seq()` ou dans la fonction `extract_par()`. De plus, pour les deux premiers cas se pose le problème de savoir si la copie est celle des éléments en cours de traitement (i.e. correspondant à l'appel courant de `local_run()`), ou celle des prochains éléments traités (i.e. correspondant au prochain appel de `local_run()`).

Copier une partie des données volées dans la fonction `extract_par()` permet de commencer le calcul dès que la fonction `extract_seq()` est appelée, et constitue donc le moyen minimisant *a priori* la perte de temps liée à la programmation du DMA. Néanmoins, cette stratégie impose en particulier de changer l'interface d'AWS, ce que l'on refuse de s'autoriser. Les alternatives sont de programmer le DMA au début des fonctions `extract_seq()` et `local_run()` pour des résultats similaires. Le choix conservé dans les expérimentations sera celui de programmer le DMA au début de la fonction `local_run()`, car seule cette solution est faisable à la fois en PAR et AWS.

Concernant le choix des données à copier par le DMA, copier les prochains éléments traités requiert d'être capable de distinguer le premier et le dernier appel à `extract_seq()` après un `extract_par()`, et induit donc plus de complexité dans ces fonctions. Comme copier les éléments en cours de traitement n'ajoute qu'un surcout très faible sans modifier l'interface d'AWS ou les fonctions `extract_par()` et `extract_seq()`, nous avons décidé de nous limiter à cette solution.

4.4.2 Utilisation de caches

Pour notre micro-kernel, les caches peuvent sembler inutiles puisque chaque élément n'est accédé qu'une seule fois. Toutefois, l'utilisation de caches permet de précharger de manière anticipée une ligne mémoire. Afin de maintenir la cohérence entre tous les caches, nous avons utilisé un mécanisme matériel basé sur répertoire (figure 4.3) détaillé dans [dMP08].

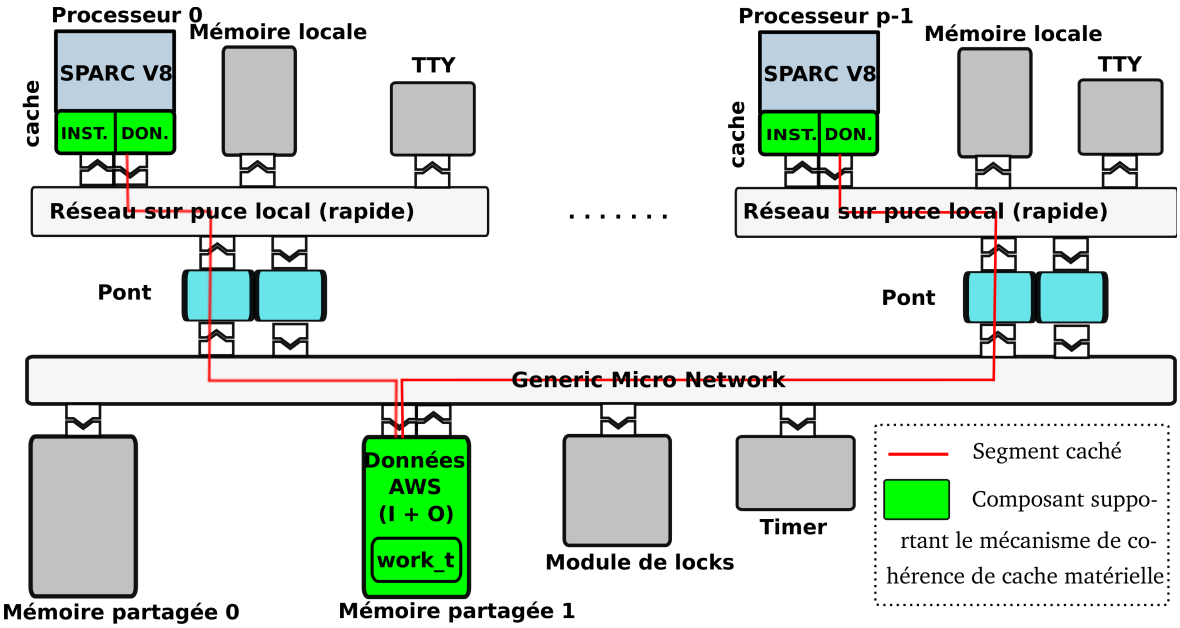


FIG. 4.3 – Architecture avec les caches cohérents

4.4.3 Utilisation de caches et DMAs

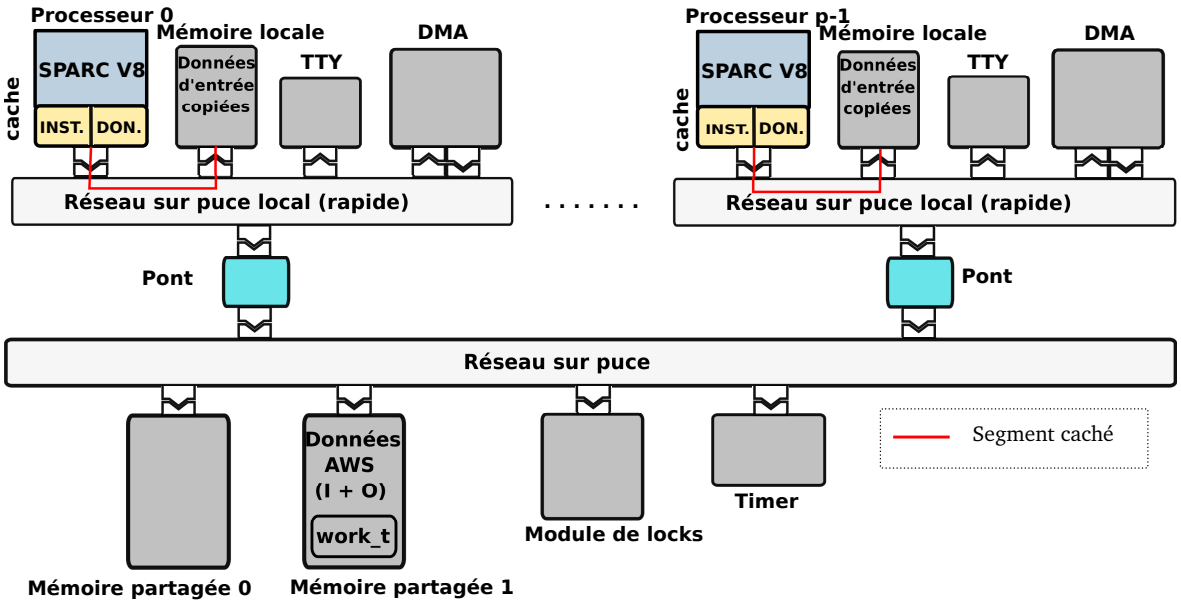


FIG. 4.4 – Architecture avec des caches et des DMAs

Nous avons aussi exploré l'usage conjoint de caches et de DMAs, puisque ces derniers opèrent sur des parties différentes du transfert. L'architecture résultante est montrée figure 4.4. Le segment mémoire caché est celui correspondant à la mémoire locale.

Pour optimiser les accès et éviter les problèmes de cohérence entre caches et mémoires, l'adresse de base du tableau dans la mémoire locale et la taille des éléments localement accédés sont systématiquement alignés sur la taille d'une ligne de cache. Cela peut être assuré au moment de l'allocation par l'utilisation d'un allocateur qui garantit l'alignement (`posix_memalign`).

4.4.4 Distribution des verrous et des structures de travail

Un autre moyen de réduire les latences d'accès est de distribuer les verrous sur chaque nœud au lieu de les avoir tous accessibles depuis le réseau d'interconnexion global. De manière similaire, les structures `work_t` peuvent être conservées localement et non en mémoire partagée.

Cela nécessite que chaque processeur ait un accès au réseau d'interconnexion local des autres processeurs. En conséquence, le temps d'accès au verrou dans le cas d'un vol est augmenté, mais les vols sont prouvés être rares (eq. 4.3 [BR02]). Suivant le *Work First Principle*, la majorité des accès aux verrous ou aux structures `work_t` sont donc locaux ; cela arrive notamment lorsqu'un nœud extrait du travail pour le traiter séquentiellement.

Afin de limiter le nombre de configurations matérielles/logicielles à comparer, nous avons retenu les deux configurations suivantes pour cette expérimentation :

1. L'architecture de base, sans cohérence de cache ou DMA (voir figure 4.1 avec les données partagées non cachées), qui fournit les mesures de référence pour les performances
2. L'architecture avec les caches cohérents

4.5 Résultats et analyse

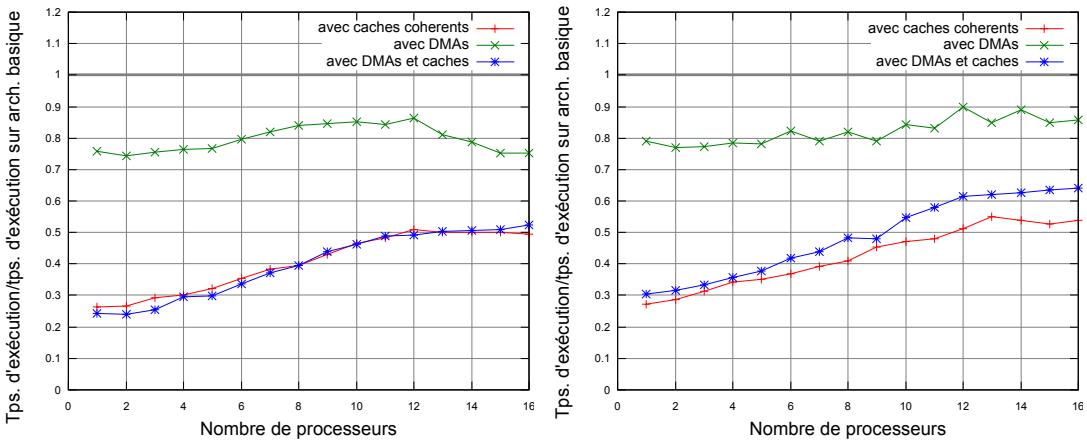
Les simulations ont été faites avec les modèles SystemC *cycle-accurate* de l'environnement SoCLib [The08].

Nous avons fait les expérimentations avec deux configurations du micro-kernel : la première consiste à avoir un tableau d'un total de 100 000 éléments, tandis que le tableau de la seconde ne contient que 10 000 éléments. Comme expliqué en section 3.3.3.5, le nombre d'éléments extraits par la fonction `extract_seq()` doit être $O(\log(m))$, où m est le nombre d'éléments restant à traiter.

En pratique, il faut choisir un paramètre α et extraire $\alpha.\log_2(m)$. Ce paramètre, appelé ratio d'extraction séquentielle, varie normalement d'une application à l'autre si l'on veut obtenir des performances optimales : un grand ratio évite le surcout d'extractions séquentielles inutiles, mais à l'inverse limite potentiellement le parallélisme. À défaut de pouvoir faire varier ce ratio au moyen d'un grand nombre de simulations en raison du temps de ces dernières, nous avons choisi de fixer ce ratio à une valeur. Nous avons essayé de choisir une valeur représentative de celles que l'on peut trouver dans les applications, la valeur retenue étant de 2^7, soit 128. Il est à noter que pour le micro-kernel, les performances seront toujours meilleures à mesure que le ratio grandit, étant donné qu'il n'y a quasiment aucun vol. Toutefois, cela aurait peu d'intérêt d'évaluer le surcout lié à AWS dans un tel cas, qui ne serait alors pas représentatif d'applications réelles.

4.5.1 Comparaison des temps d'exécution sur les architectures définies

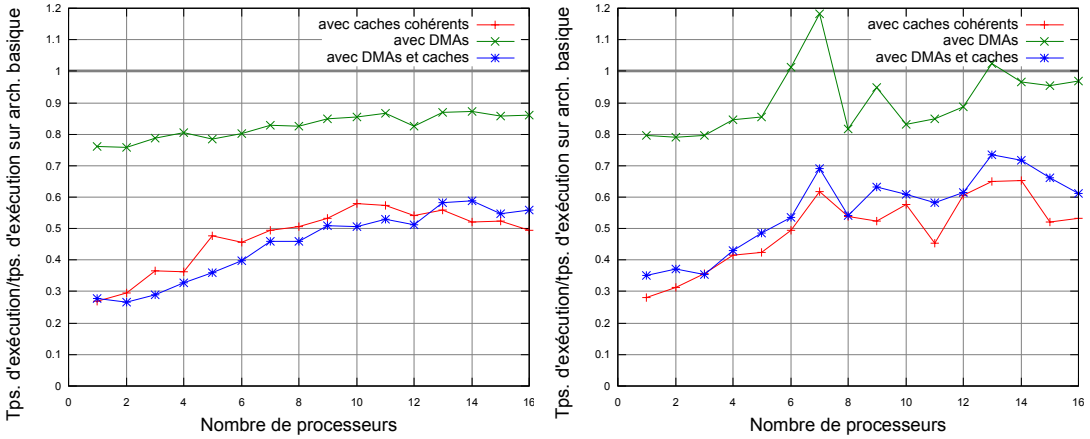
Les figures 4.5(a) et 4.5(b) montrent les temps d'exécution normalisés (par rapport aux temps sur l'architecture de base) pour les algorithmes PAR et AWS, avec des tableaux de respectivement 100 000 éléments et 10 000 éléments.



(a) PAR (b) AWS

FIG. 4.5 – Temps d'exécution sur les différentes architectures pour 1 à 16 processeurs, normalisés p/r aux temps sur l'architecture de base, avec 100k éléments

La première information que l'on peut noter est que les temps de simulation obtenus avec le PAR et AWS se comportent de la même manière pour les trois configurations, mais sont un peu moins réguliers avec AWS. Cela s'explique par le trafic dû aux synchronisations



(a) PAR

(b) AWS

FIG. 4.6 – Temps d'exécution sur les différentes architectures pour 1 à 16 processeurs, normalisés p/r aux temps sur l'architecture de base, avec 10k éléments

finales dans AWS. Selon l'ordre des requêtes envoyées durant cette phase ($\mathcal{O}(p^2)$), certains threads peuvent commuter et passer en attente, ce qui introduit un surcout.

En allant plus dans les détails pour chaque architecture, on remarque que l'architecture avec les caches et celle avec les DMAs et caches ont les meilleurs résultats et sont équivalentes (accélération de 4 à 1.7), tandis que l'architecture avec les DMAs seuls n'exhibe pas un gain de temps significatif comparé à l'architecture de base. Néanmoins, pour un nombre de processeurs élevé, des améliorations sont visibles pour le PAR et AWS pour cette architecture. Cela peut s'expliquer par le fait qu'avec beaucoup de processeurs, la latence sur le réseau d'interconnexion global devient suffisamment importante pour que copier les données dans les mémoires locales en vaille la peine.

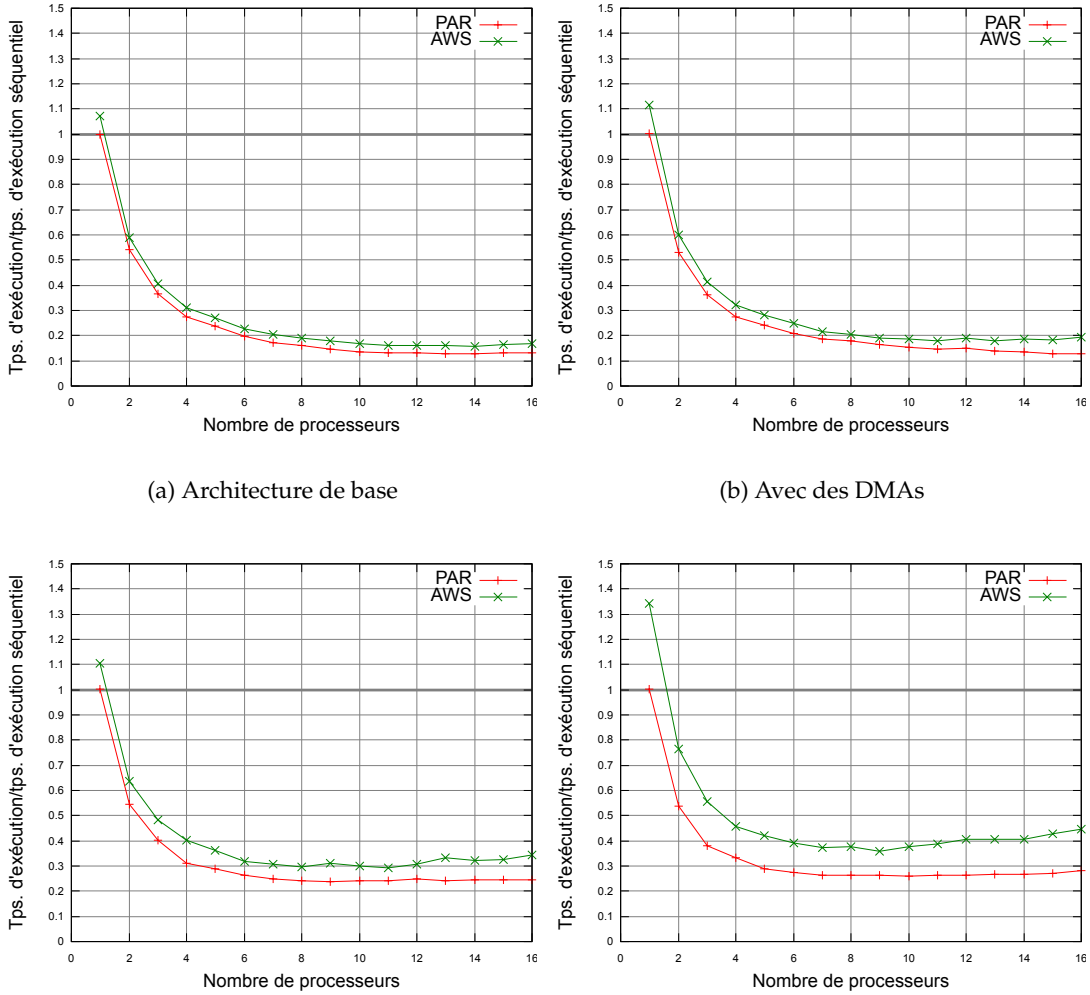
On peut aussi voir que le temps gagné décroit à mesure que le nombre de processeurs augmente pour les deux configurations les plus rapides (caches cohérents, DMAs et caches). En effet, ces configurations optimisent les accès pour les données, induisant des calculs plus rapides. Cependant, cela n'aide pas à améliorer le cout du parallélisme ou les surcouts dus aux synchronisations, qui représentent alors un plus grand pourcentage du temps total avec beaucoup de processeurs.

Les figures 4.6(a) et 4.6(b) montrent les temps d'exécution avec des tableaux de 10 000 éléments. Comparé aux résultats précédents, la non-régularité des temps d'exécution avec AWS est amplifiée par la petite taille des données en entrée. La gestion du parallélisme dans AWS est trop couteuse pour cette taille des données en entrée. Cela montre donc une limitation d'AWS qui est l'impossibilité de fournir des bonnes performances par rapport à une parallélisation statique lorsque les données en entrée sont trop petites.

4.5.2 Comparaison des temps d'exécution séquentiels et parallèles sur une architecture donnée

4.5.2.1 Avec des données en entrée de grande taille

La figure 4.7 montre les temps d'exécution normalisés pour chaque architecture et pour 100K éléments.



(a) Architecture de base

(b) Avec des DMAs

(c) Avec des caches cohérents

(d) Avec DMAs et caches

FIG. 4.7 – Temps d'exécution normalisés sur les différentes architectures pour 1 à 16 processeur et 100K éléments

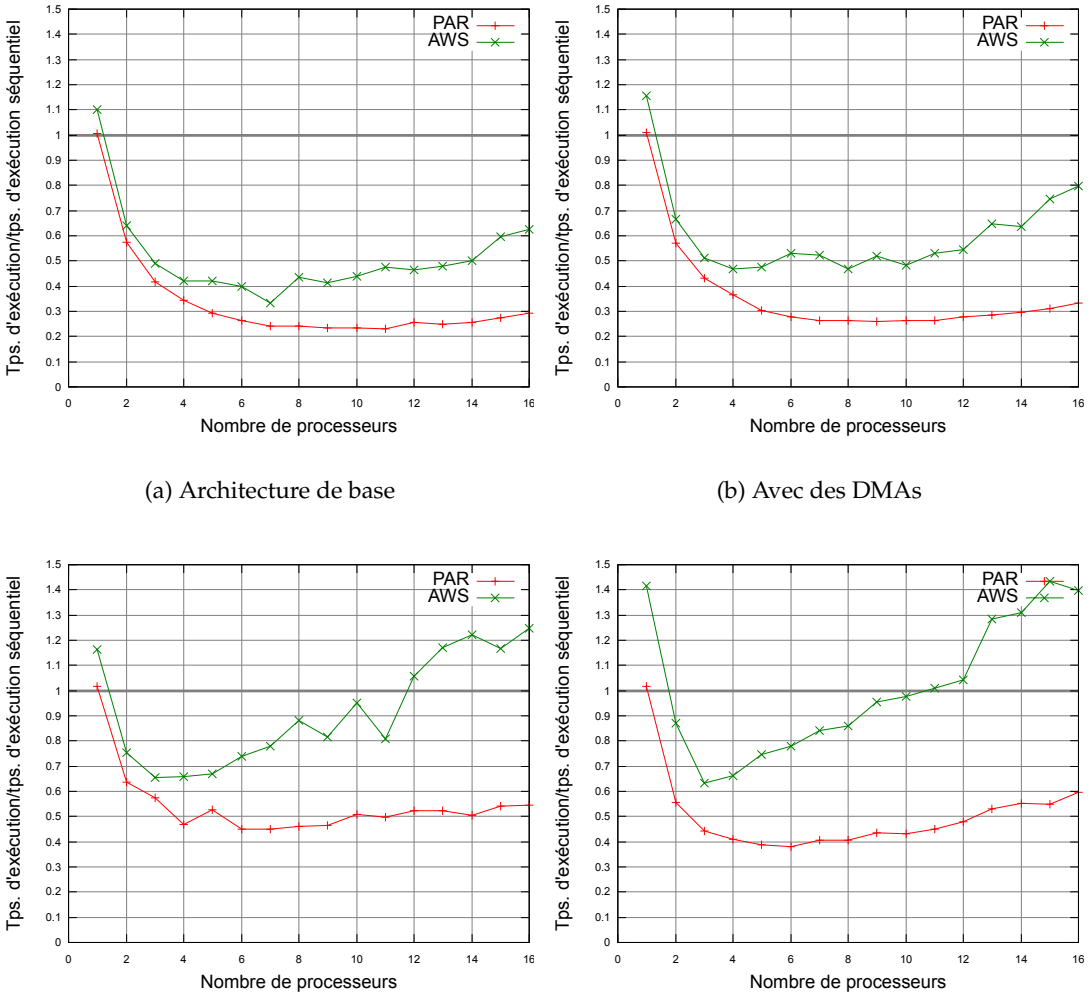
Si toutes les courbes ont globalement le même comportement, on peut remarquer qu'à la fois pour le PAR et pour AWS, le temps gagné grâce au parallélisme est logiquement plus important pour les architectures les plus lentes : en effet, comme le temps total d'exécution est plus long, le surcout de parallélisation, restant à peu près constant, est proportionnellement plus petit par rapport au temps total. Ainsi, l'accélération maximale obtenue est presque de 7 pour l'architecture la plus lente contre 4 pour les plus rapides.

Le même effet se produit pour AWS qui est de 5% à 35% plus lent que le PAR selon la configuration, à cause du surcout lié à l'équilibrage dynamique du travail. Ceci étant, un point important à remarquer est que la configuration avec les DMAs et les caches est significativement plus lente que celle avec les caches seuls, en particulier pour AWS. Le surcout d'AWS comparé au PAR est effectivement autour de 20% pour les caches cohérents, tandis qu'il est environ de 35% pour les DMAs et caches. Ces résultats suggèrent donc qu'utiliser des caches cohérents est la meilleure solution à notre problème.

Finalement, ces graphes montrent que pour cette taille d'entrée, le nombre de processeurs optimal est autour de 6.

4.5.2.2 Avec des données en entrée de petite taille

Les courbes montrant les temps d'exécution normalisés avec des tableaux de petite taille sont données figure 4.8.



(a) Architecture de base

(b) Avec des DMAs

(c) Avec des caches cohérents

(d) Avec DMAs et caches

FIG. 4.8 – Temps d'exécution normalisés sur les différentes architectures pour 1 à 16 processeur et 10K éléments

Une fois encore, ces graphes révèlent que le surcout lié à AWS est prohibitif quand les données à traiter sont de trop petite taille. Les temps d'exécution sur les architectures les plus rapides (DMAs et caches, caches cohérents) dépassent rapidement le temps d'exécution séquentiel. Par ailleurs, le nombre optimal de processeurs pour cette taille d'entrée semble être de 4.

Finalement, la dernière tendance à noter est que plus la performance pour une architecture donnée est bonne, plus le gain apporté par l'ajout de processeurs est faible, à la fois pour AWS et pour le PAR. Une fois encore, cela peut s'expliquer par le fait qu'une partie de l'application ne peut pas être parallélisée.

Bien que nous nous attendions à ce que AWS soit plus lent que le PAR, nous pensions que son surcout serait plus faible avec un jeu d'entrée de petite taille. Cela montre qu'améliorer

les performances sur la partie locale de l'exécution sera limité à un certain point par la création de parallélisme et les synchronisations. Ceci a motivé l'expérimentation suivante.

4.5.3 Distribution des verrous et des structures de travail

Sur la figure 4.9 sont présentés les gains de temps relatifs des configurations avec les structures `work_t` et les verrous distribués. Les temps pour le PAR sont de manière assez évidente identiques, puisqu'il n'y a pas d'accès aux structures `work_t` partagées, ni aux verrous correspondants.

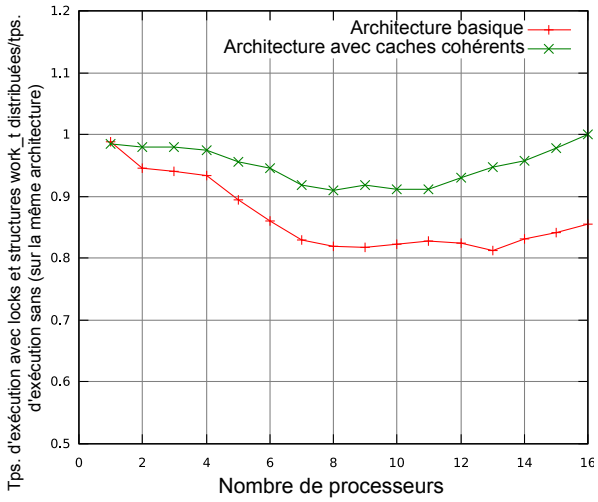


FIG. 4.9 – Temps d'exécution normalisés avec les structures `work_t` et verrous distribués, sur deux architectures

Premièrement, le temps gagné n'est pas négligeable, et même si nous sommes dans un cas où il y a peu de vols, le nombre de vols est globalement toujours faible. Par ailleurs, le temps gagné est plus important en proportion pour l'architecture de base. Cela peut être expliqué par le fait qu'en présence de caches, les structures de travail seront elles aussi cachées si elles sont situées en mémoire principale ; en conséquence, déplacer ces structures dans les mémoires locales ne résulte pas en un gain de temps.

4.5.4 Comparaison du surcout théorique et du surcout pratique d'AWS

Comme vu dans l'équation 4.3, le surcout théorique d'AWS est quadratique en le nombre de processeurs quand les processeurs victimes sont choisis cycliquement. La figure 4.10 trace le surcout (*i.e.* la somme des cycles passés sur tous les processeurs moins le temps séquentiel) pour le micro-kernel avec 100K éléments et sur l'architecture de base. Sur la même figure est aussi tracée une régression quadratique des données, montrant ainsi que les résultats théoriques attendus sont confirmés par nos expériences.

4.5.5 Conclusion

Les deux approches présentées pour améliorer les performances en utilisant des DMAs ou des caches suivent dans une certaine mesure les deux modèles mémoire basiques existants pour les MPSoCs contemporains : caches cohérents gérés matériellement et adressés

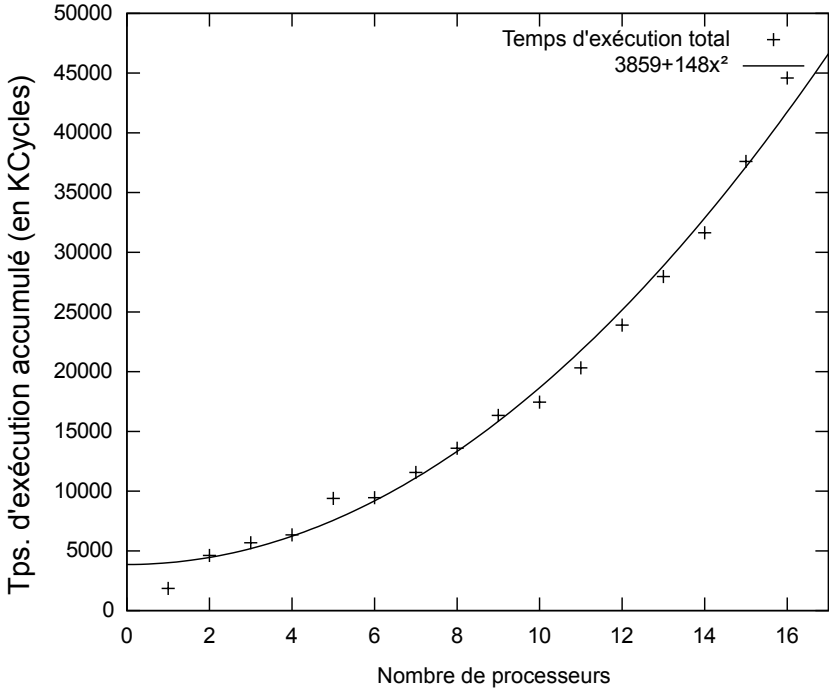


FIG. 4.10 – Comportement du surcout lié à AWS

de manière implicite, et mémoires locales gérées par le logiciel et adressées de manière explicite [LAS+07].

Nos résultats tendent à montrer que pour les algorithmes utilisant AWS, les deux architectures utilisant des caches marchent mieux qu'une copie explicite seule, et que la solution avec les caches cohérents passe mieux à l'échelle avec des petites données que la solution avec des DMAs et des caches. Cependant, notre modèle peut être remis en cause pour la raison que les mémoires locales ne sont pas accessibles en un cycle par le processeur.

Nous avons remarqué que l'implémentation d'AWS est très sensible à des petits changements de synchronisation. Après investigation, nous avons trouvé que cela est dû au fait que lorsqu'un thread passe en attente (parce que le mutex requis est déjà pris), il consomme beaucoup de temps à cause du double changement de contexte requis pour céder le processeur et le reprendre. Utiliser des spin locks au lieu des mutexes a permis de réduire cette sensibilité. Les résultats montrent aussi qu'AWS est plus sensible aux variations de taille qu'une parallélisation statique, c'est pourquoi nous recommandons l'emploi d'AWS quand la taille des données d'entrée est suffisamment grande.

4.6 Performances sur deux applications

Nous avons utilisé deux applications pour notre étude : une réduction du bruit temporel de la luminance et de la chrominance (TNR), et le calcul et tracé de sous-ensembles de l'ensemble de Mandelbrot.

Nous avons choisi ces applications car nous avons voulu couvrir les deux extrémités du spectre au regard de la régularité de la charge de travail.

4.6.1 Présentation de l'application TNR

L'application TNR est un programme de filtrage d'image. Il contient plusieurs calculs successifs sur une image : réduction du bruit temporel, réduction du bruit spatial, détection de mouvement et atténuation de la couleur. Chaque calcul est une itération sur tous les pixels de l'image via une double boucle `for`. La taille des images utilisées dans la séquence est de 714x244.

L'application est *a priori* très régulière puisque les données sont partagées en blocs et que le nombre d'éléments traités varie très peu d'un bloc à un autre.

Nous avons effectué des simulations pour un nombre de processeurs variant entre 1 et 16 pour le décodage de 4 ou 6 images. Puisque les calculs sur cette application sont relativement longs, nous avons choisi de ne pas traiter toutes les configurations, mais de se restreindre aux suivantes :

- l'architecture de base pour le PAR (*PAR basique*)
- l'architecture de base pour AWS (*AWS basique*)
- l'architecture avec des caches, et des verrous distribués pour AWS (*AWS amélioré*)

4.6.2 Résultats pour TNR

Image	Temps de calcul sur		
	PAR basique	AWS basique	AWS amélioré
1	6 512	6 514	5 559
2	6 527	6 518	5 571
3	6 529	6 531	5 576
4	6 532	6 531	5 578
5	6 529	6 527	5 571
6	6 525	6 523	5 567

TAB. 4.2 – Temps d'exécution de TNR (en KCycles)

Les résultats détaillés pour le décodage de 6 images sur 4 processeurs sont présentés dans la table 4.2. La comparaison des colonnes pour les configurations PAR basique et AWS basique montre que les temps d'exécution sont très proches entre le PAR le AWS, et qu'aucun des deux n'obtient de meilleurs résultats que l'autre.

Le gain de performance relatif aux caches et aux verrous distribués est approximativement de 15% par image. Ce gain de temps est plus faible qu'avec le micro-kernel car il y a dans le cas de cette application beaucoup plus de calculs pour un accès à la mémoire ou aux verrous.

Les résultats généraux pour 1, 2, 4, 8 et 16 processeurs sont présentés Figure 4.11. Ce graphe montre le temps moyen de décodage pour 4 images. Comme avec 4 processeurs, AWS et PAR ont des performances toujours similaires, mais ce graphe permet de mettre en évidence que la configuration *AWS amélioré* est plus efficace quand le nombre de processeurs est élevé. Cela s'explique par le fait que quand le nombre de processeurs augmente, la latence globale augmente aussi, donc exploiter la localité avec les caches et les verrous locaux donne de meilleurs résultats. De plus, le nombre d'accès aux verrous croît en $O(p^2)$, donc distribuer les verrous quand le nombre de processeurs est élevé réduit la congestion du réseau.

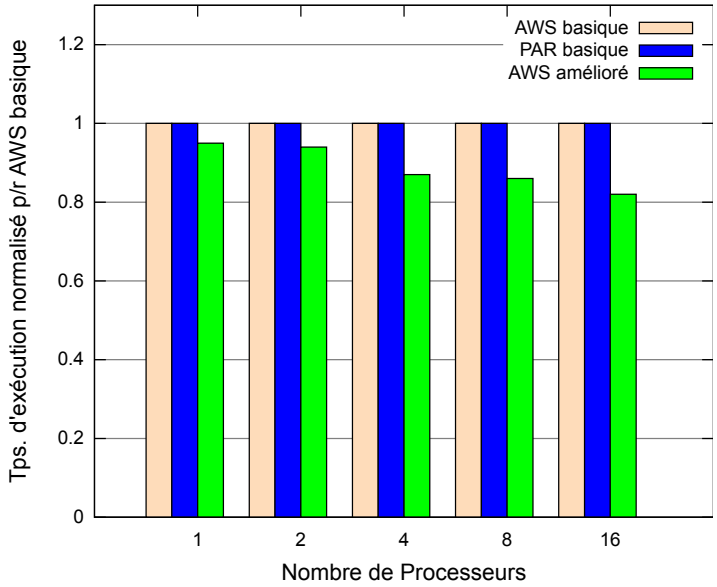


FIG. 4.11 – Temps d'exécution de TNR pour le décodage de 4 images, normalisé p/r AWS basique

Cela montre que même pour des applications régulières, qui sont le pire cas pour les algorithmes AWS, le surcout lié à l'équilibrage dynamique de charge reste très limité et que les résultats obtenus sont très acceptables.

4.6.3 Présentation de l'application Mandelbrot

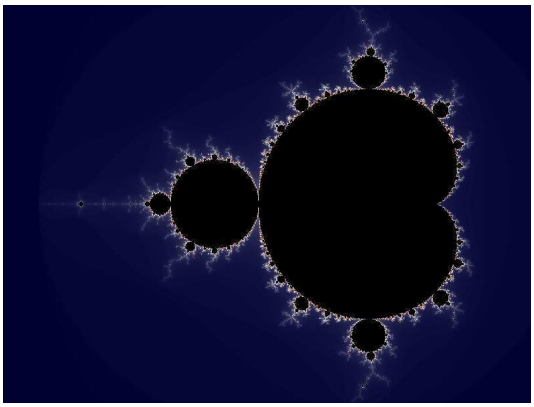
L'application Mandelbrot consiste à créer une séquence d'images représentant un zoom sur un point localisé sur le bord de l'ensemble, de manière à créer en sortie une vidéo de type Mjpeg. Le calcul consiste à vérifier si la suite $z_{n+1} = z_n^2 + c$ avec $c \in \mathbb{C}$ et $z_0 = 0$ converge vers un point de $\mathbb{C}$. Ce calcul accède très peu de données et prend place entièrement dans le cache.

Taille des images	800×600
Nombre d'itérations maximum	jusqu'à 5000 pour l'image 4
Coordonnées du centre	$(-0.74364421961; 0.13182604688)$
Zoom relatif p/r à l'image 1	$\{1, 5.64, 1.02 \times 10^6, 4.53 \times 10^6\}$

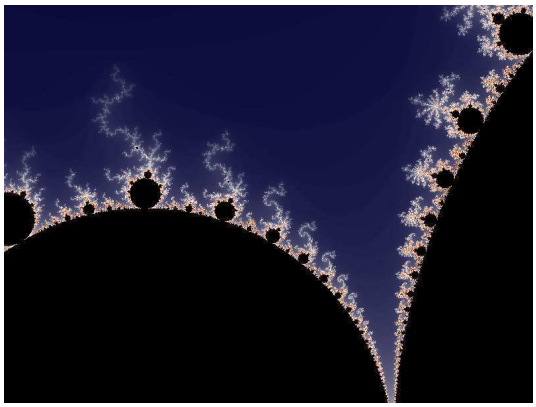
TAB. 4.3 – Paramètres des images calculées pour Mandelbrot

Par opposition à TNR, cette application ne peut pas être parallélisée efficacement *a priori* puisque l'on ne sait pas à l'avance pour quels pixels les calculs vont rapidement diverger ou au contraire converger. Pour la simulation, nous avons lancé le calcul de 4 images (représentées figure 4.12) sur 4 processeurs, et pour une de ces images, nous avons fait des simulations pour un nombre de processeurs allant de 1 à 16. Les paramètres de l'application sont présentés dans la table 4.3. Les architectures utilisées pour les simulations sont les architectures *AWS basique* et *PAR basique*.

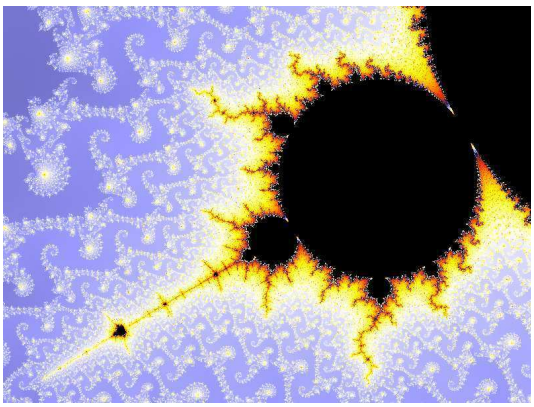
Le nombre maximum d'itérations est choisi de telle sorte que l'image ait un bon rendu



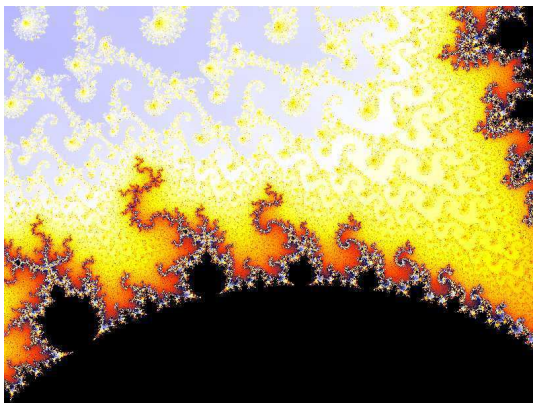
(a) Image 1



(b) Image 2



(c) Image 3



(d) Image 4

FIG. 4.12 – Images calculées en simulation

final, i.e. assez élevé pour que tous les points convergeant soient colorés (et non considérés divergents).

4.6.4 Résultats pour Mandelbrot

Les résultats des calculs pour 4 processeurs sont montrés dans la table 4.4.

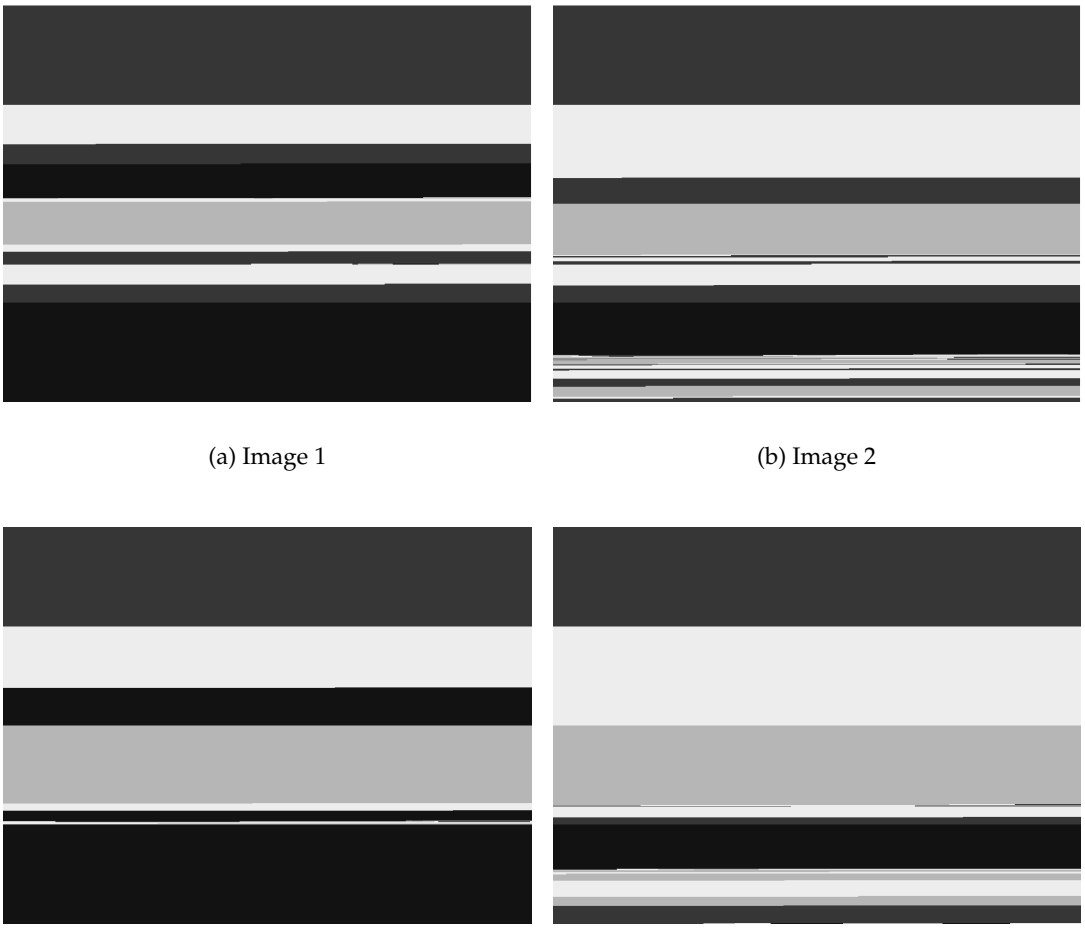
Ces résultats montrent qu'AWS surpasse le PAR sur cette application, avec une accélération variant de 1.5 à 2.31. Même si le nombre d'images est trop faible pour permettre une généralisation, cela prouve néanmoins qu'AWS fait mieux qu'une parallélisation statique dans le cas de calculs parallèles déséquilibrés.

En plus du calcul des images dans AWS, nous affichons pour chacune d'elle une image identifiant le processeur ayant calculé un pixel particulier (figure 4.13). Cela permet de visualiser la répartition du travail sur différents processeurs dans ces cas particuliers, et la manière dont le mécanisme de vol de travail impacte le calcul – notamment le fait que la plupart des vols ont lieu vers la fin.

Enfin, la figure 4.14 montre les temps de calcul de l'image #4 pour 1 à 16 processeurs. Cette figure montre que le temps gagné par l'équilibrage dynamique de charge fourni par

Image	Temps de calcul avec PAR	Temps de calcul avec AWS	Accélération relatif p/r au PAR	Nombre de vols	% de pixels volés
1	5 212	2 958	1.76	34	29.8
2	11 995	6 139	1.95	49	30.7
3	20 549	13 706	1.50	19	15.0
4	59 269	25 591	2.31	42	18.9

TAB. 4.4 – Résultats pour Mandelbrot sur 4 processeurs (temps en Kcycles)



(a) Image 1 (b) Image 2

(c) Image 3 (d) Image 4

FIG. 4.13 – Représentation visuelle du travail sur les processeurs pour les images calculées. Légende : ■ Proc. 0, ■ Proc. 1, ■ Proc. 2, ■ Proc. 3.

AWS n'est pas dépendant du nombre de processeurs, compte tenu du fait qu'il y a un facteur d'environ 2 pour 2, 4, 8 et 16 processeurs.

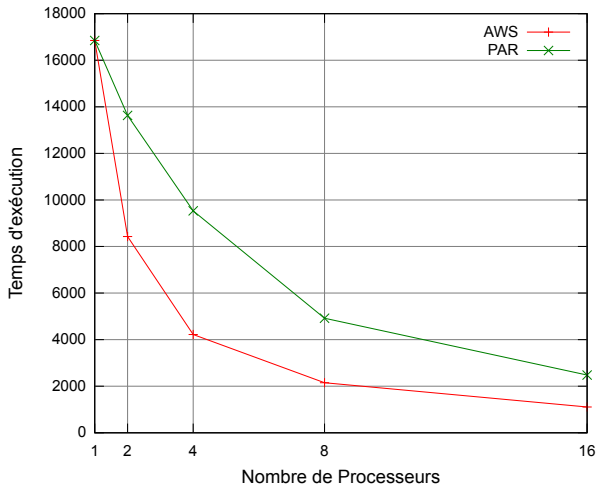


FIG. 4.14 – Temps d'exécution pour le calcul de l'image #4

4.6.5 Comparaison avec une solution centralisée de répartition de charges

4.6.5.1 Motivation et description

Si la stratégie PAR correspond à une parallélisation naïve, elle n'est pas viable à gros grain lorsque le temps de calcul varie fortement d'une entité à l'autre. Ainsi, pour l'application Mandelbrot vue au-dessus, il est peu réaliste d'adopter une telle solution.

Une approche plus correcte dans ce cas et différente du vol de travail consiste à avoir une structure de travail partagée unique, de laquelle chaque nœud extrait du travail lorsqu'il n'en a plus. Ainsi, le travail est réparti, mais le travail restant est centralisée, contrairement à AWS où il est distribué. Par ailleurs, la quantité de travail extraite par chaque nœud peut être fixe ou variable.

Nous implémentons donc une telle solution dans AWS, avec 2 variantes :

- une solution centralisée non adaptative (*CEN NAD*) : lors de chaque extraction, k éléments sont retirés du travail global
- une solution centralisée adaptative (*CEN AD*) : lors de chaque extraction $\alpha \times \log_2(n)$ éléments sont retirés, pour n éléments restants (α identique à celui d'AWS)

Le but recherché ici est de montrer la viabilité d'AWS face à une solution centralisée de répartition de charges.

4.6.5.2 Expérimentations et résultats

Les expérimentations sont menées sur une application irrégulière à gros grain (Mandelbrot) et une application régulière à grain fin : le micro-kernel, préféré ici à TNR pour des raisons de temps de simulation mais aussi car son grain est plus fin.

Les paramètres des applications sont donnés dans la table 4.5, et les conditions d'expérimentation sont les suivantes : les simulations sont faites sur une architecture avec des caches, mais sans autre modification par rapport à l'architecture de base (voir figure 4.3).

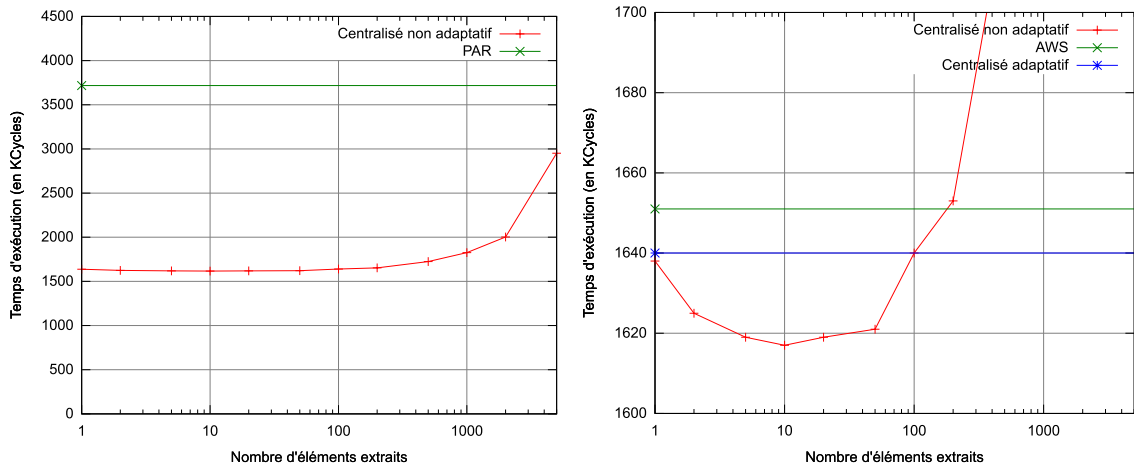
Pour chaque application, 4 configurations sont expérimentées : PAR, AWS, CEN AD et CEN NAD. Dans ce dernier cas, différentes valeurs sont utilisées comme nombre d'éléments k à extraire (de 1 à 5 000). Pour plus de clarté les résultats sont présentés sur deux graphes ayant des échelles différentes en y.

Communs	Valeurs de k	1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000
	Nombre de processeurs	4
	Ration d'extraction α	4
Mandelbrot	Résolution	200×150
	Coordonnées du centre	[0.74364424 ; 0.13182604]
	Étendue horizontale	0.00000066
	Nombre d'itérations max.	5 000
Micro-kernel	Nombre d'éléments	100 000

TAB. 4.5 – Paramètres des simulations pour la comparaison à une solution centralisée

L'impact de k est double : un petit k maximise le parallélisme potentiel mais induit aussi un surcout d'extraction qui peut devenir important. À l'inverse, un grand k minimise le surcout d'extraction, mais limite potentiellement le parallélisme.

Les figures 4.15 (a) et (b) montrent les résultats pour Mandelbrot.



(a) Échelle réduite

(b) Échelle agrandie

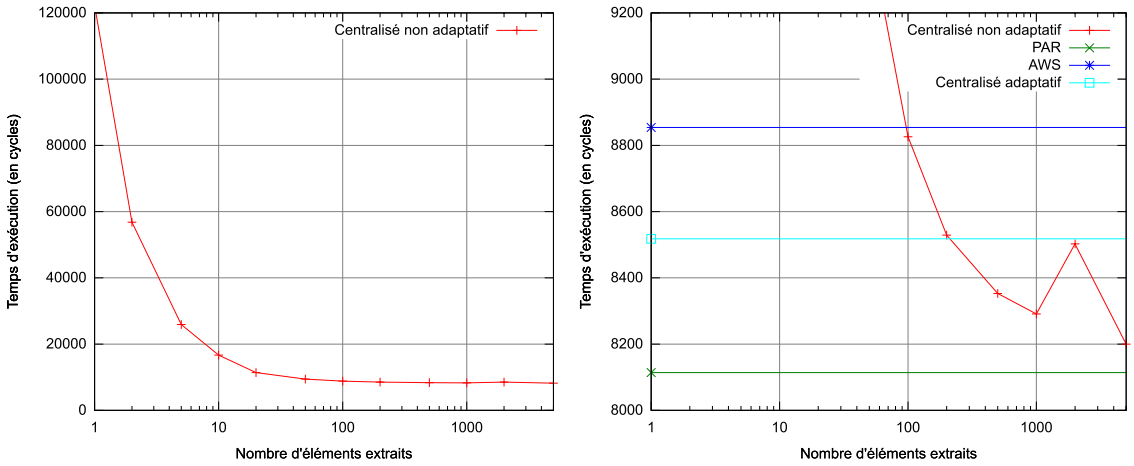
FIG. 4.15 – Comparaison des performances des différentes stratégies sur l'application Mandelbrot

On peut voir sur ces figures que la solution CEN NAD fait mieux que la solution AWS pour les valeurs de k inférieures à 200, et mieux que la solution CEN AD pour les valeurs de k inférieures à 100. Néanmoins, par rapport à la valeur de k optimale, les surcouts restent faibles puisqu'ils sont respectivement de 1.4% et 2.1% pour CEN AD et AWS.

La solution PAR est quand à elle la plus lente, avec un surcout de 130% par rapport la solution CEN NAD pour la valeur de k optimale. Elle peut être vue comme la solution CEN NAD avec un k égal au nombre d'éléments total divisé par le nombre de nœuds.

Les figures 4.16 (a) et (b) montrent les résultats pour le micro-kernel.

Due à la nature de l'application, la tendance est cette fois-ci inversée : la meilleure solution est le PAR, et pour la solution CEN NAD, les résultats sont globalement meilleurs à mesure que k augmente. Les surcouts des solutions CEN AD et AWS sont respectivement



(a) Échelle réduite

(b) Échelle agrandie

FIG. 4.16 – Comparaison des performances des différentes stratégies sur l'application Mandelbrot

de 5% et 9%, soit plus élevés que pour Mandelbrot, mais relativement raisonnables. La solution CEN NAD avec un k de 1 a un surcout supérieur à 1400%, ce qui montre que le surcout d'extraction peut être prohibitif à grain fin, bien qu'il s'agisse ici d'un cas extrême.

4.6.5.3 Conclusion

Une solution centralisée avec un nombre d'éléments extraits constant k donne toujours les meilleurs résultats pour une certaine valeur de k. Néanmoins, les solutions adaptatives possèdent l'avantage de fournir des bonnes performances sans avoir besoin de connaitre le grain de l'application, ce qui est requis pour trouver une bonne valeur de k. Comme leurs surcouts sont globalement faibles, on s'orientera en général vers une solution adaptative.

Par ailleurs, au vu de ces résultats, la solution centralisée adaptative semble donner des meilleurs résultats qu'AWS (distribuée). Néanmoins, cette comparaison est à approfondir, notamment en fonction de la latence du réseau : pour une latence élevée, une solution distribuée peut s'avérer plus avantageuse.

4.7 Conclusion de l'étude

Dans la première partie de ce chapitre, nous nous sommes donc concentrés sur le problème des choix de conception simples des architectures MPSoC en vue de l'utilisation des algorithmes basés sur la bibliothèque AWS. Nous avons montré que l'ensemble des choix de conception défini au niveau matériel et logiciel permettait d'obtenir des améliorations de performances à la fois pour le PAR et AWS.

Nous nous sommes ensuite concentrés sur le surcout induit par AWS comparé à des algorithmes parallèles ordonnancés de manière statique, et nous avons mis en évidence que ce surcout lié à l'équilibrage dynamique de charge n'était pas négligeable quand les données à traiter étaient de petite taille. Nous avons aussi trouvé que distribuer les données locales aux nœuds ou les verrous qui sont principalement accédés localement permet un gain de

performance dans le cas d'AWS. Enfin, nous avons appliqué ces résultats à deux applications réelles et montré qu'AWS pouvait entrainer un gain significatif de performances dans le cas d'applications ayant une charge de travail non uniforme, tout en ayant un surcout très limité pour les applications ayant une charge de travail uniforme.

Nous avons finalement comparé la solution AWS à deux solutions centralisées de répartition de charge : une adaptative et une non adaptative. Les résultats montrent qu'une solution adaptative (centralisée ou distribuée) permet d'avoir des bonnes performances quelque soit le grain de l'application au prix d'un surcout faible. Par ailleurs, la solution centralisée adaptative donne des meilleurs résultats qu'AWS dans les expérimentations faites, mais une étude prenant en compte la latence d'accès aux données centralisées reste à faire.

Le modèle d'architecture choisi dans ce travail, bien que réaliste pour les MPSoCs, reste simple et ne prend pas en compte la possibilité d'avoir des plus hauts niveaux de hiérarchie mémoire et de réseaux d'interconnexion, ou encore la possibilité d'avoir plusieurs processeurs situés sur un même réseau d'interconnexion local. Nous avons aussi limité notre étude à 16 processeurs, bien que dans un futur proche, des architectures ayant un plus grand nombre de cœurs verront le jour.

Nous croyons néanmoins que le paradigme de programmation d'AWS peut être utile en vue des architectures intégrant beaucoup de cœurs, y compris dans le domaine de l'embarqué. En ce sens, nous pensons que ce travail est important en tant que première étude de cette classe d'applications sur les architectures MPSoC.

4.8 Analyse du cœur de l'algorithme d'AWS

Comme nous l'avons vu au chapitre 3, la partie fonctionnelle – ou cœur – d'AWS est constituée des fonctions effectuant l'équilibrage dynamique des charges de travail. Ce sont elles qui effectuent les appels aux fonctions `extract_par()`, `extract_seq()` et `local_run()` définies par l'utilisateur. La deuxième partie de ce chapitre s'intéresse aux algorithmes de cette partie fonctionnelle d'AWS.

En particulier, cette partie fonctionnelle est découpée en deux fonctions. La première, `loop_core_adaptive`, présentée au chapitre 3 est principalement constituée de deux boucles imbriquées : la boucle la plus externe est empruntée tant qu'il reste du travail dans le système global, tandis que la boucle interne est empruntée tant qu'il reste du travail local. Ainsi, lorsqu'un processeur sort de la première boucle, il tente ensuite de voler les autres processeurs tour à tour. En cas de succès, il recommence la boucle interne, tandis qu'en cas d'échec, cela signifie que tout le travail est terminé (ou en passe de l'être). La deuxième fonction, appelée `steal`, est simplement l'externalisation des vols.

Comme vu dans la problématique, le modèle de programmation lock-free permet de répondre à certains des problèmes posés par les verrous. Dans certains cas, il permet aussi des implémentations plus efficaces qu'avec des verrous. Dans ce but, nous étudions une implémentation lock-free du cœur de l'algorithme. Une deuxième implémentation alternative reposant sur l'accélération des extractions séquentielles est aussi présentée.

Les applications utilisées pour l'évaluation de ces 3 implémentations sont les mêmes que précédemment. Néanmoins, contrairement aux expérimentations précédentes, toutes les expérimentations relatives à ce travail sont faites en natif, sur des cœurs de type x86.

4.9 Algorithme original

Cette section présente l'algorithme original, qui utilise les verrous. Comme la fonction `loop_core_adaptive` a déjà été donnée au chapitre 3, nous donnons seulement dans cette section la fonction de vol.

AWS définit un type nœud (`node_t`), un nœud étant une abstraction du travail. Dans le cadre de notre travail dans AWS, nous avons systématiquement associé au maximum un nœud par processeur. En effet, mettre un nombre de nœud supérieur au nombre de processeurs n'a pas vraiment de sens dans ce contexte. De manière générale, nous ne ferons dans la suite pas de distinction entre un nœud et un cœur de calcul.

Dans l'algorithme original d'AWS, chaque structure `work_t` est protégée par un verrou, qu'il faut donc acquérir avant d'extraire une partie de ce travail localement ou par l'intermédiaire d'un vol. Chaque nœud peut être volé pendant qu'il exécute sa routine `local_run()`. La fonction `steal` est donnée ci-après.

```
1  /* Essaie de voler tour à tour du travail de tous les autres noeuds
2   * node : noeud voleur
3   * stolen : si une des tentatives de vol est un succès,
4   *          contient le travail volé
5   */
6  status_t steal(node_t* node, work_t* stolen){
7    node_t* remote;
8
9    /* Initialisation de la victime */
10   int current = victim_reset(node);
11
12   /* Tentatives de vol successives sur tous les autres noeuds */
13   for (i = 0; i < aws_global->nodes_nb - 1; i++){
14     remote = aws_global->nodes[current];
15     lock(&remote->mutex);
16     status = extract_par(remote->work, stolen);
17     if (status == STATUS_OK){
18       unlock(&remote->mutex);
19       break;
20     }
21     else {
22       unlock(&remote->mutex);
23       current = victim_next();
24     }
25   }
26   return status;
27 }
```

4.10 Algorithme lock-free

Comme nous nous sommes intéressés à la programmation lock-free, il nous a paru intéressant d'essayer de changer le cœur adaptatif d'AWS pour le rendre lock-free, de manière à voir s'il était possible de gagner en performances. Nous nous sommes autorisés à utiliser les primitives CAS simple et CAS sur 2 mots contigus (appelé CAS2 dans la suite, mais qui n'est pas un CAS-2) qui sont décrites dans le chapitre 2, et que nous avons implémentées à l'aide de macros C et des instructions x86 cmpxchg et cmpxchg8b (voir annexe B pour le détail de ces macros).

Nous avons conçu et réalisé un certain nombre d'algorithmes lock-free (5 au total), dont la plupart étaient inefficaces, ou alors présentaient des erreurs à l'exécution. En particulier, du fait de l'utilisation de techniques comme l'insertion de code assembleur dans le code C, ou le non respect d'un typage strict pour avoir des performances acceptables, certains de ces bogues ne se manifestaient que lors de l'utilisation d'optimisations du compilateur (à partir de -O1), rendant donc leur correction très difficile. Nous avons aussi étudié une solution basée sur la *Safe Memory Reclamation* [Mic04], mais qui s'est avérée non concluante. Au final, seule une solution entièrement lock-free s'est avérée concluante ; c'est celle que nous présentons ici.

4.10.1 Philosophie et structures de l'algorithme

Cette solution utilise la comparaison de pointeurs avec versionnement sur les structures de travail. En effet, une solution basée uniquement sur la comparaison de pointeurs serait fausse pour une raison bien connue du domaine de programmation lock-free, et qui porte le nom de problème ABA. Ce problème vient du fait que la primitive CAS ne pourrait pas ici détecter que le pointeur n'est plus à jour si deux extractions (ou plus) laissaient le pointeur de référence pointer vers la même case que lors de la lecture initiale. Rajouter un numéro de version au pointeur permet d'éviter ce problème, au prix d'un CAS sur deux mots.

Plus précisément, chaque nœud se voit attribuer deux tableaux de taille (NODES_NUMBER + 1), NODES_NUMBER étant le nombre de nœuds dans le système.

Le premier tableau (`work`) est un tableau de structures `work_t` qui contient d'une part la structure de référence définissant le travail restant pour ce nœud, appelé dans la suite travail de référence, et NODES_NUMBER autres structures d'autre part. Ces autres structures servent à contenir des valeurs spéculatives du travail de référence suite à l'extraction de travail liée à un vol ou à une extraction locale de travail. Le deuxième tableau (`ptr`) est un tableau de pointeurs, et sert à faire la correspondance entre les nœuds et les différentes structures de travail du tableau `work`, ainsi que le lien vers la structure `work_t` de référence.

Par convention, la structure de référence est celle pointée par la case (NODES_NUMBER) du tableau `ptr`, et la structure utilisée pour l'extraction de travail par le nœud i est la structure pointée par la case i du tableau `ptr`.

Lors d'une mise à jour, les champs du pointeur et de la version du tableau `ptr` sont mis à jour simultanément, par l'intermédiaire de la primitive CAS2.

4.10.2 Mécanisme de vol

Lorsqu'un processeur souhaite en voler un autre, il commence par recopier de manière atomique le pointeur et le numéro de version du travail de référence dans une structure temporaire (`wd_old` dans l'algorithme). Il essaie ensuite de voler le travail contenu dans cette structure, et de placer le résultat dans la structure `work_t` qui lui est allouée sur ce nœud.

Si le vol réussit, le processeur essaie ensuite de changer la structure correspondant au travail de référence : cette structure est composée d'un nouveau numéro de version et du travail restant après le vol. Le changement est fait à l'aide d'un CAS2 : si la structure de référence actuelle est toujours telle qu'elle était avant le vol, alors pointeur et version de cette structure de référence sont mis à jour de manière atomique. Le processeur met alors à jour la structure `work_t` pour sa prochaine tentative de vol sur ce nœud (il s'agit de l'ancienne structure `work_t` de référence).

La figure 4.17 illustre les structures de données et les mécanismes propres à cet algorithme à travers un exemple de vol.

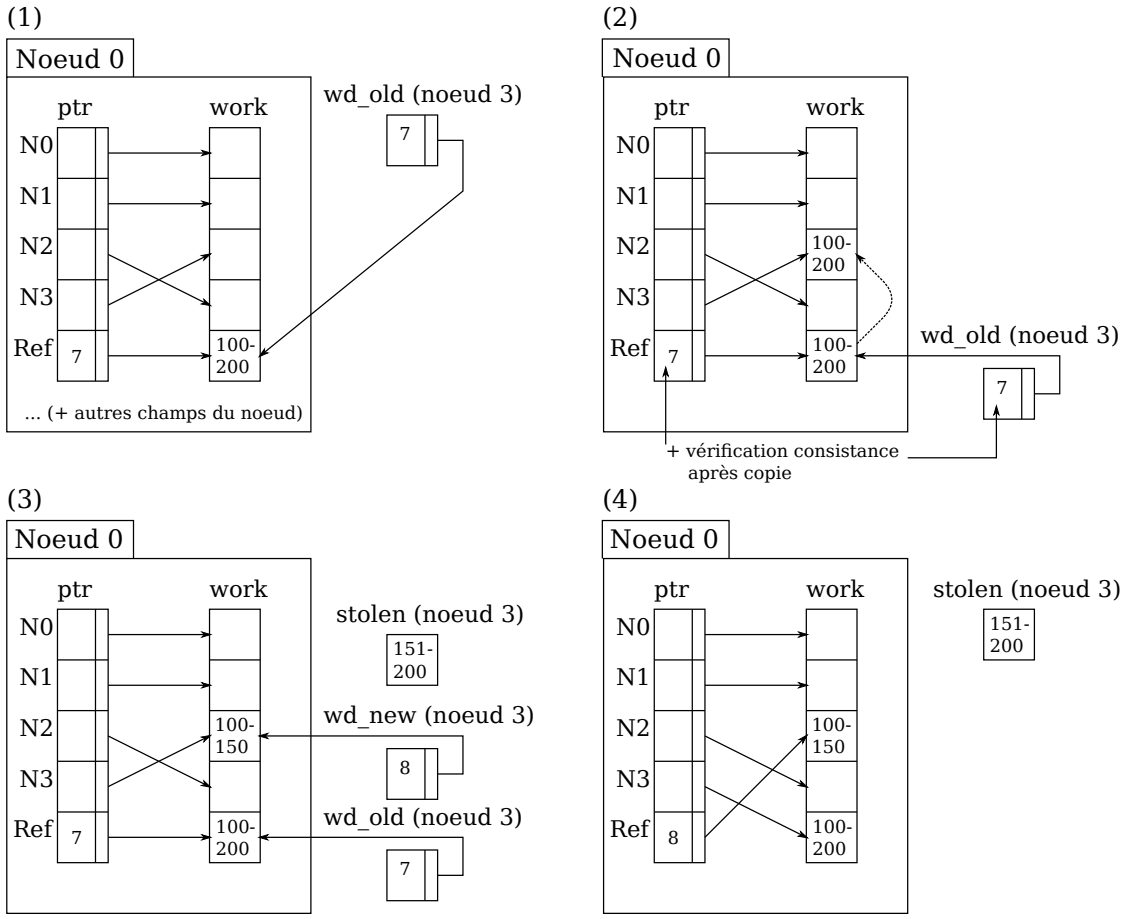


FIG. 4.17 – **Exemple de vol avec la version lock-free du cœur d'AWS et 4 nœuds**
Cet exemple montre sur un cas simple l'évolution des données lors d'un vol. Dans l'état initial, le travail de référence du nœud 0 en est à la version 7.
(1) Le nœud 3 souhaite voler le nœud 0, et commence donc par recopier de manière atomique le pointeur vers le travail de référence et le numéro de version du nœud 0. Le résultat de cette copie est placé dans sa structure `wd_old`.
(2) Le nœud 3 copie ensuite le contenu du travail courant dans sa copie personnelle, sur laquelle il effectuera le vol. À la fin de la copie, il effectue un test de consistance des données copiées en vérifiant que le numéro de version n'a pas changé.
(3) Le nœud 3 effectue le vol sur sa copie personnelle par l'appel à la fonction `extract_par()`. Les opérations n'ont pas besoin d'être atomiques, et le résultat se trouve dans la structure `wd_new`.
(4) Numéro de version et pointeur du travail de référence sont mis à jour de manière atomique, par l'intermédiaire de la primitive CAS2. Si le numéro de version avait changé entre temps, la mise à jour n'aurait pas eu lieu.

4.10.3 Mise à jour locale des données volées ou extraites

Un point important concerne le fait qu'une fois sorti de l'appel à la fonction `steal`, le processeur doit mettre à jour son nœud local avec les données volées ; cette mise à jour est faite de manière atomique, i.e. on ne copie pas directement le nouveau `work_t` à la place de l'ancien, mais on passe encore une fois par une structure intermédiaire (en l'oc-

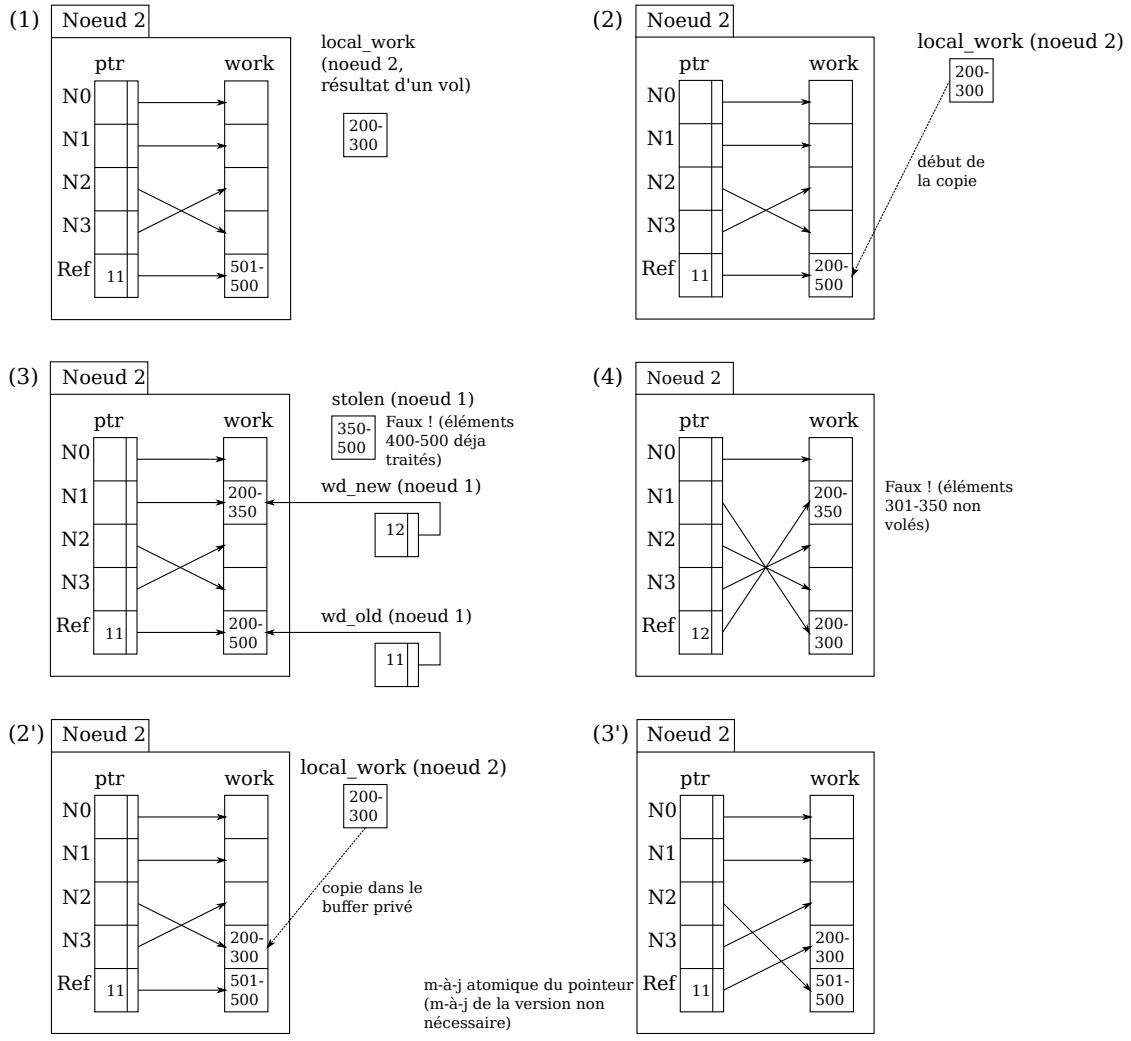


FIG. 4.18 – **Exemple de mise à jour des données locales après un vol avec et sans atomicité**
Cet exemple montre l'importance de l'atomicité de la recopie d'un travail volé.
(1) Initialement, le nœud 2 n'a plus de travail à faire localement. Il obtient suite à un vol une nouvelle plage d'éléments à traiter.
(2) On suppose que la recopie des nouvelles valeurs n'est pas atomique. Le nœud 2 commence donc la recopie du travail volé vers sa structure de référence.
(3) Au milieu de cette copie, le nœud 1 tente un vol. La plage de valeurs qu'il voit alors est inconsistante. Les éléments volés sont faux.
(4) Le nœud 1 termine son vol. Les éléments restants au nœud 2 sont faux. Le nœud 2 finit sa recopie, mais celle-ci n'a plus lieu dans le travail de référence.
(2') (alternative à 2) Cette fois, la recopie des nouvelles valeurs est atomique. Le nœud 2 utilise sa structure de travail temporaire pour la recopie.
(3') (suite de 2') Le pointeur est mis à jour, puis la structure temporaire pour le travail du nœud 2 est mise à jour. On ne met pas à jour la version car cela n'est pas nécessaire.

currence celle qui serait utilisée pour l'extraction locale de travail) pour éviter qu'une recopie de ces données ait lieue alors qu'elles sont dans un état inconsistant. Un test de consistance des données copiées a aussi lieu avant l'appel à la fonction `extract_par()` (resp.

`extract_seq()`). Ce dernier est important car il est possible que la fonction `extract_par()` (resp. `extract_seq()`) produise une erreur à l'exécution si les données passées en entrée sont inconsistantes, et que ce cas n'a pas été prévu par l'utilisateur dans l'écriture de ces fonctions. Néanmoins, même sans ce test, un vol ou une extraction locale serait impossible car la version ne serait plus la bonne au moment du test avec le CAS2 en fin d'extraction.

La figure 4.18 montre sur un exemple pourquoi il est nécessaire d'avoir une copie locale atomique de données volées.

L'extraction de travail séquentielle se passe globalement de la même manière : le processeur utilise alors simplement la structure lui correspondant tout comme s'il s'agissait d'un vol.

4.10.4 Algorithme

On définit les types suivants :

```
typedef struct {
  aws_work_t* ptr;
  unsigned long int version;
} work_desc_t;

typedef union {
  work_desc_t part;
  unsigned long long int all;
} work_desc;

typedef struct {
  work_desc ptr[NODES_NUMBER+1];
  work_t work[NODES_NUMBER+1];
} node_work_set;
```

On rajoute dans la définition du type nœud :

```
typedef struct {
    ...
    node_work_set work_set;
    ...
} node_t;
```

La fonction `steal` devient la suivante. Les lignes modifiées par rapport à l'algorithme original sont les lignes en grisé.

```
status_t steal(node_t* node, work_t* stolen){
  node_t* remote;
  int self_index = node->cpu;
  char* local_work_p;
  work_desc wd_old, wd_new;

  /* Initialisation de la victime */
  int current = victim_reset(node);

  /* Tentatives de vol successives sur tous les autres noeuds */
  i = 0;
  while (i < aws_global->nodes_nb-1){
    remote = aws_global->nodes[current];
    local_work_p = remote->work_set.ptr[self_index].part.ptr;
    wd_old.all   = remote->work_set.ptr[NODES_NUMBER].all;
```

```
16
17      memcpy(local_work_p,wd_old.part.ptr,AWS_WORK_SIZE_MAX);
18
19      /* Pour garantir la consistance des données copiées */
20      if (wd_old.part.version != remote->work_set.ptr[NODES_NUMBER].part.
            version){
21        continue;
22      }
23      status = extract_par(local_work_p,stolen,2);
24      if (status == STATUS_OK){
25        wd_new.part.version = wd_old.part.version + 1;
26        wd_new.part.ptr = local_work_p;
27        if (cas2(&remote->work_set.ptr[NODES_NUMBER].all,wd_old.all,wd_new.
              all)){
28          remote->work_set.ptr[self_index].part.ptr = wd_old.part.ptr;
29          break;
30        }
31        /* Si le CAS échoue, on retente de voler le même noeud */
32      }
33      else {
34        current = victim_next();
35        i++;
36      }
37    }
38    return status;
39  }
```

La fonction `loop_core_adaptive` devient quant à elle la suivante.

```
1  void loop_core_adaptive(node_t* node){
2    int self_index = node->cpu;
3    char *local_work_p;
4    char *t;
5    work_desc wd_old, wd_new;
6    bool has_global_work = true;
7    bool has_local_work = true;
8    char local_work[AWS_WORK_SIZE_MAX];
9
10   while(has_global_work){     /* steal-loop */
11     while(has_local_work){    /* local-loop */
12       local_work_p = node->work_set.ptr[self_index].part.ptr;
13       wd_old.all    = node->work_set.ptr[NODES_NUMBER].all;
14
15       memcpy(local_work_p,wd_old.part.ptr,AWS_WORK_SIZE_MAX);
16
17       /*  Pour garantir la consistence des données copiées */
18       if (wd_old.part.version != node->work_set.ptr[NODES_NUMBER].part.
             version){
19         continue;
20       }
21
22       status = extract_seq(local_work_p,&local_work);
23
24       if (status == STATUS_OK){
```

```
25          wd_new.part.version = wd_old.part.version + 1;
26          wd_new.part.ptr = local_work_p;
27          if (cas2(&node->work_set.ptr[NODES_NUMBER].all,wd_old.all,wd_new.
                all)){
28            node->work_set.ptr[self_index].part.ptr = wd_old.part.ptr;
29            local_run(&local_work[0],aws_global->internal_user_data);
30          }
31        }
32        else { /* STATUS_KO */
33          has_local_work = false;
34        }
35      } /* fin de la local-loop */
36
37      status = steal(node,&local_work[0]);
38      if (status == STATUS_OK){
39        /* Mise à jour du work_t de référence de manière atomique, pour
40         * garantir la consistance des données que l'on copie dans le cas
41         * où un autre noeud essaie de voler en meme temps (et que par
42         * exemple les memcpy ne vont pas à la meme vitesse)
43         * Note : on est sûrs que les vols en parallèle ont échoué,
44         *        on n'a pas besoin de tester la consistance avec un CAS
45         */
46        memcpy(node->work_set.ptr[self_index].part.ptr,local_work,
              AWS_WORK_SIZE_MAX);
47
48        t = node->work_set.ptr[NODES_NUMBER].part.ptr;
49        node->work_set.ptr[NODES_NUMBER].part.ptr = node->work_set.ptr[
              self_index].part.ptr;
50        node->work_set.ptr[self_index].part.ptr = t;
51        has_local_work = true;
52      }
53      else {
54        has_global_work = false;
55      }
56    } /* fin de la steal-loop */
57  }
```

4.11 Algorithme visant une granularité faible

Nous proposons enfin une dernière implémentation du cœur de l'algorithme d'AWS : l'idée derrière cet algorithme est de tirer parti du fait que les vols sont rares et de maximiser la boucle d'extraction locale (`local-loop`), même s'il faut pour cela ralentir le temps pris pour les vols. Le point clé de l'algorithme consiste à ne pas devoir prendre un verrou à chaque fois que l'on veut faire un appel à `extract_seq()`, mais au contraire à le libérer lorsqu'un autre nœud souhaite voler. Lorsque c'est le cas, le nœud souhaitant faire un vol incrémente une variable qui est vérifiée entre deux appels à `extract_seq()` (`steal_request`). Si lors de la vérification, cette variable est différente de 0, le nœud volé se suspend jusqu'à ce que tous les vols aient eu lieu. Deux variantes de cet algorithme sont proposées : une où la variable `steal_request` est protégée par un verrou (nous l'appellerons `AWS Lock Lock`) et une où elle est modifiée à l'aide de CAS (appelée `AWS Lock CAS`). Ce choix contrôlé à l'aide du flag NO_CAS. Les résultats de ces deux variantes sont sensiblement équivalents, mais les deux implémentations sont néanmoins présentées.

On modifie la définition du type nœud en rajoutant le(s) champ(s) suivant(s) :

```
typedef struct {
    ...
    volatile int steal_request;
    #ifdef NO_CAS
      mutex_lock steal_request_lock;
    #endif
    ...
} node_t;
```

La fonction de vol reste assez proche de la fonction originale, à ceci près qu'elle doit incrémenter et décrémenter la variable `steal_request`. On définit alternativement la commande `INC_STEAL_REQUESTS` par l'ensemble d'instructions suivantes :

```
#ifdef NO_CAS
   #define INC_STEAL_REQUESTS ({              \
        lock(&remote->steal_request_lock);    \
        remote->steal_request++;              \
        unlock(&remote->steal_request_lock); \
        })
#else
   #define INC_STEAL_REQUESTS ({              \
       do {                                   \
         temp = remote->steal_request;        \
         temp2 = temp + 1;                    \
       } while (!cas(&remote->steal_request,temp,temp2)); \
       })
#endif
```

On définit de la même manière la commande `DEC_STEAL_REQUESTS`. La boucle principale de la fonction `steal` devient la suivante :

```
  /* Tentatives de vol successives sur tous les autres noeurds */
  for (i = 0; i < aws_global->nodes_nb - 1; i++){
    remote = aws_global->nodes[current];
    INC_STEAL_REQUESTS;
    lock(&remote->mutex);
    status = extract_par(remote->work, stolen);
    if (status == STATUS_OK){
      unlock(&remote->mutex);
      DEC_STEAL_REQUESTS;
      break;
    }
    else {
      unlock(&remote->mutex);
      DEC_STEAL_REQUESTS;
      current = victim_next();
    }
  }
```

La fonction principale effectue un test sur la variable `steal_request` avant chaque appel à la fonction `extract_seq()`, mais sans avoir besoin de protéger l'accès du fait que la variable n'est pas modifiée. La boucle principale de la fonction `loop_core_adaptive` est modifiée de la manière suivante :

```
    while (has_local_work){ /* Local loop */
```

```
2         if (node->steal_request != 0){
3           unlock(node_mutex);
4           while (node->steal_request != 0);
5           lock(node_mutex);
6         }
7
8         status = extract_seq(shared_work, &local_work);
9
10        if(status == STATUS_OK){
11          local_run(&local_work, aws_global->internal_user_data);
12        }
13        else { /* STATUS_KO */
14          has_local_work = false;
15        }
16      } /* Fin de la local-loop */
```

4.12 Expérimentations et résultats

Si pour les expérimentations en simulation, nous avions fixé arbitrairement la valeur du ratio d'extraction séquentielle, évaluer les performances des différents versions du cœur de l'algorithme nécessite de prendre ce ratio en paramètre. Cela est possible compte tenu des temps de simulation natifs. Pour rappel, ce ratio doit être défini pour chaque application afin de minimiser le temps d'exécution. Pour les expérimentations réalisées dans cette section, les valeurs du ratio seront toutes les puissances de 2 comprises entre 2 et 256.

De manière à évaluer les algorithmes présentés au-dessus, nous avons repris les mêmes applications que précédemment, en changeant quelques paramètres, notamment le nombre de simulations. Ces modifications sont résumées dans le tableau 4.6. Toutes les simulations ont été faites avec 16 nœuds sur une machine possédant 24 cœurs. Dans la suite, l'algorithme original du cœur d'AWS est noté `AWS orig`, l'algorithme lock-free présenté en section 4.10 `AWS LF`, et les deux dernières variantes de l'algorithme `AWS Lock CAS` et `AWS Lock Lock`.

TAB. 4.6 – Configuration pour les applications simulées en natif

Micro kernel	100 000 000 éléments
	Nombre de simulations : 1 000
Mandelbrot	Nombre d'itérations max. : 8 000
	Nombre de simulations : 100
TNR	20 frames traitées
	Nombre de simulations : 100

4.12.1 Résultats pour le micro-kernel

Les résultats pour le micro kernel sont présentés figures 4.19.

Comme attendu, les algorithmes `AWS Lock CAS` et `AWS Lock Lock` se comportent très bien dans le cas où le ratio d'extraction est faible. Néanmoins, ils se comportent tout aussi bien pour un ratio plus élevé : la figure montre que le temps d'exécution ne dépend pas du ratio pour ces algorithmes. Cela s'explique d'une part du fait du faible nombre de vols,

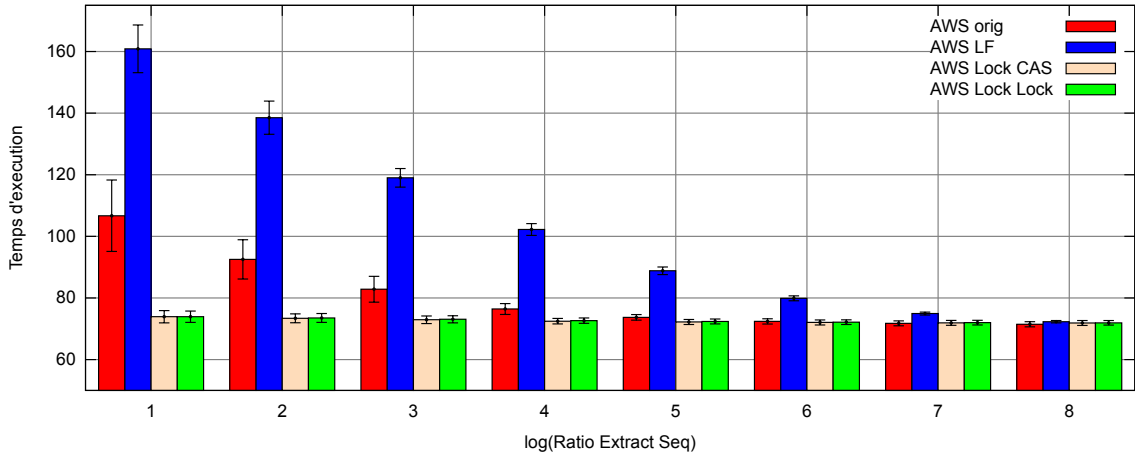


FIG. 4.19 – Temps d'exécution du micro-kernel en natif sur 16 cœurs avec 100 millions d'éléments, en fonction du ratio d'extraction séquentielle

et d'autre part du fait que le cout quasi nul de l'opération de traitement crée un grain artificiellement fin. À l'inverse, l'algorithme `AWS LF` est très mauvais pour un ratio faible et s'améliore lorsque le ratio augmente, jusqu'à atteindre les performances des algorithmes `AWS Lock CAS` et `AWS Lock Lock` pour un ratio de 8. L'algorithme `AWS orig` se situe entre les deux, avec des performances s'améliorant une fois encore à mesure que le ratio augmente.

Cela montre que les algorithmes `AWS Lock CAS/Lock` répondent bien au critère selon lequel ils ont été conçus : maximiser le temps d'exécution à grain très fin ou lorsqu'il y a peu de vols, ce qui est le cas dans ce micro-kernel. Cela laisse à penser que cet algorithme est particulièrement adapté pour AWS. Toutefois, si les vols sont rares en pratique, il reste à montrer que pour une application réelle, pour laquelle le grain est nécessairement plus élevé, le nombre de vols est suffisamment faible pour tirer parti de l'accélération de la `local-loop`.

4.12.2 Résultats pour les deux applications

Les résultats pour l'application Mandelbrot sont donnés figure 4.20 et ceux pour l'application TNR figure 4.21.

Ces graphes montrent que la tendance s'inverse : pour Mandelbrot, les algorithmes `AWS Lock CAS/Lock` ne sont viables pour aucune des valeur du ratio d'extraction séquentielle, avec des temps d'exécution qui croissent très vite à mesure que le ratio augmente. *A contrario*, l'algorithme `AWS LF` est tout aussi performant que l'algorithme original. Les résultats pour TNR confirment cette inversion de tendance : pour les valeurs du log du ratio supérieures à 2, l'algorithme `AWS LF` est le plus performant, tandis que les algorithmes `AWS Lock CAS` et `AWS Lock Lock` sont les plus lents. Ces derniers sont toutefois les plus efficaces pour les deux valeurs les plus faibles du ratio, mais sans gros avantage.

Pour ces valeurs de granularité, le nombre de vols est donc trop important pour qu'on puisse se permettre d'attendre la fin de l'exécution de la `local-loop` de la victime avant de pouvoir effectuer le vol. En résumé, avec une granularité de travail minimum, l'algorithme `AWS LF` est le plus performant quelle que soit la régularité de l'application, avec un net avantage pour les applications régulières.

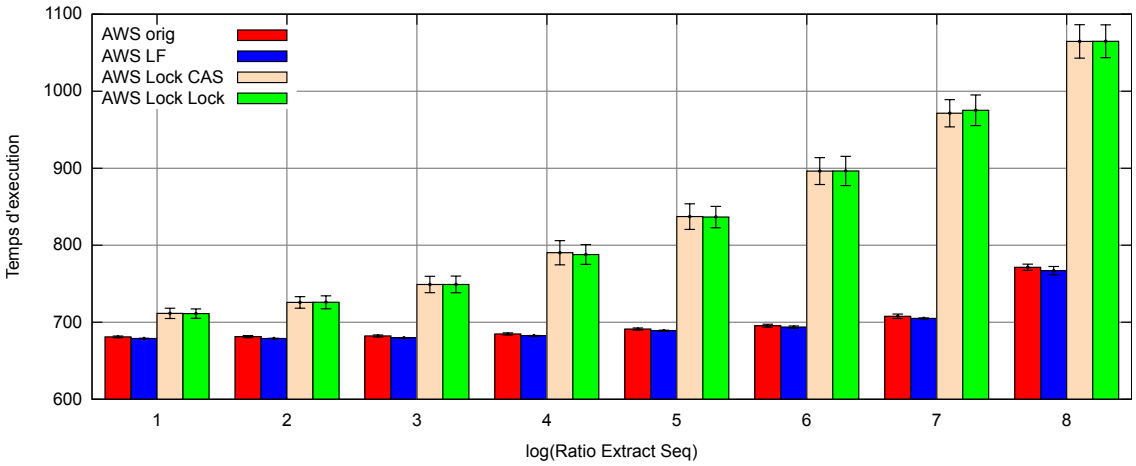


FIG. 4.20 – Temps d'exécution de Mandelbrot en natif sur 16 cœurs en fonction du ratio d'extraction séquentielle

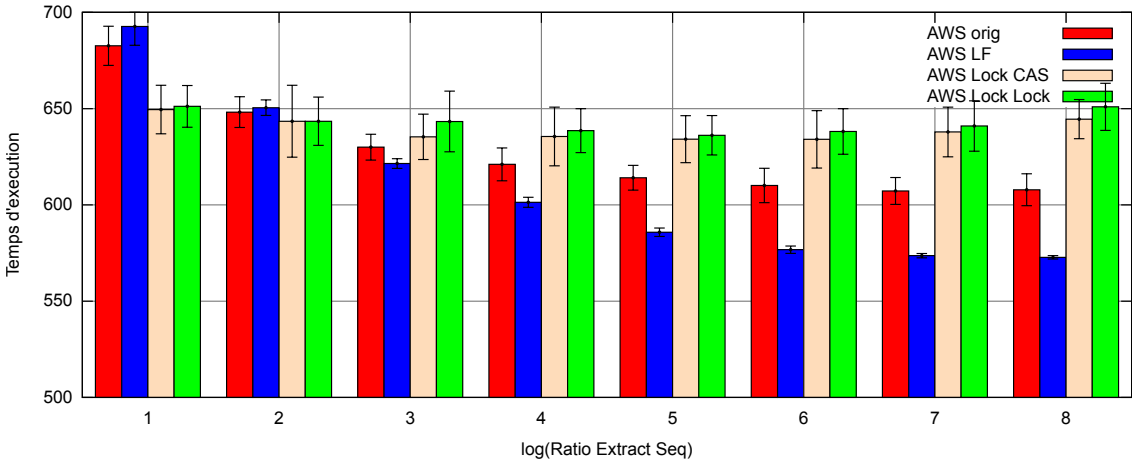


FIG. 4.21 – Temps d'exécution de TNR en natif sur 16 cœurs en fonction du ratio d'extraction séquentielle

4.13 Conclusion

Nous avons dans la seconde partie de ce chapitre comparé plusieurs versions du cœur d'AWS, montrant ainsi l'impact que peut avoir l'implémentation d'une bibliothèque sur les performances, notamment en fonction du modèle de programmation qu'elle utilise.

Les résultats de ces expérimentations semblent *a priori* surprenants car le cout de l'instruction CAS2 est élevé, mais ils supportent l'hypothèse selon laquelle certains algorithmes lock-free peuvent être meilleurs que leurs équivalents à base de verrous. En effet, si le faible nombre d'applications disponibles pour AWS ne permet pas de conclure de manière sûre, il semble que l'implémentation lock-free pour le cœur d'AWS soit la plus efficace, avec cependant une limite lorsque le grain de l'application devient trop fin.

Les résultats de la seconde partie de ce chapitre montrent enfin que malgré le faible nombre de vols, il est important qu'un vol puisse avoir lieu sur un nœud pendant que celui-ci traite du travail localement.

Chapitre 5

État de l'art sur les mémoires transactionnelles

Les mémoires transactionnelles proposent un paradigme de programmation qui a pris son essor récemment à travers de nombreux travaux de recherche. Les mécanismes à base de verrous pour garantir l'exclusion mutuelle sont connus pour être difficiles à utiliser et sujets aux erreurs. Aussi, le but d'un système TM est de fournir une primitive d'atomicité pour les programmes parallèles, nécessaire à la construction des transactions, ensemble d'instructions s'exécutant en isolation et de manière atomique du point de vue des autres processeurs. En ce sens, les mémoires transactionnelles peuvent être vues comme une généralisation des primitives lock-free.

Les systèmes TM matériels sont les systèmes requérant une implémentation matérielle des primitives permettant l'utilisation des transactions. Nous présentons dans ce chapitre le contexte des travaux existants relatifs aux systèmes TM matériels, dits systèmes HTM.

5.1 Terminologie

À défaut de bonnes traduction en français, nous emploierons dans la suite le terme *commit* pour désigner la fin d'une transaction et le fait que ses modifications soient visibles par tous les autres threads, et le terme *abort* pour désigner l'interruption d'une transaction suite à un conflit. Par extension, nous utiliserons les verbes *commiter* et *aborter*. Enfin, nous étendrons ces actions aux processeurs, i.e. un processeur commite si la transaction qu'il est en train d'exécuter commite.

Le terme transactionnel désigne ce qui a trait à une transaction. Ainsi, des données transactionnelles sont des données accédées durant une transaction.

5.2 Axes de conception des systèmes HTM

De manière à fournir les primitives requises par une transaction, un système TM se doit d'enregistrer les adresses lues et écrites par les différentes transactions qui s'exécutent. Ces ensembles d'adresses lues et écrites permettent au système d'accomplir trois tâches critiques : la détection des conflits (CD), la gestion des versions (VM) et la résolution des conflits (CR). Chacune de ces fonctions représente une dimension majeure dans l'espace de conception d'un système HTM.

5.2.1 Détection des conflits

Un conflit se produit lorsque deux transactions ou plus accèdent à la même donnée (la définition de donnée dépend ici de la granularité du système), et que l'un au moins de ces accès est une écriture. Dans ce cas, une au moins des transactions ne pourra pas aller jusqu'à la phase de commit. La politique de détection des conflits dicte le moment où ces conflits sont détectés. On distingue classiquement deux types de détection des conflits : la détection des conflits *précoce* (de l'anglais *eager*) et la détection des conflits *paresseuse* (de l'anglais *lazy*).

La détection précoce des conflits cherche à détecter les conflits le plus tôt possible, c'est-à-dire dès qu'est faite la référence à la mémoire. Cette stratégie peut résulter en un gain de performance, puisqu'elle permet de résoudre certains conflits en gelant des processeurs, et sans utiliser systématiquement les aborts, qui résultent en un gâchis de travail. En effet, les transactions ne peuvent alors pas utiliser des valeurs qui ne sont pas à jour car déjà modifiées par une autre transaction.

À l'inverse, la détection des conflits paresseuse détecte les conflits le plus tard possible, i.e. lorsque la première de plusieurs transactions conflictuelles commite. Cela permet d'une part d'éviter de geler certains processeurs inutilement [1], et peut d'autre part garantir qu'au moins une des transactions conflictuelles commite.

De manière à pouvoir assurer la détection des conflits, les systèmes HTM requièrent en général des bits supplémentaires par ligne de cache permettant de coder le fait qu'une ligne ait été lue ou écrite durant une transaction.

Au niveau de la granularité des accès, plusieurs choix sont possibles. La solution quasiment systématiquement retenue consiste à choisir une granularité identique à celle utilisée pour le protocole de cohérence de cache, i.e. la ligne (ou bloc). Cela permet d'une part de simplifier les protocoles transactionnels, et d'autre part d'ajouter un surcout matériel raisonnable, en particulier plus raisonnable que pour la granularité du mot mémoire, même si la contrepartie est la présence de faux conflits.

5.2.2 Gestion des versions

La gestion des versions définit la manière dont sont stockées en même temps dans une transaction les valeurs intermédiaires (ou spéculées) et les valeurs précédant le début de la transaction, afin de pouvoir les restaurer dans le cas d'un abort. Une fois encore, deux approches sont classiquement utilisées : la gestion de version précoce stocke les nouvelles valeurs en place et les anciennes valeurs autre part, par exemple dans un log (espace de mémorisation réservé à cet usage). À l'inverse, la gestion de version paresseuse laisse les valeurs pré-transactionnelles en mémoire, et place les nouvelles valeurs dans un tampon.

L'avantage de la gestion de version précoce est que les commits sont plus rapides, mais la contrepartie en est que les aborts sont plus lents, ce qui peut amplifier les effets de la congestion. La gestion paresseuse résulte quant à elle en des commits plus lents, mais des aborts plus rapides.

5.2.3 Résolution des conflits

La troisième dimension de l'espace de conception d'un système TM, la résolution des conflits, détermine les actions à entreprendre lorsqu'un conflit est détecté. Ces actions

[1] Soit le cas de trois transactions T_1, T_2 et T_3 où T_1 est en conflit avec T_2 et T_2 est en conflit avec T_3, mais où T_1 et T_3 ne sont pas en conflit. Avec une CD précoce, on peut avoir que T_3 est gelée en attente de T_2 et T_2 en attente sur T_1. Avec une CD paresseuse, T_1 et T_3 s'exécutent en même temps, et si une de ces transactions commite avant T_2, les deux pourront commiter sans avoir attendu.

dépendent en particulier de la détection des conflits : une détection des conflits précoce doit résoudre les conflits dès qu'un processeur fait une requête sur une donnée qui entre en conflit avec une ou plusieurs autres transactions. La politique de résolution peut alors geler le processeur ayant fait la requête, ou aborter une ou plusieurs des transactions en conflits. Avec une détection des conflits paresseuse, la résolution des conflits se fait au moment du commit d'une transaction, et si cette dernière entre en conflit avec d'autres transactions. Les actions à prendre peuvent être de geler ou aborter le processeur qui commite, ou bien aborter tous les processeurs en conflits pour compléter la phase de commit commencée.

5.3 Classification des systèmes

Selon les points de conception définis dans la section précédente, la majorité des systèmes HTM publiés tombent dans trois grandes catégories :

- CD paresseuse/VM paresseuse/Processeur qui commite gagne (LL)
- CD précoce/VM paresseuse/Processeur qui fait la requête gagne (EL)
- CD précoce/VM précoce (EE)

5.3.1 Les systèmes LL

Les systèmes LL mettent dans un tampon les valeurs spéculées jusqu'à ce que la transaction commite. Ce tampon ne peut pas être un simple tampon, car il y a nécessité de pouvoir aussi rechercher une valeur dans le tampon et la modifier. Ainsi, toutes les transactions en cours peuvent utiliser de manière indépendantes les valeurs pré-transactionnelles d'une donnée. De façon à garantir l'atomicité du commit, ces systèmes utilisent en général un arbitre comme un jeton de commit. Un processeur ne peut commiter que s'il possède le jeton, et il y a exactement un jeton dans le système à chaque instant. Une transaction qui commite informe les autres transactions des emplacements lus et modifiés, suite à quoi les transactions recevant ces informations s'abortent si besoin. De cette manière, la transaction qui commite gagne toujours. Cette politique a deux avantages : premièrement, elle garantit l'avancement du système en assurant qu'il y a toujours un commit en cas de conflit. Deuxièmement, une transaction qui commite n'est jamais retardée par une transaction qui aborte.

5.3.2 Les systèmes EL

Les systèmes EL détectent les conflits au moment des accès à la mémoire, mais retardent la résolution au moment du commit. Lorsqu'une requête crée un conflit, la transaction ayant fait cette requête gagne le conflit (i.e. la requête aboutit), et les transactions conflictuelles doivent alors aborter. Tout comme les systèmes LL, cette catégorie de systèmes simplifie les aborts puisque les nouvelles valeurs sont mises dans un tampon et ne modifient pas les valeurs présentes avant la transaction. Ces systèmes ont été principalement conçus pour des raisons de complexité d'implantation moindre, et de compatibilité avec les protocoles de cache. Néanmoins, leurs performances sont significativement en-dessous des autres catégories [BMV+07].

5.3.3 Les systèmes EE

Les systèmes EE détectent aussi les conflits au moment des accès à la mémoire, mais cherchent à résoudre ces conflits le plus tôt possible. Les mises à jour sont faites directement dans les caches, et les valeurs précédant la transaction sont sauvegardées dans un log

en cas de modification. La classification faite ici ne spécifie pas de politique de résolution particulière pour cette catégorie, car les possibilités sont multiples : abort de la transaction créant le conflit, abort des autres transactions, gel de la transaction créant le conflit, etc. De plus, cela peut être combiné à d'autres choix transversaux comme l'ajout d'un temps d'attente (dit *backoff*) après un abort. Toutes ces stratégies favorisent des commits rapides, mais ralentissent les aborts puisque le parcours du log est alors nécessaire.

5.4 Fonctionnalités des systèmes TM

Si les catégories présentées au-dessus définissent la politique principale d'un système donné, elles ne précisent pas les fonctionnalités supportées ou non par les systèmes, ou encore certaines spécificités de conception visant la simplification ou l'optimisation du système. Nous présentons dans cette section les particularités ou fonctionnalités majeures des systèmes HTM.

5.4.1 Transactions bornées et non-bornées

Si les systèmes classiques supportent en général les transactions qui débordent du cache, voire qui accèdent un très grand nombre d'emplacements mémoire, ils ne supportent pas les transactions dites non-bornées. Cette terminologie, un peu ambigüe, vient du fait que les systèmes HTMs sont en général conçus pour les CMPs. Dans ce cadre, il est de mise que le système d'exploitation décide de changer le thread qui s'exécute sur un cœur, ou décide encore de déplacer une page contenant des données transactionnelles en mémoire (swap). Ainsi, ce terme dénote le fait qu'une transaction dans un système puisse supporter un changement de contexte ou le fait d'être déplacé sur le disque dur. Pour pouvoir supporter cela, les transactions requièrent entre autres d'être virtualisées afin que leur état puisse être sauvegardé en mémoire.

Ainsi, une distinction est faite entre les systèmes qui supportent la virtualisation des transactions et les autres.

5.4.2 Utilisation de signatures

Souvent, les systèmes TM sont obligés de faire une approximation des emplacements mémoire accédés durant une transaction quand ces dernières débordent du cache. Une technique utilisée consiste ainsi à utiliser un bit de débordement par cache et à signaler un conflit dès qu'une transaction s'exécutant sur le processeur correspondant reçoit une requête de cohérence. Cela résulte en de nombreux cas de faux conflits. Pour éviter ce problème et mieux gérer les grandes transactions, [CTTC06] a introduit l'idée d'utiliser des signatures matérielles pour représenter les ensembles d'adresses lues et écrites. Les signatures donnent une sur-approximation de ces ensembles, qui peuvent mener à des faux positifs, mais pas à des faux négatifs – c'est-à-dire qu'il est possible qu'une transaction aborte inutilement, mais pas l'inverse. Néanmoins, pour les petites transactions ne débordant pas du caches, les risques de faux positifs sont très faibles, et pour les grandes transactions, ils sont moins élevés qu'avec une autre stratégie si les signatures sont bien choisies, et leur taille suffisamment grande.

Le cout de cette solution est qu'elle nécessite l'implantation matérielle du calcul des signatures, ainsi que de plusieurs opérations pour l'insertion, la recherche d'un élément ou l'intersection de deux signatures. Une façon de procéder est d'utiliser des filtres de Bloom [Blo70] pour l'implémentation des signatures [SYHS07].

Enfin, la compatibilité de cette solution est meilleure avec une détection des conflits précoce, car la coupler avec une solution paresseuse nécessite de diffuser les signatures de la transaction qui commite à tous les processeurs.

5.4.3 Entrelacement de transactions

Un des principaux avantages des mémoires transactionnelles est qu'elles apportent un modèle de programmation modulaire. Cela n'est pas le cas avec des verrous : afin de garantir l'absence d'étreintes mortelles, les programmeurs ont souvent besoin de savoir quels sont les verrous accédés par les fonctions appelées et les fonctions appelant le module. Ce problème ne se pose pas avec les transactions, mais il en résulte que plusieurs transactions peuvent être entrelacées. Ainsi se pose le problème de la sémantique de l'entrelacement.

```
begin_transaction();
   begin_transaction();
      x = x + 1;
   end_transaction();
   ...
   begin_transaction();
      a = x + a;
      b = b + 1;
   end_transaction();
end_transaction();
```

FIG. 5.1 – Exemple d'entrelacement de plusieurs transactions

La figure 5.1 illustre ce point sur un exemple : que se passe-t-il si la deuxième transaction interne entre en conflit et doit aborter, alors que nécessairement la première transaction interne a déjà terminé sa phase de commit ? Trois sémantiques existent pour répondre à cette question : la sémantique à plat, la sémantique fermée et la sémantique ouverte [MBM+06b].

5.4.3.1 La sémantique à plat

La sémantique à plat consiste à ne considérer que la transaction la plus externe, et à ignorer les transactions internes. D'un point de vue réalisation, cette solution est de loin la plus simple, car un registre suffit pour garder le compte du niveau d'entrelacement. Ainsi, dans le cas de l'exemple donné, un abort ayant lieu dans la deuxième transaction la plus interne aura pour conséquence de faire recommencer la transaction la plus externe.

L'inconvénient de cette solution est qu'elle peut mener à un gâchis de travail. En effet, si une transaction interne courte entre en conflit et doit aborter, alors que ce conflit ne porte que sur les variables propres à cette transaction, la transaction la plus externe – possiblement longue – va aborter et recommencer alors qu'il suffirait de faire recommencer cette transaction interne.

5.4.3.2 La sémantique fermée

La sémantique fermée vise à corriger le problème posé par la sémantique à plat. Pour chaque transaction sont enregistrées les emplacements mémoire lus et écrits. Il est ainsi possible lors d'un abort d'aborter la transaction conflictuelle la plus interne : il faut pour

cela partir de la transaction ayant levé le conflit et remonter la hiérarchie des transactions en vérifiant à chaque étape si la transaction actuelle est toujours en conflit. Dans le cas de l'exemple, si la deuxième transaction interne a un conflit sur la variable `b` (et que cette variable n'est pas accédée en dehors de cette transaction particulière), seule cette transaction va aborter et recommencer.

La contrepartie de cette solution est qu'il faut répliquer tous les mécanismes de détection des conflits pour chacun des niveaux hiérarchiques, ce qui a un cout matériel élevé. En pratique, il faut fixer une profondeur maximale de transaction au-delà de laquelle on a une sémantique à plat.

5.4.3.3 La sémantique ouverte

Dans les deux sémantiques présentées, un commit interne n'est pas un vrai commit dans le sens où il peut être défait. Dans la sémantique ouverte, un commit interne est un vrai commit dans le sens où ses modifications sont visibles des autres threads. Cette solution est assez compliquée et implique entre autres de définir pour chaque transaction des opérations d'annulation à effectuer en cas d'abort. Pour plus de précision, le lecteur peut se référer à [MBM+06b]

5.4.4 Autres particularités/fonctionnalités des systèmes TM

5.4.4.1 Opérations d'entrées/sorties

Du fait de leur nature, les transactions ne sont pas très compatibles avec les opérations d'entrées/sorties et les appels systèmes. En effet, les opérations d'entrées/sorties effectuent souvent des actions irréversibles, qui ne peuvent pas être défaites dans le cas d'un abort (par exemple, la lecture ou l'écriture d'un fichier). Beaucoup de systèmes TM ne prennent pas en compte ce problème et considèrent que les opérations d'entrées/sorties ne sont pas à utiliser au sein des transactions. Même les systèmes qui apportent une réponse à ce problème n'apportent jamais une réponse entièrement satisfaisante, ce qui constitue un léger frein à l'expansion du modèle.

On pourra se référer à [BZ07] pour un survol du problème. [BLM06] et [LZL+08] abordent aussi cette question.

5.4.4.2 Interaction avec les verrous

Pour palier le problème évoqué au-dessus, on pourrait penser utiliser des verrous au milieu des transactions. Malheureusement, cela pose d'autres problèmes. Les accès aux verrous sont en général non cachés, donc court-circuitent le système TM. Cette approche ne marche pas car une prise de verrou suivie d'un abort mènerait systématiquement à une étreinte mortelle. Cependant, même en instaurant un mécanisme pour cacher les accès aux verrous, de nombreuses pathologies se mettent en place [HVS08]. L'utilisation conjointe de transactions et de verrous est donc à éviter d'une manière générale.

5.4.4.3 Interaction avec les données non-transactionnelles

Les systèmes transactionnels doivent enfin faire face au problème d'un accès non transactionnel sur une donnée qui est par ailleurs dans une transaction. Deux approches existent : l'isolation faible permet alors à cet accès d'observer l'état intermédiaire de la donnée, alors que l'isolation forte ne le permet pas. Si la plupart des systèmes STM ne fournissent qu'une

isolation faible, les systèmes HTM se doivent en général de fournir une isolation forte, car la granularité d'accès est plus grande : il se peut qu'une ligne mémoire contienne à la fois des mots transactionnels (données partagées) et des données privées à un thread. Garantir une isolation faible ne permettrait pas de traiter correctement ce cas. Il est à noter que nos travaux identifient ce problème de présence de différents types de données dans une ligne comme une source majeure de complications dans l'implémentation d'un système HTM (voir chapitre 6, section 6.2.3).

5.5 Systèmes HTM existants

Herlihy *et al.*, et leur système **HMTM** [HM93], ont les premiers proposé le concept de mémoire transactionnelle comme étant une technique matérielle fournissant des mises à jour atomiques sur un nombre borné d'emplacements en mémoire. Cette technique était basée sur le couplage entre le protocole de cohérence de cache existant et les besoins liés à l'atomicité.

Suite à cette proposition, le domaine des mémoires transactionnelles est redevenu inactif pendant une dizaine d'années. Puis, en 2004, il a connu un regain d'intérêt croissant du fait de l'arrivée progressive des machines multicœur. Depuis, une grande variété de systèmes HTM a vu le jour. Néanmoins, un seul système intégré embarquant un système HTM a aujourd'hui été réalisé : le processeur Rock de Sun [DLMN09, DLM+10].

De fait, le domaine des mémoires transactionnelles est encore peu mature. Si beaucoup de systèmes ont vu le jour, aucune solution n'a été reconnue comme étant la meilleure.

De plus, et en dépit des points de conception présentés, la classification et comparaison de ces systèmes et des solutions présentées est rendue très difficile par le fait que les auteurs n'utilisent pas toujours la même terminologie, ou alors une terminologie mal définie, et du fait que les suppositions au regard du matériel varient beaucoup d'un système à un autre. Aussi, en pratique, les systèmes présentés dans la suite abordent tous le problème avec leur propre approche et peu se comparent aux autres.

Enfin, les systèmes présentés dans la suite ne sont que les principaux. Pour une bibliographie plus complète sur le sujet, on pourra se référer à [BHHR].

5.5.1 Les systèmes précurseurs

Un des premiers systèmes proposés depuis le regain d'intérêt pour les systèmes HTM est **TCC** [HWC+04]. Ce système propose un modèle de programmation dans lequel tout le code est exécuté dans des transactions, c'est-à-dire que tous les accès vers la mémoire sont transactionnels. Les frontières entre transactions sont définies par l'utilisateur au moyen d'une primitive de synchronisation. Ce système se range dans la catégorie LL, puisque les processeurs doivent acquérir un jeton afin de pouvoir commiter. Les débordements de cache sont gérés de façon très sommaire puisque lorsqu'une transaction déborde, elle requiert le jeton avant de pouvoir continuer. Enfin, les besoins matériels de ce systèmes sont très élevés.

UTM [AAK+05] est le premier système conçu de manière à supporter des transactions non-bornées. Il s'agit d'un système EE dans lequel les transactions sont virtualisées par l'intermédiaire d'un log de transaction se trouvant en mémoire. La stratégie utilisée pour la détection des conflits est compliquée et requiert un support matériel important. De fait, ce système n'a connue aucune implémentation et est resté à l'état de concept.

LTM [AAK+05], présenté dans le même papier, est une dégradation du système précédent afin de pouvoir en obtenir une implémentation. En particulier, les transactions ne sont pas virtualisées, ce qui ne les rend pas robuste à un changement de contexte.

La politique de débordement consiste à copier les données débordées dans une table de débordement en mémoire. Tous les accès à ces données se font ensuite en parcourant cette table.

VTM [RHL05] est un système EL complexe mais complet, qui virtualise les transaction. VTM utilise une structure indépendante (XADT) branchée sur le processeur et le cache. Cette structure contient à la fois les nouvelles valeurs et les données débordées, et implémente plusieurs opérations comme l'ajout d'une entrée dans la table, la recherche d'un bloc enregistré, le commit et l'abort d'une transaction. VTM utilise des filtres de Bloom pour détecter les conflits entre les ensembles de lignes lues et écrites. Enfin, grâce à la virtualisation, VTM supporte le changement de contexte et la pagination. Néanmoins, ce système est très couteux.

5.5.2 L'expansion des systèmes TM

LogTM [MBM+06a] est un système EE qui garde trace des lignes lues et écrites par l'intermédiaire de bits supplémentaires R et W associés à chaque ligne de cache. LogTM fait le choix de résoudre les conflits en mettant en attente les processeurs plutôt que de les aborter quand cela est possible, et détecte les cycles possibles en utilisant une méthode d'estampilles distribuées. La principale contribution de LogTM est l'approche proposée pour gérer les débordements dans les transactions : ce système utilise un bit de débordement par cache qui est activé lors d'un débordement, ce qui permet de ne pas avoir de log de débordement.

Bulk [CTTC06] est un système LL qui met dans un tampon les nouvelles valeurs jusqu'au commit. Comme pour deux des systèmes précédents, les conflits sont détectés par l'intermédiaire de signatures, qui sont alors diffusées lors de la phase de commit. Bulk supporte l'entrelacement fermé de transactions, ainsi que le changement de contexte, ce qui en fait un système relativement complet.

XTM [CCM+06] est une extension de TCC qui supporte un deuxième mode pour l'exécution des transactions, entièrement en logiciel, ce qui en fait un système hybride. Une transaction qui déborde du cache génère une exception qui fait recommencer la transaction en logiciel. Dans ce deuxième mode d'exécution, les transactions sont entièrement virtualisées mais ont un surcout élevé et une détection des conflits peu précise. Il en résulte au final des performances mitigées.

PTM [CNV+06] suit la même approche consistant à virtualiser les transactions et utilise une granularité de détection des conflits de la taille d'une page lorsque les transactions excèdent les capacités matérielles.

5.5.3 Les systèmes récents

LogTM-SE [YBM+07] et **LogTM-VSE** [SVG+08] sont des variantes de LogTM qui ajoutent un support matériel - les signatures - pour enregistrer les lignes lues et écrites au lieu d'utiliser des bits R et W par ligne de cache. LogTM-VSE tire parti de cette représentation sous forme de signatures pour virtualiser les transactions et supporter ainsi les changements de contexte.

OneTM [BDLM07] est un système HTM **EE** similaire à LogTM pour le log. Le point central de OneTM est l'utilisation d'un cache de permission conservant les requêtes de cohérence, pour pouvoir étendre la borne à laquelle les transactions débordent. OneTM a deux variantes : la première permet un seul débordement à la fois et gèle toutes les autres transactions, tandis que la seconde permet un seul débordement à la fois mais laisse les autres transactions s'exécuter tant qu'elles ne débordent pas.

RTM [SSH+07] est un système hybride qui implémente au niveau matériel les différentes catégories de systèmes HTM, tout en laissant le choix de la politique au niveau logiciel. L'avantage de cette approche, couteuse au niveau matérielle, est qu'elle permet facilement de définir des politiques de résolution des conflits avancées, et de basculer entre différents schémas de détection. RTM se base sur un protocole de cohérence à écriture différée, dans lequel cinq états sont ajoutés aux quatre traditionnels pour une ligne de cache. En raison de la gestion logicielle des débordements, les performances se dégradent rapidement lorsque les transactions débordent des structures matérielles.

FlexTM [SDS08] est une amélioration de RTM dans laquelle est ajouté un dispositif matériel pour la gestion de signatures. La détection des conflits est aussi découplée de la gestion des versions, ce qui n'était pas le cas dans RTM. Enfin, le protocole transactionnel est simplifié puisqu'il ne comporte plus que deux états supplémentaires.

TokenTM [BGH+08] est un système EE qui utilise l'abstraction des jetons pour garantir la cohérence des données et détecter les conflits. L'avantage de cette approche est qu'elle permet d'éviter les faux positifs comme dans le cas des signatures. Le cas de partage d'une ligne en lecture à la fois en dehors et à l'intérieur d'une transaction, généralement immédiat dans les systèmes EE, est ici complexe mais possible par l'intermédiaire d'un mécanisme compliqué de fission et de fusion des états transactionnels. TokenTM supporte les changements de contexte grâce au niveau d'abstraction des jetons, mais le cout matériel de ce système est relativement élevé.

FASTM [LMG09] est un système EE similaire à LogTM-SE, avec pour différence une modification du protocole de cohérence permettant d'avoir des aborts rapides – i.e. sans parcours de log – dans le cas où la transaction qui aborte n'a pas dépassé la capacité du cache L1. Pour ce faire, les valeurs modifiées pré-transactionnelles sont propagées vers les niveaux inférieurs de la hiérarchie mémoire de manière à garder le plus possible les valeurs spéculées dans le cache L1.

5.6 Environnement de simulation et méthodes de validation des systèmes existants

La majorité des systèmes présentés utilisent l'environnement de simulation Simics GEMS [MSB+05], qui permet de modéliser et de simuler des systèmes multiprocesseurs. L'approche consiste à découpler la simulation du temps et celle des fonctionnalités. De plus, certains composants matériels ont des modèles temporels qui ne sont qu'approximatifs. En ce sens, le niveau d'abstraction utilisé est différent de celui que nous visons dans notre étude, et les méthodes de validation utilisées ne s'appliquent pas à notre contexte.

Les autres systèmes utilisent des modèles de précisions du même ordre, en général par l'intermédiaire de simulateurs pilotés par évènements. Ces modèles sont dans tous les cas moins précis que les simulations *cycle-accurate*.

5.7 Conclusion

Si les systèmes récents ont porté l'accent sur la virtualisation des transactions, nécessaire pour l'adoption du modèle dans le monde réel, peu de travaux ont étudié des propriétés de plus bas niveau. En particulier, la supposition d'un protocole de cohérence à écriture différée est toujours faite. Par ailleurs, peu de travaux se sont intéressés à l'implantation d'un système TM sur un MPSoC. [FMB+07] aborde la question, mais avec des hypothèses idéalistes (ajout d'un second cache propre aux transactions, les transactions ne débordent

jamais de ce cache) et des expérimentations faibles. [GAG08] analyse un protocole avec détection des conflits au niveau du répertoire, mais demande un cout matériel très élevé : associer un numéro de série à chaque transaction, et pouvoir enregistrer un numéro par processeur et par ligne de cache.

Nous nous proposons premièrement d'aborder ce problème en concevant, implémentant et comparant d'un point de vue performances deux implémentations d'un système TM visant les MPSoCs : l'une basée sur un protocole à écriture simultanée, et l'autre basée sur un protocole à écriture différée. Dans ce but, les hypothèses que nous ferons par rapport aux points énoncés dans cette section seront les suivantes :

- Sémantique d'entrelacement à plat
- Support pour les grandes transactions (i.e. qui débordent du cache) mais pas de virtualisation
- Pas d'opérations d'E/S dans les transactions
- Atomicité forte vis-à-vis des accès non-transactionnels

Dans un second temps, nous nous intéresserons à des variantes du système basé sur le protocole à écriture différée, notamment au niveau de la politique de résolution des conflits, et nous les comparerons selon deux types de critères : performances et robustesse, au sens de la résistance aux pathologies et aux garanties apportées vis-à-vis de la terminaison.

Chapitre 6

Étude du protocole de cohérence pour les mémoires transactionnelles

DANS ce chapitre, nous allons aborder la question de l'implantation d'un système TM matériel avec les contraintes définies précédemment, en vue d'étudier la viabilité du modèle de programmation transactionnel dans le domaine de l'embarqué. Cela sera fait par l'intermédiaire de la comparaison de deux systèmes HTM : un système original basé sur un protocole à écriture simultanée et un plus traditionnel basé sur un protocole à écriture différée.

6.1 Introduction

Le modèle de programmation apporté par les mémoires transactionnelles permet de faciliter l'écriture de programmes parallèles corrects. Cela implique évidemment que le système sous-jacent doit faire le traitement nécessaire pour garantir les propriétés des transactions. Aussi, de tels systèmes sont complexes, en particulier lorsqu'ils sont implantés en matériel. Or, cela doit être le cas, au moins en partie, pour que les programmes reposant sur ces systèmes soient performants.

Un système TM matériel est toujours intimement lié au protocole de cohérence de cache, puisque ce dernier va partiellement définir les états des lignes de la mémoire. Comme le domaine de l'embarqué remet en cause la suprématie du protocole à écriture différée, en particulier avec l'arrivée des NoCs, il est légitime de se demander si un système HTM peut être implanté au-dessus d'un tel protocole, auquel cas se pose le problème de savoir comment les deux implantations se comparent l'une à l'autre, en termes de performances et de cout.

Cette partie fait trois contributions :

- La conception d'un système TM matériel basé sur des caches à écriture simultanée dans lequel l'information relative aux données accédées dans la transaction est enregistrée au niveau du répertoire (i.e. la détection de conflits est faite au niveau du répertoire). À notre connaissance, aucun système basé sur ces idées n'a été précédemment proposé.
- Une comparaison au niveau cycle entre une implémentation de ce système et une implémentation d'un système TM matériel plus classique basé sur une cohérence à écriture différée - en dehors du protocole de cohérence de cache, les architectures sont identiques.
- Enfin, l'étude de l'influence de la répartition mémoire pour le système à écriture

différée, qui montre que la distribution physique des bancs mémoire peut mener à des gains importants pour les applications ayant une congestion élevée.

Dans la suite de ce chapitre, nous nous référerons au système TM à écriture simultanée par LightTM-WT, tandis que le système TM à écriture différée sera appelé LightTM-WB, le nom LigthTM sera quant à lui utilisé pour référencer les deux systèmes. Ce nom reflète les choix de conception et hypothèses faites dans le but de viser le domaine de l'embarqué.

6.2 Protocole de cohérence de cache sans mémoire transactionnelle

Cette section donne une vue d'ensemble du mécanisme de cohérence de cache utilisé comme base pour les systèmes TM présentés dans la suite. Elle ne présente cependant pas tous les détails étant donné que cela serait très technique et ne constitue pas une nouveauté de ce travail, mais nous considérons important de décrire la manière dont l'architecture de cohérence de cache est construite afin de comprendre les résultats présentés dans la suite. En particulier, des différences notables apparaissent comparé à un protocole à écriture différée basé sur l'espionnage (ou *snoop*) d'un bus.

Nous avons choisi de présenter trois points clés du protocole de cohérence de cache : la machine d'états finie (FSM), le mécanisme d'invalidation, et un exemple de scénario résultant de l'utilisation d'un répertoire. Ces trois points sont présentés pour le protocole à écriture différée, mais en dehors de la FSM, les principes sont aussi valides pour l'écriture simultanée.

6.2.1 Machine d'états finie

La FSM du protocole pour une ligne de cache est représentée figure 6.1. Elle consiste en 4 états (**M**,**E**,**S**,**I** - **M**odified, **E**xclusive, **S**hared, **I**nvalid) et n'est pas différente de celle d'un protocole d'espionnage, excepté que le changement d'état vers l'état **I** est fait sur réception d'une invalidation. Bien sûr, cette FSM n'est que l'automate de haut-niveau et ne représente pas les requêtes d'allocation ni les états intermédiaires du protocole d'accès aux données.

6.2.2 Mécanisme d'invalidation

Nous présentons maintenant brièvement comment fonctionne le mécanisme d'invalidation. Un exemple est montré figure 6.2.

Quand le contrôleur reçoit une requête d'échec de lecture ou d'écriture, il envoie une requête d'invalidation à tous les possesseurs de la copie. Une fois que les réponses ont été reçues par le contrôleur, la mémoire est mise à jour (états, possesseurs et éventuellement données) et une réponse à la requête initiale est envoyée. Le contrôleur mémoire ne prend pas en compte des requêtes d'autres caches à partir du moment où la requête initiale est reçue, et jusqu'à ce que la réponse ait été envoyée, afin de garantir la cohérence mémoire.

6.2.3 Exemple de scénario : invalidations tardives

Cette partie détaille un exemple de complication qui peut arriver avec l'utilisation d'un répertoire à la place d'un espionnage de bus. Cet exemple est décrit figure 6.3.

Bien que cette section n'aille pas plus loin dans les implications de l'implémentation d'un système TM matériel au-dessus d'un NoC, nous supposons que ces dernières sont du même type que celles se produisant pour la cohérence de cache. Le protocole est globalement plus

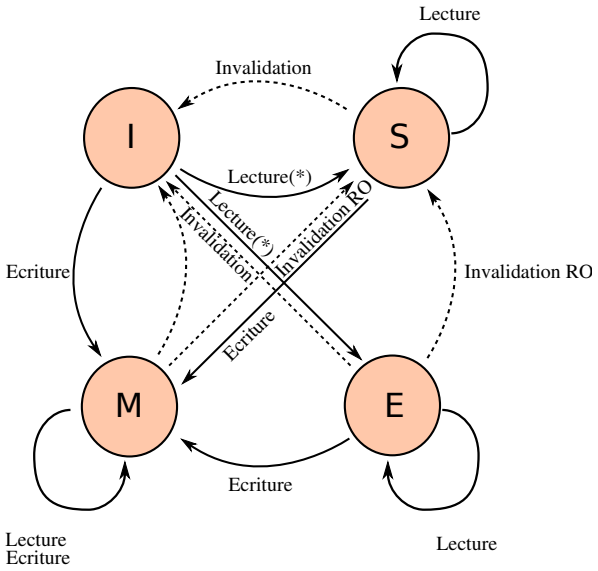


FIG. 6.1 – **FSM d'une ligne de cache pour le protocole à écriture différée sans les transactions** Les transitions marquées d'une astérisque sont prises selon la réponse de la mémoire

complexe puisqu'il y a plus de types de requêtes, et des précautions doivent être prises pour éviter les étreintes mortelles et gérer les requêtes périmées.

Une partie de la complexité est aussi induite par le fait que le système doive supporter des lignes contenant à la fois des données transactionnelles et non-transactionnelles. En effet, si pour un protocole de cohérence de cache il est possible de raisonner au niveau d'une ligne mémoire complète, cela n'est pas le cas pour un protocole transactionnel : une ligne peut être accédée au sein et en dehors d'une transaction en même temps, en particulier si elle contient à la fois des variables locales et des variables partagées.

Il n'est en effet pas possible de déterminer à la compilation les variables transactionnelles des autres en vue de les regrouper, sans changer le modèle de programmation. Quels que soient les efforts d'alignement à la compilation, un programme peut contenir des variables globales partagées transactionnelles, ainsi que des variables déclarées globales, mais utilisées implicitement comme des variables locales par un des threads (et donc, en dehors de toute transaction).

Le protocole de cohérence pour les lignes de cache doit donc gérer la combinaison des deux protocoles en parallèle.

6.3 LightTM-WT

Cette section introduit LightTM-WT, un système TM original basé sur un schéma CD précoce/VM précoce/Gel du processeur ayant fait la requête. Comme la plupart des systèmes existants, LightTM-WT utilise des bits R et W associés à chaque ligne de cache pour se souvenir des lignes lues et écrites à l'intérieur d'une transaction. Ces deux bits par ligne s'ajoutent au bit déjà requis pour encoder l'état de la ligne (valide ou invalide).

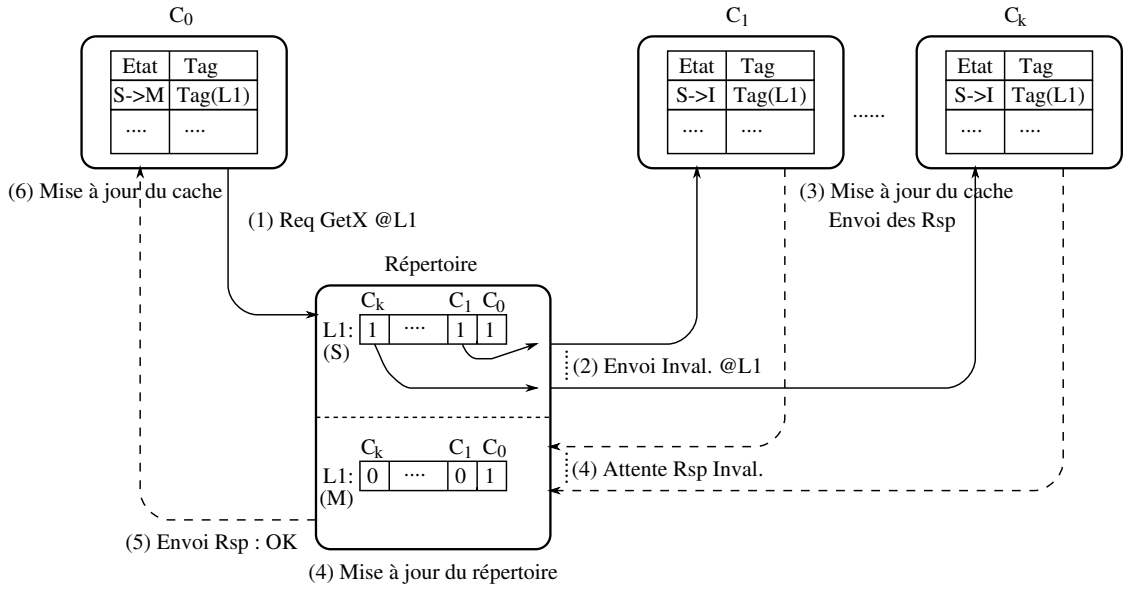


FIG. 6.2 – **Mécanisme d'invalidation pour le protocole de cohérence de cache**
Cet exemple montre comment marche la cohérence sur un cas simple. Dans l'état initial, les caches C_0, C_1 et C_k ont une copie valide de la ligne L1 dans l'état S.
(1) Le processeur 0 fait un défaut d'écriture (requête d'écriture sur une ligne de cache dans l'état **S**), ce qui fait émettre à C_0 une requête GetX.
(2) La requête est reçue par le contrôleur mémoire, qui envoie des invalidations vers les caches ayant une copie de la ligne (sauf C_0).
(3) À la réception de la requête d'invalidation, les caches C_1 et C_k se mettent à jour en changeant l'état de la ligne en invalide. Ils envoient alors une réponse à la requête d'invalidation.
(4) Une fois que le contrôleur mémoire a reçu toutes les réponses aux requêtes d'invalidation, il met à jour le répertoire.
(5) Il envoie ensuite une réponse à la requête initiale de C_0.
(6) C_0 reçoit la réponse et se met à jour en changeant l'état de la ligne en **M**.

6.3.1 Détection de conflits

LightTM-WT réalise une détection de conflits précoce en envoyant d'abord une requête transactionnelle au contrôleur mémoire pour le premier accès de chaque ligne au sein d'une transaction. Ce dernier vérifie alors si un autre processeur a déjà accédé cette ligne à l'intérieur d'une transaction, et soit répond à la requête si aucun autre processeur n'a accédé cette ligne, soit gèle le processeur en n'envoyant pas de réponse si un processeur a déjà accédé la ligne. Cela requiert d'augmenter chaque ligne mémoire de $\lceil \log_2(n+1) \rceil$ *bits transactionnels* pour pouvoir coder l'identifiant (*id*) de tous les processeurs plus une valeur par défaut.

Dans le cas où un processeur est gelé, son *id* est enregistré par le contrôleur mémoire comme étant en attente sur cette ligne. Puisque chaque processeur peut-être en attente sur un nombre fini de requêtes - précisément, le nombre de mots contenus dans une ligne pour notre protocole - le cout requis pour cela est bien borné.

Aussi, si le processeur est mis en attente, le contrôleur mémoire doit vérifier qu'aucun cycle de requêtes n'est créé entre deux processeurs ou plus de manière à éviter les étreintes

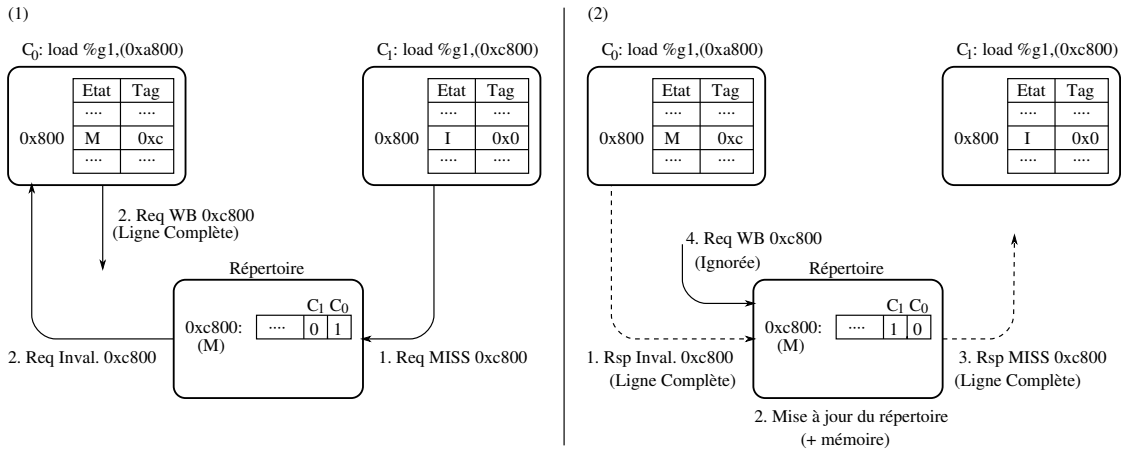


FIG. 6.3 – **Exemple d'invalidation tardive due au retard causé par le NoC**
Dans l'état initial, C_0 possède une copie valide de la ligne 0xc800 dans l'état **M**. On suppose pour des raisons de simplicité que les adresses sont sur 2 octets et le tag sur 4 bits.
(1) 1. C_1 émet une requête de miss qui est reçue par le contrôleur mémoire. **2.** Le contrôleur mémoire envoie une requête d'invalidation au possesseur de la ligne, C_0. Environ au même moment, C_0 veut lire l'adresse 0xa800, qui est placée au même endroit en cache que la ligne modifiée 0xc800. Le cache envoie donc une requête de ré-écriture à la mémoire pour recopier cette ligne en mémoire.
(2) 1. La requête d'invalidation est reçue par C_0. Comme les invalidations sont prioritaires sur les autres requêtes, le cache doit détecter que la ligne invalidée est celle qui est en train d'être recopiée en mémoire, et doit répondre avec une copie complète de la ligne. **2.** La mémoire reçoit la réponse et met à jour la ligne avec les nouvelles valeurs, ainsi que son état. **3.** La mémoire répond à la requête de C_1 avec une copie de la ligne. **4.** La requête de C_0 est prise en compte par la mémoire, mais doit être ignorée puisque les valeurs ne sont maintenant plus à jour.

mortelles. La détection de cycles est faite en gardant dans un registre du contrôleur mémoire le processeur possédant la ligne visée par la requête, et en vérifiant si ce processeur lui-même est en attente, etc., jusqu'à ce qu'un processeur ne soit pas en attente ou qu'un processeur soit en attente sur une ligne possédée par le processeur ayant fait la dernière requête.

Ce mécanisme de détection des requêtes au niveau du répertoire a néanmoins un inconvénient majeur : la détection des requêtes doit se faire de manière centralisée. Cela signifie que LightTM-WT n'est pas compatible avec une topologie mémoire mémoire distribuée sans mettre en œuvre un mécanisme de centralisation des requêtes transactionnelles en attente, afin de pouvoir détecter les cycles.

Enfin, un processeur ayant acquis un accès transactionnel sur une nouvelle ligne ne la possède probablement pas en cache, c'est pourquoi nous avons implémenté une optimisation consistant à envoyer la ligne mémoire en même temps que la réponse à la requête transactionnelle dans le cas où la donnée n'est pas en cache.

La figure 6.4 illustre la détection de conflits sur un exemple simple avec 2 alternatives : gel sans création de cycle, et cycle détecté et abort.

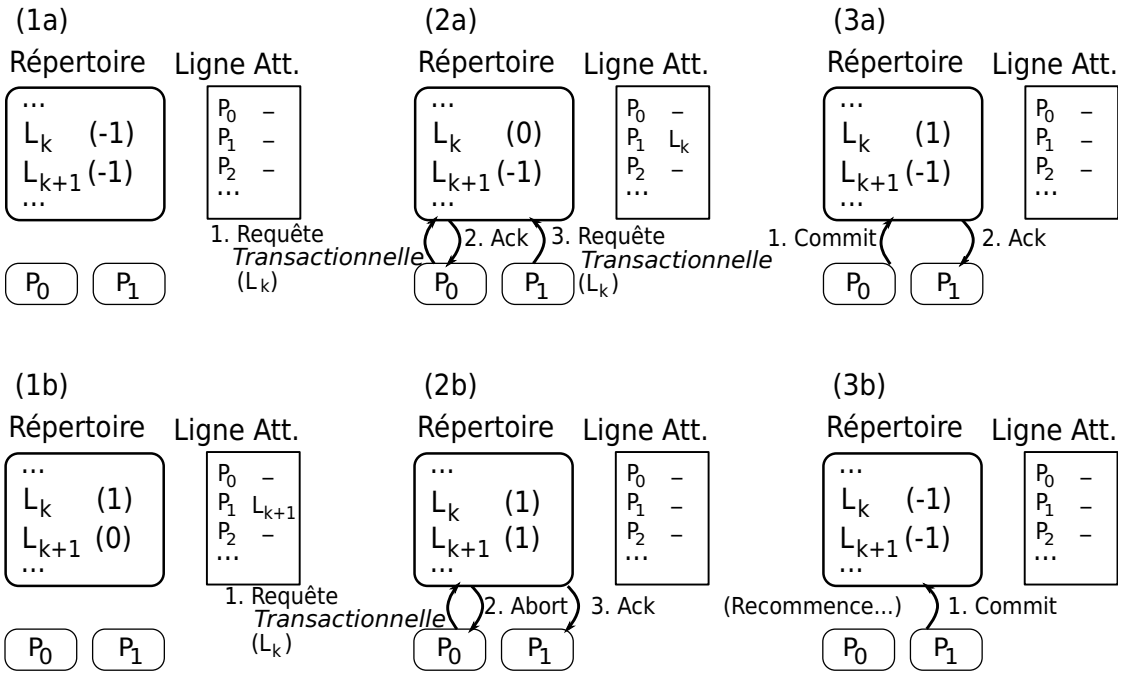


FIG. 6.4 – **Illustration de la détection de conflits sur deux cas dans LightTM-WT**
Cet exemple illustre la détection de conflits dans LightTM-WT. Dans le premier cas (1a, 2a, 3a), un processeur essaie d'accéder à une ligne à l'intérieur d'une transaction tandis qu'un autre processeur est déjà en train de la modifier, mais aucune dépendance n'existe. Dans le second cas (1b, 2b, 3b), une requête transactionnelle envoyée par un processeur crée un cycle, de quoi résulte un abort. Le répertoire comme le tableau des lignes en attente (`Ligne Att.`) font partie du contrôleur mémoire.
(1a) Aucune ligne ne fait partie d'une transaction
(2a) Les processeurs P_0 et P_1 commencent une transaction et essaient d'accéder à la ligne L_k. Comme ces requêtes sont sérialisées par la mémoire, la première requête reçue (celle émise par le processeur P_0) est traitée avant de recevoir la seconde. Le contrôleur mémoire répond à la requête du processeur P_0, puis traite la requête de P_1. Comme la ligne est déjà en mode transactionnel, aucune réponse n'est envoyée à P_1 et ce dernier est de fait mis en attente pour la ligne L_k.
(3a) P_0 commite sa transaction, relâchant ainsi L_k. La réponse peut être envoyée à P_1.
(1b) Les processeurs P_0 et P_1 exécutent une transaction. P_0 a obtenu un accès transactionnel sur L_{k+1} et P_1 sur L_k. De plus, P_1 a tenté d'accéder à la ligne L_{k+1} et a été mis en attente.
(2b) P_0 effectue une requête transactionnelle sur L_k, créant ainsi un cycle. Un abort est envoyé à P_0, qui relâche alors sa ligne, et une réponse est envoyée à P_1.
(3b) P_1 commite sa transaction, relâchant L_k et L_{k+1}.

6.3.2 Gestion de version

Bien que nous présentions LightTM-WT comme ayant une gestion de version précoce, cette terminologie ne correspond pas exactement à la définition que nous en avons donné chapitre 5 section 5.2.2. Les écritures ne sont en effet pas propagées en mémoire, suivant la définition de l'écriture simultanée, mais sont à la place enregistrées dans le cache. Il ne s'agit pas non plus d'une gestion de version paresseuse puisque cela impliquerait que les nouvelles valeurs spéculées soient mises dans un tampon, ce qui n'est pas le cas. En fait,

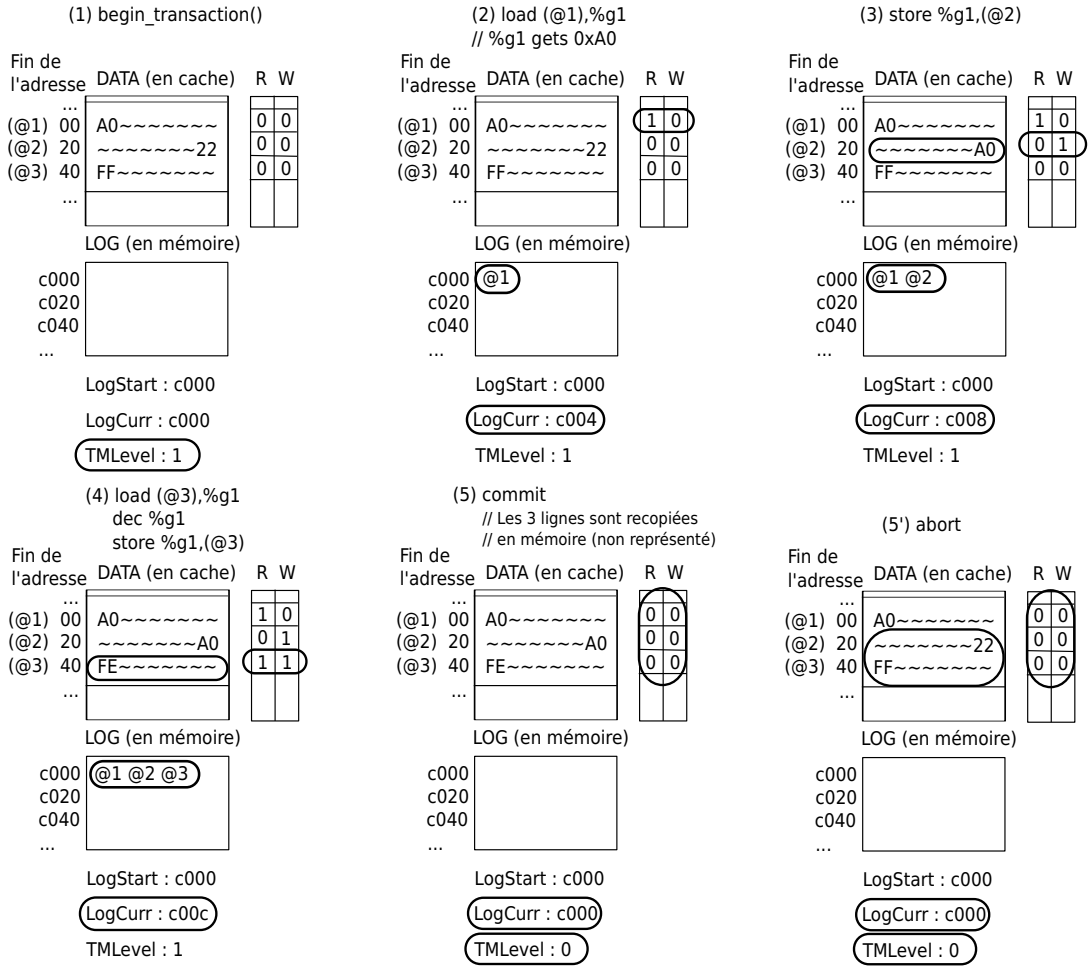


FIG. 6.5 – **Vue logique du cache et du log au cours d'une transaction dans LightTM-WT**
Cet exemple représente une vue partielle schématique de la gestion de version dans un cas d'exécution simple d'une transaction avec un processeur. La mémoire principale n'est pas représentée pour des raisons pratiques, ses valeurs étant mises à jour au moment du commit seulement. **LogStart** est un registre contenant l'adresse de base du log, **LogCurr** contient l'adresse courante du log, et **TMLevel** le niveau d'imbrication des transactions. On suppose que les lignes font 8 mots (de 4 octets) de long, et que la taille du cache est 4Ko. Le symbole ∼ dénote un mot mémoire non significatif. Les endroits entourés indiquent qu'une modification s'est produite depuis la dernière étape.
(1) Le processeur exécute l'instruction `begin_transaction();`, incrémentant seulement son registre **TMLevel**.
(2) Le processeur exécute une lecture à l'intérieur d'une transaction, ajoutant au log l'adresse de base de la ligne (@1) et positionnant à 1 le bit R.
(3) Le processeur exécute une écriture, ajoutant au log l'adresse de base de la ligne, et positionnant à 1 le bit W. L'écriture n'est pas propagée en mémoire.
(4) Le processeur exécute une *lecture-modification-écriture* sur un mot d'une ligne ayant pour adresse de base @3. Cette adresse est ajoutée au log lors de la lecture ; les bits R et W sont consécutivement mis à 1.
(5) Le processeur exécute l'instruction `end_transaction();`, commitant ainsi sa transaction. Le log est parcouru à l'envers pour que le cache puisse informer la mémoire des lignes commitées. Les bits R et W correspondants sont remis à 0 pour chaque requête de commit.
(5') (Alternative à 5) Le processeur reçoit un signal d'abort en réponse à une autre requête transactionnelle. Le log est parcouru à l'envers, et pour chaque adresse dans le log, la ligne est recopiée de la mémoire vers le cache. Les bits R et W sont aussi remis à 0.

tout se passe comme si les écritures à l'intérieur d'une transaction étaient propagées dans la mémoire principale, mais comme il est nécessaire lors d'un commit d'informer la mémoire principale des lignes qui sont relâchées, nous avons décidé de retarder l'écriture de ces nouvelles valeurs au moment du commit pour éviter de propager les valeurs spéculées. De cette manière, la mémoire n'est mise à jour qu'au moment du commit, suivant plus l'idée d'une gestion de version paresseuse. Ce choix implique aussi qu'utiliser un log pour les anciennes valeurs devient inutile puisque les anciennes valeurs sont en mémoire principale pendant la transactions. Dans notre implémentation, un log sera néanmoins utilisé pour garder trace des lignes mémoires accédées durant la transaction (i.e. ce log ne contiendra que des adresses).

La figure 6.5 illustre l'évolution du cache de données et du log au cours de l'exécution d'une transaction.

6.3.3 Résolution des conflits

La résolution des conflits a lieu quand un cycle de requêtes transactionnelles en attente est détecté. Si un processeur du cycle a fait un débordement de cache au cours de la transaction, tous les processeurs, excepté celui-là, reçoivent un abort. Les processeurs recevant le signal d'abort relâchent leurs lignes accédées durant la transaction comme lors d'un commit et recommencent leurs transactions. Si aucun processeur n'a fait de débordement du cache, la décision est prise sur la base d'un index. De plus, pour éviter quelques cas pathologiques pouvant mener à une étreinte active matérielle, un mécanisme particulier est mis en place pour s'assurer que le processeur n'ayant pas aborté suite au cycle commitera.

6.3.4 Gestion des débordements

Le terme "débordement" dans le contexte des mémoires transactionnelles réfère au remplacement d'une ligne de cache à l'intérieur d'une transaction, quand la ligne remplacée a déjà été accédée dans cette transaction. La politique de débordement de cache dans une transaction définit donc quoi faire quand deux (ou plus) lignes mémoire différentes d'une transaction sont placées au même endroit en cache. Avec les contraintes que nous avons déjà définies, deux options sont possibles : utiliser un log spécial pour les transactions qui ont débordé, ou recopier en mémoire les données qui débordent. Nous avons choisi cette deuxième solution pour des raisons de complexité et de cout, le prix à payer étant qu'au plus une transaction peut être en débordement à un instant donné. En effet, une fois qu'une transaction a débordé, les anciennes valeurs sont perdues, et la transaction ne peut alors plus aborter. Dans le cas d'un cycle, une transaction qui a débordé sera toujours celle qui ne sera pas abortée.

La figure 6.6 montre un exemple de transaction faisant un débordement de cache.

Comparé à un protocole de cohérence de cache à écriture différée, le protocole proposé a l'avantage que le contrôleur de cache n'a pas besoin d'envoyer des requêtes de cohérence vers d'autre processeurs à la réception d'une requête transactionnelle puisque la valeur la plus à jour se trouve déjà en mémoire. Cependant, un inconvénient est que les commits sont plus lents puisqu'il doit y avoir une requête (ou un paquet de requêtes) envoyé à la mémoire pour chaque ligne accédée durant la transaction. De plus, l'inconvénient principal est que la concurrence de plusieurs lectures entre deux transactions n'est pas possible avec ce protocole, alors qu'elle est naturelle avec un protocole basé sur l'écriture différée. Ainsi, les transactions accédant un grand nombre de données en lecture seule risquent d'être fortement pénalisées par ce protocole.

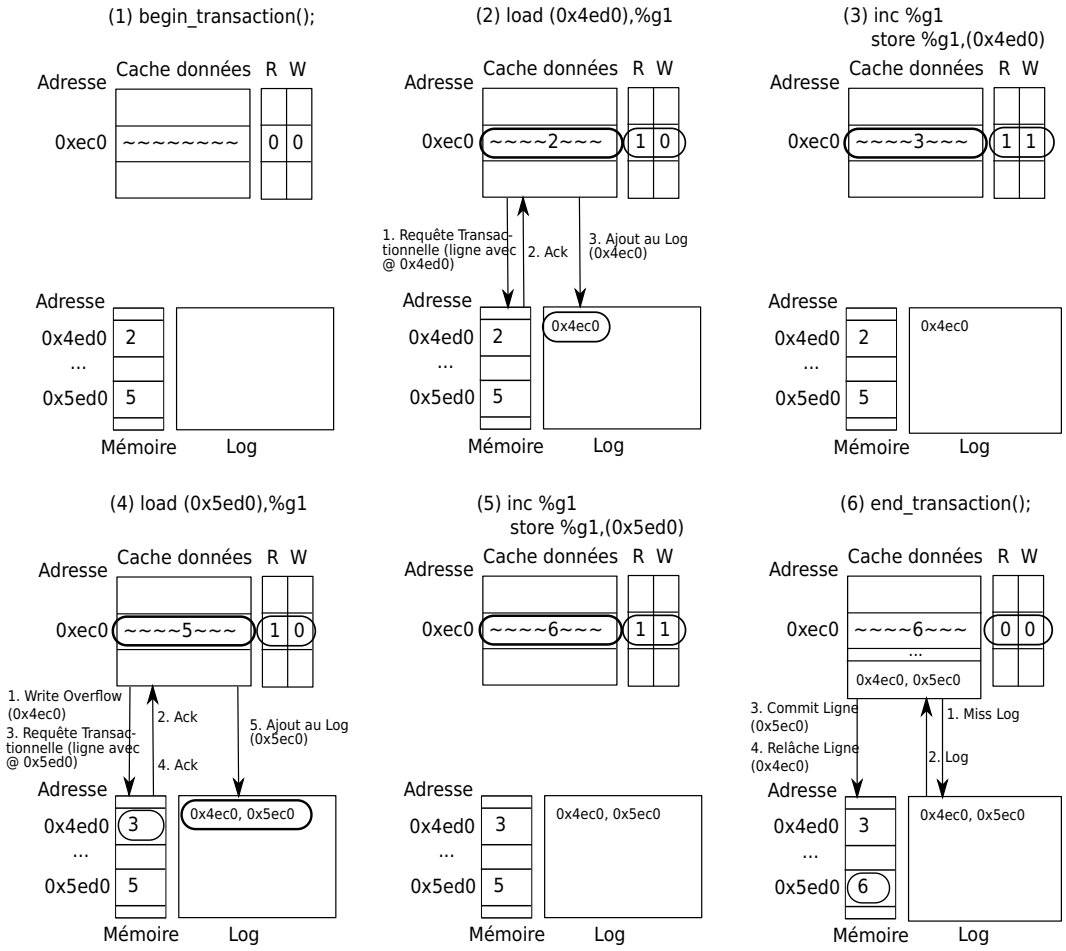


FIG. 6.6 – **Exemple d'exécution d'une transaction avec un débordement du cache dans LightTM-WT**

Cet exemple illustre une transaction accédant à deux variables situées dans des lignes mémoires différentes mais étant placées dans la même ligne de cache. Les hypothèses sur la taille et le nombre de lignes de cache sont les mêmes que précédemment.

(1) Le processeur exécute l'instruction `begin_transaction();`, aucune action particulière n'est effectuée.

(2) Le processeur exécute une lecture à l'adresse 0x4ed0. Le cache effectue d'abord une requête transactionnelle combinée à une requête de miss vers le contrôleur mémoire. Ce dernier répond à la requête, le bit R du cache est mis à 1, et l'adresse de base de la ligne est ajoutée au log.

(3) Le processeur modifie la valeur chargée et l'enregistre. L'écriture n'est pas propagée vers la mémoire et le bit W est mis à 1.

(4) Le processeur exécute une lecture à une adresse provoquant un débordement de cache à l'intérieur de la transaction (détecté par la valeur 1 du bit R). Le cache envoie d'abord une requête d'écriture pour recopier en mémoire la ligne courante du cache qui a été modifiée, et remet à 0 les bits R et W qui y sont associés. Comme aucun autre processeur n'a fait de débordement de cache, le contrôleur mémoire répond à la requête (dans le cas contraire, il aurait mis le processeur en attente). Le processeur effectue ensuite une requête transactionnelle (et un miss) pour la ligne ayant l'adresse de base 0x5ec0. Le contrôleur mémoire répond à cette requête de manière positive, après quoi le bit R du cache est mis à 1, et l'adresse de base de la ligne est ajoutée au log.

(5) Le processeur exécute une modification suivie d'une écriture sur la même donnée, ce qui positionne à 1 le bit W. L'écriture n'est pas propagée.

(6) Le processeur commite sa transaction. Il récupère d'abord le log en cache si nécessaire (et si cela n'efface pas une ligne à commiter), après quoi il envoie à la mémoire une requête de commit pour la ligne commençant à l'adresse 0x5ec0, et une requête pour relâcher la ligne commençant à l'adresse 0x4ec0 (aucune donnée n'est transmise). Les bits R et W sont remis à 0.

6.4 LightTM-WB : un système basé sur des caches à écriture différée

La version de LightTM avec des caches à écriture différée suit un schéma de CD précoce/VM précoce, la politique de résolution des conflits dépendant du type de requête et de l'état de la ligne.

6.4.1 Détection des conflits

LightTM-WB effectue une détection de conflits précoce naturellement par l'intermédiaire du protocole de cohérence. Quand un processeur requiert une ligne en vue d'une lecture, le contrôleur fait suivre cette requête aux caches ayant potentiellement une copie de la ligne dans l'état **E** ou **M**. La différence avec le protocole standard est que sans support pour les transactions, cette requête résultera toujours en un succès, la ligne modifiée étant alors systématiquement ré-écrite en mémoire. Dans le cas transactionnel, la requête peut se solder par une réponse négative dans le cas d'un conflit. Une approche similaire est utilisée dans le cas où un processeur requiert une ligne pour une écriture : le contrôleur fait suivre la requête à tous les processeurs ayant une copie, même dans l'état **S**. Quand un conflit est détecté, un Nack (Negative ACKnowledgement, ou réponse négative, RspN) n'est pas nécessairement envoyé : le processeur peut décider de s'aborter localement et de répondre positivement à la requête. Ce choix est fait selon le type de requête et l'état local de la ligne.

Quand un conflit est détecté et que le processeur ayant fait la requête doit finalement aborter, le processeur recevant l'invalidation qui cause le conflit répond normalement à la requête d'invalidation, en dehors du fait que la réponse contient l'information Nack. Au niveau de la mémoire, si parmi toutes les requêtes envoyées par le contrôleur et faisant suite à la requête initiale, une reçoit une réponse négative, le Nack est alors propagé au processeur ayant fait la requête initiale ; dans le cas contraire, une réponse standard est envoyée. Cette approche est différente du protocole utilisé dans LogTM dans lequel le processeur recevant l'invalidation envoie directement la réponse négative au processeur ayant fait la requête initiale.

La figure 6.7 illustre la détection de conflits sur un exemple simple avec 3 fins alternatives : un abort local, une réponse négative menant à un abort pour le processeur ayant fait la requête initiale, et une réponse positive sans abort dans le cas de lectures concurrentes.

6.4.2 Gestion de versions

LightTM-WB suit le même schéma que LogTM pour la gestion des versions : un log est alloué en mémoire, et avant chaque écriture à l'intérieur d'une transaction, une copie de la ligne correspondante est écrite dans le log, contrairement à LightTM-WT qui ne met dans le log que les adresses. Ainsi, en cas d'écriture à l'intérieur d'une transaction, le cache contient les nouvelles valeurs, les anciennes valeurs sont sauvegardées dans le log, tandis que la mémoire contient des valeurs qui ne sont plus à jour. Il est à noter que le fait de mettre dans le log les valeurs avant modification par la transaction est indépendant de l'état de la ligne au début de la transaction : si la ligne est dans l'état **S** ou **I**, une requête est envoyée à la mémoire pour obtenir l'état **M**, et après cela la ligne est ajoutée au log dans tous les cas (même si elle était déjà en **M**). L'implémentation actuelle contient une petite limitation : le log doit être dans une partie de la mémoire qui ne contient pas de données partagées, i.e. un bloc mémoire ne peut pas contenir à la fois des données partagées et des données de log.

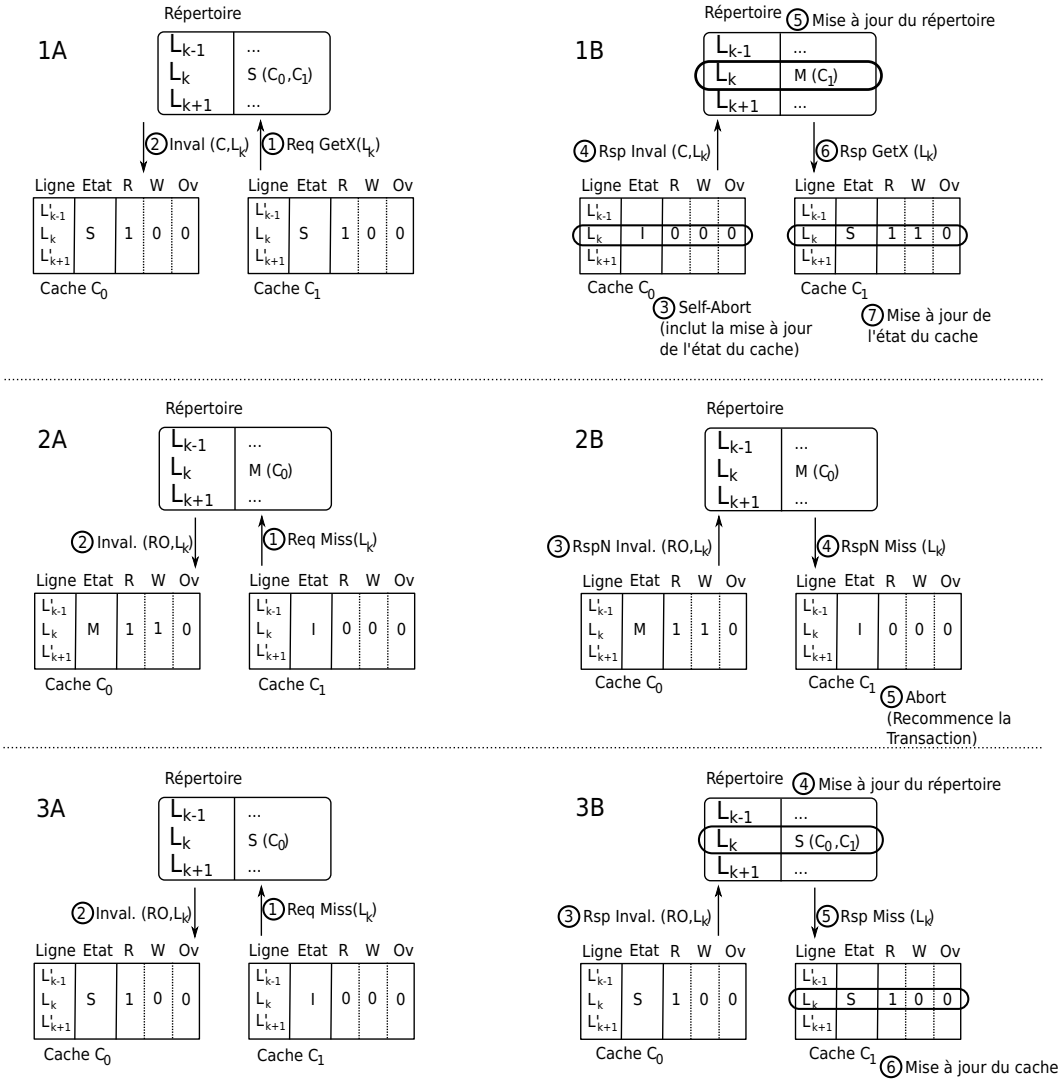


FIG. 6.7 – **Exemple de détection de conflits dans LightTM-WB**

Cet exemple montre comment marche la détection de conflits dans LightTM-WB sur trois cas simples. Pour les trois cas, on suppose que les processeurs associés aux caches sont à l'intérieur d'une transaction.

(1A) La ligne L_k est dans l'état **S** et les caches C_0 et C_1 en possèdent une copie valide. (1) Le cache C_1 souhaite écrire un mot de la ligne et émet une requête GetX vers la mémoire. (2) Avant de répondre à la requête, le contrôleur mémoire envoie une requête d'invalidation *complète* (notée Inval(C,...)) vers le cache C_0.

(1B) (3) Comme la ligne L_k a seulement été lue dans la transactions, la décision est prise de s'aborter. (4) Une réponse positive est donc envoyée au contrôleur mémoire. (5) Le contrôleur mémoire met à jour l'état de la ligne. (6) Une réponse positive est envoyée de la mémoire au cache C_1. (7) Le cache C_1 peut mettre à jour l'état transactionnel de la ligne et continuer sa transaction.

(2A) La ligne L_k est possédée dans l'état **M** par le cache C_0, et elle a été lue et modifiée à l'intérieur de la transaction courante. (1) Le cache C_1 envoie une requête de miss sur L_k à destination du contrôleur mémoire. (2) Une requête d'invalidation *lecture seule* (read-only, notée Inval(RO,...)) est envoyée à C_0.

(2B) (3) Comme la ligne a été modifiée dans la transaction courante, le cache C_0 répond négativement à la requête. (4) Une réponse négative est donc envoyée au cache C_1. L'état du répertoire est inchangé. (5) Le cache C_1 aborte sa transaction.

(3A) La ligne C_k est possédée dans l'état **S** par le cache C_0, et a été lue à l'intérieur de la transaction courante. Le résultat de cet exemple n'est pas différent du cas sans les transactions - et n'est en fait pas un cas conflictuel -, mais illustre la différence avec le cas de l'écriture simultanée. (1) Le cache C_1 émet une requête de miss vers la mémoire. (2) Suite à cette requête, une requête d'invalidation *read-only* est envoyée à C_0.

(3B) (3) Comme la ligne a seulement été lue et que l'invalidation est *read-only*, une réponse positive est envoyée et le processeur n'a pas à s'aborter. (4) Le répertoire est mis à jour : C_1 est ajouté à la liste des possesseurs. (5) Une réponse positive est envoyée à C_1. (6) L'état de C_1 est mis à jour et la transaction peut continuer.

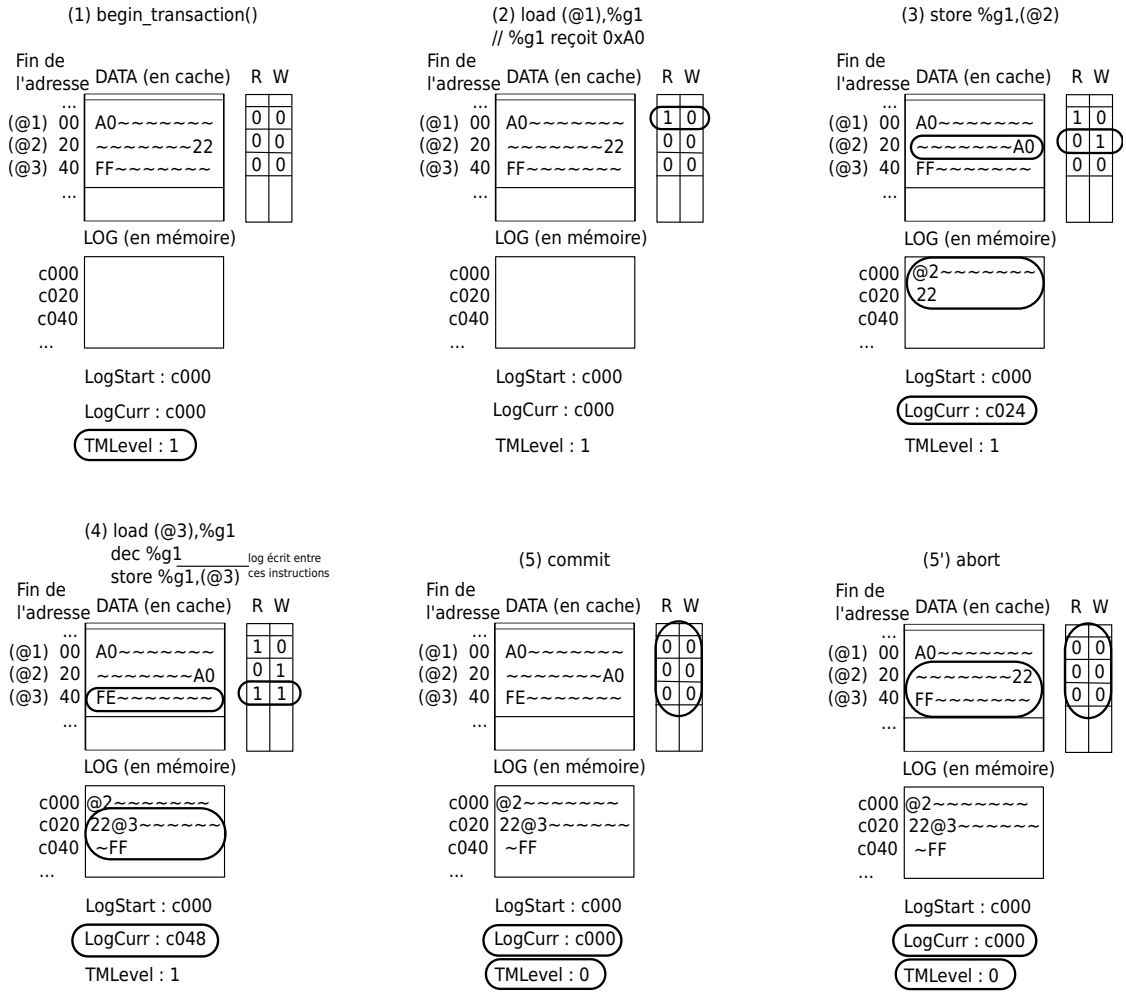
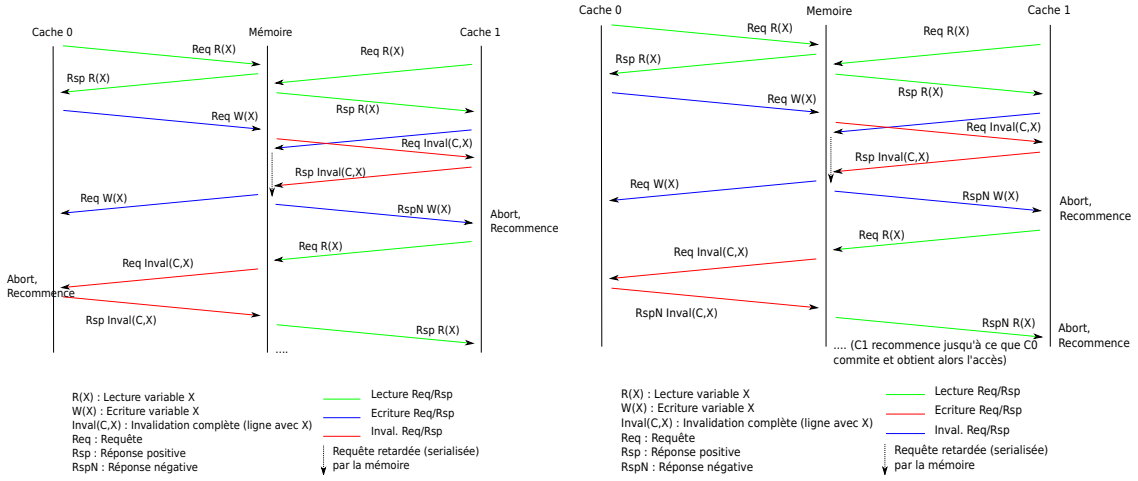


FIG. 6.8 – **Log et gestion de versions dans LightTM-WB**
Cet exemple est presque le même que pour LightTM-WT, la différence ici étant les valeurs écrites dans le log, et les moments où les entrées sont ajoutées. Contrairement à LightTM-WT, les lignes complètes sont copiées dans le log ici, mais seulement dans le cas des écritures. Le champ *Ov* de l'état transactionnel des lignes n'est pas représenté étant donné qu'il n'y a pas de débordement dans cet exemple.
(1) Le processeur exécute l'instruction `begin_transaction();`, incrémentant seulement son registre **TMLevel**.
(2) Le processeur exécute une lecture à l'intérieur de la transaction ; rien n'est ajouté au log, et le bit R de la ligne est mis à 1.
(3) Le processeur exécute une écriture, ajoutant au log l'adresse du début de ligne ainsi qu'une copie de cette ligne. Le bit W est mis à 1. L'écriture n'est pas propagée en mémoire.
(4) Le processeur exécute une lecture-modification-écriture sur un mot d'une ligne commençant à l'adresse @3. Le log est augmenté d'une entrée contenant cette adresse ainsi qu'une copie de la ligne avant l'écriture. Les bits R et W sont mis consécutivement à 1.
(5) Le processeur exécute l'instruction `end_transaction();`, qui commite la transaction. Les seules actions prises ici consistent à réinitialiser le pointeur de log, et de remettre à 0 tous les bits transactionnels du cache.
(5') (Alternative à 5) Le processeur reçoit un signal d'abort en réponse à une autre requête transactionnelle (Nack). Le log est parcouru en sens inverse, et pour chaque entrée du log, la ligne est recopiée dans le cache avec sa valeur initiale. Les bits R et W sont aussi remis à 0.

Aussi, la cachabilité du log a été étudiée via deux implémentations, une permettant au log d'être caché, et l'autre non. Bien que ces résultats ne soient pas présentés ici, cette étude a montré que de cacher le log pouvait faire gagner un temps significatif sur certains noyaux de calcul (jusqu'à 40%), mais avait un impact très limité sur des applications réelles (pas plus de 1%). Pour les expérimentations présentées dans la suite, nous avons néanmoins permis au log d'être mis en cache.

La figure 6.8 montre le même exemple de l'évolution du log et du cache que précédemment, mais avec LightTM-WB.

6.4.3 Résolution des conflits



(a) Exemple d'étreinte active avec une résolution de conflits basique

(b) Priorité donnée aux écritures pour éviter l'étreinte active précédente

FIG. 6.9 – Exemple d'étreinte active basique et stratégie pour l'éviter

La version la plus basique pour résoudre les conflits consiste pour un cache à répondre négativement à (ou Nack-er) une requête d'invalidation quand un conflit est détecté localement. Cependant, cette politique montre vite ses limites dans le cas simple où deux processeurs souhaitent incrémenter la même variable à l'intérieur d'une transaction, et peut mener à une étreinte active. En effet, incrémenter une variable est décomposée de deux requêtes matérielles : une lecture et une écriture. Comme la lecture ne prend que les droits en lecture sur la ligne (état **S**)[1], une autre requête est nécessaire quand l'écriture est émise. Pendant ce temps, et avant que cette requête n'atteigne le contrôleur mémoire, un autre cache peut avoir émis une requête pour lire la ligne, ce qui va déclencher une requête d'invalidation de la mémoire vers le premier cache, le faisant aborter. Ce schéma peut alors continuer indéfiniment (figure 6.9(a)).

Pour éviter ce phénomène - qui est amplifié quand le nombre de processeur augmente - nous avons décidé de favoriser les requêtes d'écriture sur celles de lecture. Puisqu'un cache peut soit Nack-er la requête soit aborter localement, la décision est prise de s'aborter localement quand un conflit est détecté sur une invalidation complète, signifiant qu'un cache souhaite écrire sur la ligne. Cela ne garantit pas l'absence d'étreinte active, mais permet de

[1]En réalité, avec le protocole que nous avons choisi, le premier accès en lecture à une ligne donne le droit Exclusif, mais cela ne change pas le résultat

les éviter dans le cas très commun des "lectures-modifications-écritures" sur les variables (figure 6.9(b)). Par ailleurs, il est intéressant de noter que le cache ne peut pas s'aborter localement et répondre positivement à la requête d'invalidation quand la ligne à invalider a été modifiée, car la ligne à recopier est dans le log, et si ce dernier n'est pas en cache, la requête pour aller chercher le log en mémoire devra attendre au niveau de la mémoire que la requête actuelle soit finie, créant ainsi une étreinte mortelle.

Enfin, afin de réduire la congestion suite à un abort, nous avons ajouté un temps d'attente après un abort, appelé *backoff*. L'étude du backoff sera détaillée dans le chapitre 7.

6.4.4 Gestion des débordements

Contrairement à ce qui est fait dans LightTM-WT, les débordements à l'intérieur d'une transaction dans LightTM-WB sont gérés par un bit de débordement par ligne de cache. Quand une ligne modifiée à l'intérieur de la transaction courante doit être recopiée en mémoire, elle l'est avec un flag spécial (on parle alors de ré-écriture *collante*) pour indiquer au contrôleur mémoire de continuer à faire suivre les requêtes potentiellement conflictuelles au cache. Dans l'absence de conflits, les actions sont les suivantes : les bits de débordement sont remis à 0 avec les bits R et W au moment du commit, et lors de la réception ultérieure d'une requête d'invalidation sur la ligne qui a fait un débordement, le cache se comportera comme s'il avait remplacé silencieusement cette ligne. En revanche, si une requête d'invalidation est reçue sur la ligne en débordement alors que la transaction est toujours en cours, le cache doit alors répondre négativement.

En fait, le flag associé à la ré-écriture dans le cas d'un débordement indique simplement au contrôleur mémoire de marquer le cache faisant la ré-écriture comme possédant toujours la ligne dans l'état **E**.

Cette approche peut mener à de faux conflits, puisqu'un cache peut répondre négativement à une invalidation alors qu'il n'y a pas de conflit, par exemple dans le cas suivant :

1. Le cache C_0 récupère la ligne 0x10004000 en dehors d'une transaction pour une lecture
2. Le cache C_0 commence une transaction
3. Il récupère la ligne 0x10005000 (placée en cache sur la même ligne que 0x10004000), qui remplace silencieusement la ligne 0x10004000, et la modifie
4. Il récupère la ligne 0x10006000, ce qui recopie en mémoire la ligne 0x10005000 (requête de ré-écriture *collante*) et marque la ligne comme ayant débordé.
5. Le cache C_1 réquisitionne la ligne 0x10004000 pour une lecture
6. Le contrôleur mémoire fait suivre la requête au cache C_0
7. Le tag ne correspond pas, mais la ligne a fait un débordement ; un Nack est donc envoyé.

Néanmoins, nous pouvons supposer que ce cas de faux conflit reste relativement rare pour qu'ajouter de la complexité afin de l'éviter n'en vaille pas la peine.

La figure 6.10 illustre un débordement de cache à l'intérieur d'une transaction dans le cas simple d'une transaction sans conflit, sur le même code d'exemple que pour LightTM-WT.

6.4.5 Protocole de cohérence étendu pour les transactions LightTM-WB

LightTM-WB enrichit le protocole de cohérence de cache standard MESI. La table 6.1 résume les actions prises par le cache lorsqu'une invalidation est reçue, en fonction de l'état de la ligne, et de son état transactionnel.

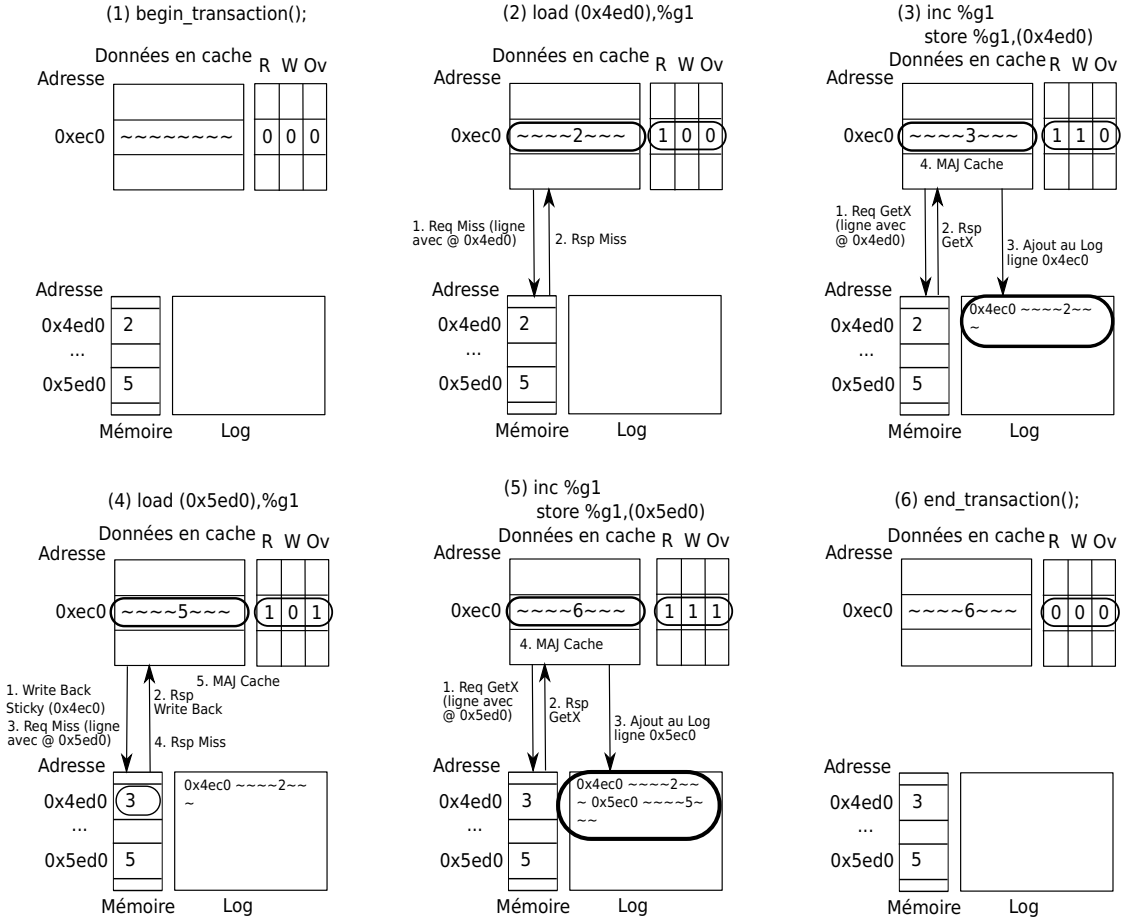


FIG. 6.10 – **Exemple d'exécution d'une transaction dans LightTM-WB avec débordement de cache**

Cet exemple est le même que pour LightTM-WT et montre l'exécution dans LightTM-WB d'une portion de code contenant une transaction accédant deux variables situées dans des lignes mémoire différentes mais placées dans la même ligne de cache. Les hypothèses sur la taille des lignes et des mots sont identiques. Cet exemple permet de mettre en évidence les différences dans la gestion des débordements entre les deux implémentations de LightTM. Les états des lignes en mémoire ne sont pas représentés pour des raisons de clarté.

(1) Le processeur exécute l'instruction `begin_transaction();` ; aucune action spécifique n'est effectuée.

(2) Le processeur exécute une lecture à l'adresse 0x4ed0. Il émet d'abord une requête de miss vers le contrôleur mémoire. Ce dernier répond à la requête positivement, et le bit R du cache est mis à 1.

(3) Le cache souhaite enregistrer une valeur à l'adresse 0x4ed0. Il émet d'abord une requête GetX vers le contrôleur mémoire pour obtenir la ligne dans l'état **M**, puis ajoute une copie de la ligne dans le log. Enfin, il se met à jour avec la nouvelle valeur, et met le bit W à 1.

(4) Le processeur exécute une lecture à une adresse provoquant un débordement de cache dans la transaction (détecté par le bit R étant à 1). Il commence par remettre à 0 les bits R et W correspondants et à 1 le bit Ov, puis envoie vers la mémoire une requête de ré-écriture *collante* pour recopier la ligne couramment en cache. Au niveau de la mémoire l'état de la ligne est changé en **E** (non représenté). Le processeur envoie une requête de miss pour la ligne commençant à l'adresse 0x5ec0. Le contrôleur mémoire répond à cette requête positivement, après quoi le contrôleur de cache met le bit R à 1.

(5) Le processeur exécute une modification puis une écriture sur cette même donnée. Une requête GetX est d'abord envoyée par le cache, et reçoit une réponse positive de la mémoire. Le cache ajoute ensuite la ligne au log, et met le bit W à 1, avant de mettre à jour la ligne de cache avec la nouvelle valeur.

(6) Le processeur commite sa transaction. Tous les bits transactionnels sont remis à 0, et le registre LogCurr est réinitialisé avec la valeur du registre LogBase. On considère qu'ainsi le log est logiquement vide, même si en réalité les valeurs du log sont inchangées.

TAB. 6.1 – **Actions prises par un cache lorsqu'une invalidation est reçue**
'-' indique que le bit correspondant est à 0, tandis que 'X' indique que le bit correspondant a une valeur indifférente. Les configurations qui ne sont pas listées ne sont pas possibles. La première partie de la table résume les actions pour le protocole MESI standard.
Les deux configurations marquées d'une astérisque ne sont habituellement pas possibles puisque de telles requêtes devraient être traitées au niveau de la mémoire seulement.

État de la ligne	**État transactionnel de la ligne**			**Type d'invalidation**	**Action(s) prise(s)**
M,E,S,I	**R**ead	**W**ritten	**O**verflowed (Débordé)	**C**omplète ou **R**ead-**O**nly	Ack, Nack, abort local Inval, Inval-RO
I	-	-	-	RO,C	Ack
S	-	-	-	RO	Ack (*)
S	-	-	-	C	Inval, Ack
E	-	-	-	RO	Inval-RO, Ack
E	-	-	-	C	Inval, Ack
M	-	-	-	RO	Inval-RO, Ack
M	-	-	-	C	Inval, Ack
S	R	-	-	RO	Ack (*)
S	R	-	-	C	Inval, abort local, Ack
S	R	-	Ov	RO,C	Nack
E	R	-	-	RO	Inval-RO, Ack
E	R	-	-	C	Inval, Ack
E	R	-	Ov	RO,C	Nack
M	X	W	-	RO,C	Nack
M	X	W	Ov	RO,C	Nack

6.5 Caractéristiques de LightTM

6.5.1 Interface

L'interface de LightTM est restreinte à cinq primitives :

- **begin_transaction()** indique que les instructions suivantes font partie d'une même transaction jusqu'à ce qu'un appel à `end_transaction()` soit fait. Si le processeur n'était pas déjà en mode transactionnel, il effectue une sauvegarde de ses registres. Toutes les requêtes suivantes effectuées par le processeur sont transformées en requêtes équivalentes transactionnelles.
- **end_transaction()** termine la transaction et la commite. Dans le cas où plusieurs transactions ont été commencées de manière imbriquée, les `end_transaction()` internes n'ont pas d'effet.
- **abort_transaction()** aborte la transaction en cours. Dans un premier temps, nous n'avons pas implémenté cette primitive, puisque nous pensons que cela ne devrait pas faire partie du modèle de programmation de laisser l'utilisateur avoir un contrôle sur cette mécanique interne. Néanmoins, certains des benchmarks utilisés par la suite

requérant de faire des aborts de manière explicite, nous l'avons implémentée.
- **store_log_address(int* address)** Enregistre l'adresse de base du log dans le registre `LogStart`. Dans l'implémentation actuelle, l'utilisateur doit faire lui-même l'allocation avant d'enregistrer l'adresse correspondante, mais une solution avec un appel à un traitant spécial en cas de débordement est envisagée.
- **store_log_size(int size)** Enregistre la taille du log dans un registre spécial. Le but de cette primitive est de pouvoir détecter les débordements de log.

6.5.2 Imbrication

Comme expliqué dans le chapitre 5, nous avons décidé que notre système aurait la sémantique la plus simple : la sémantique à plat. Ainsi, les transactions intérieures d'un groupe de transactions imbriquées sont ignorées. Les commits ne se produisent que lors du `end_transaction()` ; le plus extérieur, et un conflit réinitialise cette même transaction. D'un point de vue implémentation, un appel à `begin_transaction()` ; incrémente simplement le registre `TMLevel` du processeur, et un appel à `end_transaction()` ; le décrémente. Les commits ne se produisent qu'après une décrémentation et que si la nouvelle valeur du registre est 0.

6.5.3 Support du système d'exploitation

Le système d'exploitation (OS) utilisé pour faire tourner les différentes applications est un OS léger nommé Mutek [PG03] avec support pour les pthreads. Nous l'avons utilisé dans une configuration "Ordonnanceur décentralisé", dans laquelle chaque thread est assigné à un processeur à sa création et ne peut pas migrer d'un processeur à un autre. Même si l'OS pouvait utiliser des transactions au lieu de verrous pour ses sections critiques, nous ne l'avons pas modifié pour faire tourner les applications utilisant des transactions, mais avons restreint l'utilisation des transactions au niveau applicatif. L'implémentation actuelle de LightTM a plusieurs limitations au niveau de l'OS ; en particulier, le support pour les opérations d'entrées/sortie est nul. Le cas des allocations mémoire, via `malloc`, est discuté en annexe C : le support pour les opérations d'allocations au sein des transactions est en effet requis par certaines applications, utilisées par la suite. Pour ce chapitre, l'implémentation du `malloc` est à base de verrous, l'allocation dans une transaction est donc impossible.

6.5.4 Coût matériel additionnel

Les deux systèmes LightTM viennent avec un surcout matériel borné. Le CPU nécessite un banc de registres supplémentaire pour sauvegarder la totalité de son état interne au commencement d'une transaction. Dans l'implémentation, un seul cycle est nécessaire pour sauvegarder la registres au commencement d'une transaction. De même, le temps de restauration des anciennes valeurs dans les registres principaux lors d'un abort est d'un cycle. Une ligne d'interruption entre le cache et le CPU est aussi nécessaire pour informer le CPU de l'occurrence d'un abort.

Pour LightTM-WT, le répertoire a besoin d'être augmenté avec quelques registres additionnels, notamment LogStart et LogCurr ; de la logique supplémentaire est aussi requise pour prendre en compte le protocole étendu. Le cache lui-même a besoin de 2 (resp. 3) bits additionnels par ligne pour LightTM-WT : R-W (resp. LightTM-WB : R-W-Ov), en plus de ceux déjà requis pour encoder l'état de la ligne (1 en écriture simultanée, 2 en écriture différée).

La mémoire dans le cas de l'écriture simultanée requiert $3 \times n$ lignes pour n processeurs afin d'enregistrer les requêtes en attente, et le contrôleur mémoire $\lceil \log_2(n+1) \rceil$ bits supplémentaires par ligne pour encoder le processeur possédant la ligne. Dans le cas de l'écriture différée, les seuls changements viennent de la prise en compte du protocole étendu, en particulier les réponses négatives aux requêtes.

Bien que le log se situe en mémoire, sa taille est malgré tout pertinente : elle est d'un mot par ligne accédée pour LightTM-WT, et d'une ligne plus un mot pour LightTM-WB, mais seulement lors d'une écriture. Ces valeurs sont très dépendantes de l'application, mais en pratique, la taille du log pour l'écriture différée est plus grande.

La table 6.2 montre le nombre d'états des FSMs pour les architectures à écriture simultanée et à écriture différée, et leur équivalent LightTM pour les composants cache et mémoire. En effet, si le nombre d'états ne donne pas directement la complexité d'un protocole, il fournit une bonne estimation pour comparer la complexité de deux implantations de protocoles. On peut voir dans cette table que l'implantation originale de l'écriture simultanée est plus simple que celle de l'écriture différée, mais que l'introduction du support pour les transactions supprime cet écart.

TAB. 6.2 – Nombre d'états de FSM dans les différents systèmes

	Cache	Mémoire	Total
Écriture simultanée	38	11	49
LightTM-WT	66	19	85
Écriture différée	48	14	62
LightTM-WB	63	16	79

Nous pouvons donc remarquer que les modifications architecturales requises pour le support des transactions sont plus importantes dans le cas de l'écriture simultanée, bien que les deux systèmes requièrent globalement la même quantité de ressources.

6.6 Évaluation

Cette section évalue dans un premier temps l'approche à écriture simultanée vs. celle à écriture différée pour les deux systèmes de mémoires transactionnelles implémentés, et dans un second temps l'impact de la répartition mémoire pour le système LightTM-WB et une architecture équivalente sans transactions. En effet, une question importante pour les systèmes sur puce est la question de la distribution de la mémoire : comme discuté dans le chapitre 2, les mémoires ne sont que rarement centralisées dans les MPSoCs, au profit de bancs mémoire distribués. Ainsi, LightTM-WB possède *a priori* un avantage sur LightTM-WT puisque ce dernier système ne permet pas d'avoir une mémoire distribuée, du fait que cela n'est pas compatible avec l'implémentation de la détection des conflits. À l'inverse, cela ne demande aucune modification particulière pour le système LightTM-WB.

Les différentes comparaisons sont faites par l'intermédiaire de micro-noyaux et des applications des benchmarks SPLASH-2 [WOT+95].

6.6.1 Architecture et environnement de simulation

LightTM a été implémenté au-dessus d'un environnement de simulation multiprocesseur SPARC.

Les caractéristiques des plateformes simulées sont résumées dans la table 6.3, tandis que la figure 6.11 représente une vue schématique des plateformes utilisées pour les simulations. Pour la comparaison des deux systèmes LightTM, seule l'architecture avec une mémoire centrale (architecture 1) est utilisée.

TAB. 6.3 – Caractéristiques des plateformes de simulation

Nombre de processeurs	$n = 32$, (ou 1 à 32)
Nombre de bancs mémoire	1
Modèle du processeur	SPARC-V8 avec FPU, in order
Taille du cache de données	$16Ko$
Taille d'une ligne (données)	8 mots (32 octets)
Taille du cache d'instructions	$16Ko$
Taille d'une ligne (instructions)	8 mots
Associativité du cache	Correspondace directe
Taille du tampon d'écriture	8 mots
Topologie du NoC	Mesh 2D
Latence du NoC	10 cycles
Déf. d'un cycle sur les graphes	200 cycles simulés

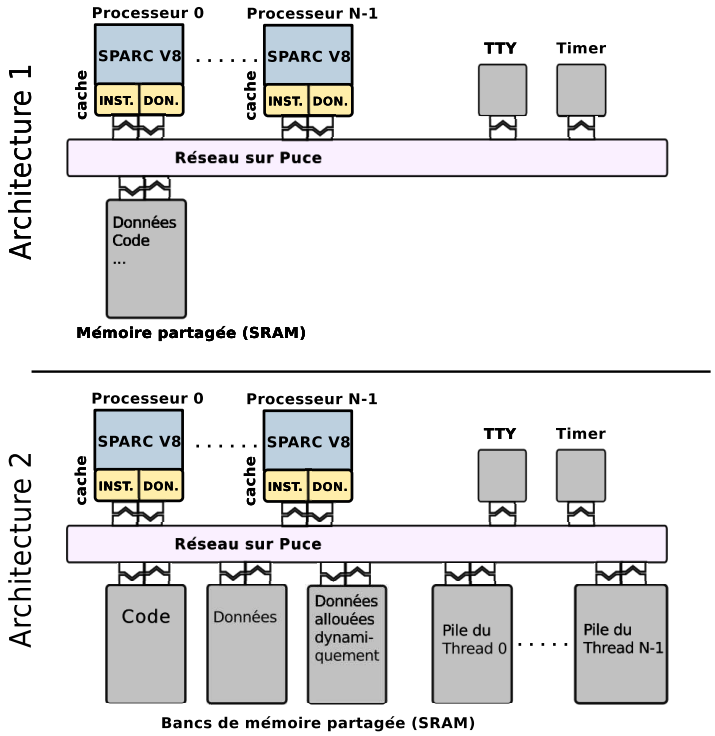


FIG. 6.11 – Plateformes utilisées pour les simulations

Afin de suivre cette approche écriture simulatnée vs. écriture différée, nous avons donc défini deux variantes de cette architecture :

– une configuration utilisant un protocole à écriture simultanée à invalidations (WTI) avec répertoire (configuration écriture simultanée, avec les transactions LightTM-WT)
– une configuration utilisant un protocole WB-MESI avec répertoire (configuration écriture différée, avec les transactions LightTM-WB)

La seconde architecture ne sera utilisée qu'avec le protocole à écriture différée.

Chaque simulation utilise un système comportant de 1 à 32 processeurs et des caches L1 à correspondance directe (avec les instructions séparées des données). La cohérence de cache est maintenue par le protocole défini en fonction de la configuration, au-dessus d'un NoC à bande passante élevée. Le protocole de communication utilisé entre les différents composants est basé sur VCI, enrichi pour supporter les transactions.

L'environnement de simulation utilise le modèle *cycle-accurate* des composants de la bibliothèque SoCLib [The08], qui modélise précisément les différents composants présents sur une puce, mais ne supporte pas les mémoires transactionnelles. Les instructions relatives aux transactions ont été ajoutées dans le modèle du processeur, et sont appelées par l'intermédiaire de macros C. En réalité, seul les appels aux fonctions `store_log_address()` et `store_log_size()` requièrent le décodage d'une nouvelle instruction, les instructions `begin_transaction()` et `end_transaction()` ayant été codées en utilisant les instructions `rd %asr` et `mov %asr` du SPARC. Lors d'un abort, une interruption est envoyée du cache vers le processeur, ce qui restaure alors les valeurs des registres sauvegardées avant le début de la transaction et la recommence.

Chaque application simulée a été compilée avec deux configurations : une avec des spin locks et une avec des transactions. Nous avons utilisé des spin locks et non des mutex locks de manière à ne pas favoriser les transactions : en effet, nous avons supposé un contexte sans commutation de threads, et dans un tel contexte les spin locks sont plus réactifs que les mutex. Nous avons tout de même effectué quelques expérimentations avec des mutex locks, et les résultats ont été bien pire pour les micro-noyaux, et plus lents ou équivalents à la plus lente des 2 autres configurations pour les benchmarks SPLASH-2 testés. Les spin locks utilisés sont des spin locks matériels idéaux de 1 cycle implémentés en mémoire, ne requérant ainsi qu'une requête du processeur pour être pris (le test-and-set étant fait par la mémoire) ou libérés.

6.6.2 Évaluation de l'approche écriture simultanée vs. écriture différée sur les micro-noyaux

Avant de simuler les applications, nous avons essayé de couvrir les cas un peu extrêmes de nos systèmes en utilisant deux micro-noyaux. Nous avons simulé ces micro-noyaux sur des architectures de 2 à 32 processeurs.

6.6.2.1 Premier micro-noyau

Le premier micro-noyau est illustré figure 6.12. Dans ce programme, tous les threads essaient d'accéder à la même variable partagée en parallèle et de l'incrémenter. Le programme s'arrête lorsque la variable a été incrémenté 10 000 fois. Il est évident que plus il y a de processeurs, plus le temps d'exécution est long puisque ajouter des processeurs ne fait qu'ajouter du trafic. Cependant, ce programme permet d'exhiber le comportement des deux systèmes avec une forte congestion.

Sur la figure 6.6.2.1 sont montrés les temps d'exécution pour les transactions. Ces résultats montrent que LightTM-WT, bien que plus lent, est plus stable que LightTM-WB quand le nombre de processeurs augmente puisque le temps d'exécution est presque constant. Ainsi, on peut s'attendre à ce que LightTM-WT ait un meilleur comportement lorsque

```
1       int end = 0;
2       int shared_var = 0;
3       while (end != 1){
4          begin_transaction();
5          if (shared_var == 10000){
6             end = 1;
7          }
8          else {
9             shared_var++;
10         }
11         end_transaction();
12      }
```

FIG. 6.12 – Premier micro-noyau utilisé pour évaluer les systèmes TM avec une congestion élevée

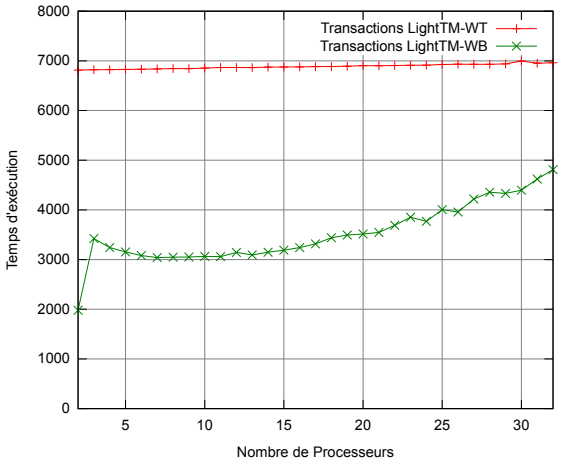


FIG. 6.13 – Temps d'exécution pour le premier micro-noyau avec les transactions LightTM

le nombre de processeurs devient très élevé, même si nous ne sommes pas allés au-delà de 32 processeurs dans nos expérimentations. Cependant, cela n'est peut-être pas la forme la plus représentative d'exécution parallèle puisque le parallélisme est très limité ; c'est pour cela que nous avons défini un second micro-noyau.

6.6.2.2 Second micro-noyau

Le second micro-noyau que nous avons écrit consiste à avoir des variables séparées pour tous les processeurs, chaque processeur incrémentant sa propre variable. Afin d'avoir des résultats significatifs, nous avons changé le nombre d'incréments en fonction du nombre de processeurs, de telle sorte que le nombre total soit toujours de 10 000.

Nous avons aussi considéré un facteur additionnel : le niveau de parallélisme inhérent, relatif au placement des données, et en particulier au niveau des blocs mémoire. Le meilleur cas consiste à avoir toutes les variables situées dans des blocs différents, tandis que le pire cas consiste à avoir des lignes remplies de variables partagées (par exemple dans notre cas avec 32 processeurs, avoir toutes les variables sur 4 lignes de 8 mots).

Pour ce micro-noyau, nous avons trouvé intéressant de comparer les temps d'exécution des transactions à leurs équivalents à base de verrous. En ce qui concerne la granularité des verrous, nous n'avons considéré qu'une granularité fine, i.e. le cas idéal où chaque pro-

cesseur utilise son propre verrou. Même si cela n'est pas très réaliste puisque dans un tel programme les verrous pourraient alors être supprimés, cela reste la situation idéale pour les verrous, et vers laquelle les programmes devraient tendre.

Nous avons ainsi pu définir deux versions du micro-noyau :

1. la version *contigüe* (figure 6.14(a)),
2. la version *non-contigüe* (figure 6.14(b))

```
1  int shared_var[NB_PROCS];
2  /* Initialisation ... */
3  int j;
4  int limit = 10000/NB_PROCS;
5  for (j = 0; j < limit; j++){
6      pthread_spin_lock(lock[
           proc_id]);
7      /* ou begin_transaction(); */
8      shared_var[proc_id]++;
9      pthread_spin_unlock(lock[
           proc_id]);
10     /* ou end_transaction(); */
11 }
```

(a) Version contigüe

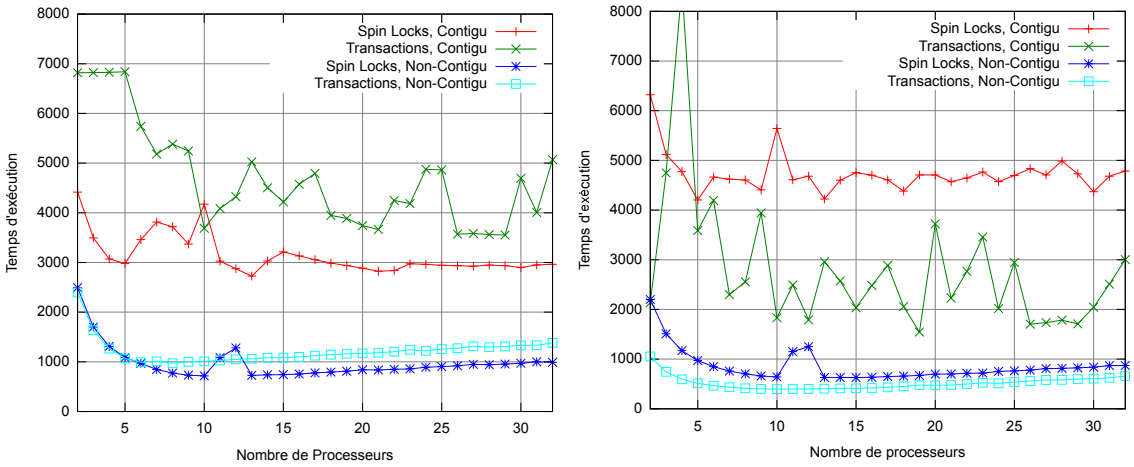
```
1  int shared_var[NB_PROCS*LINE_SIZE];
2  /* Initialisation ... */
3  int j;
4  int limit = 10000/NB_PROCS;
5  for (j = 0; j < limit; j++){
6      pthread_spin_lock(lock[proc_id]);
7      shared_var[proc_id*LINE_SIZE]++;
8      pthread_spin_unlock(lock[proc_id
           ]);
9  }
```

(b) Version non-contigüe

FIG. 6.14 – Représentation simplifiée du second micro-noyau

Bien que ces 2 cas ne soient pas réalistes, nous pensons qu'ils sont assez représentatifs des possibilités d'exécution avec beaucoup de concurrence quand aucune pathologie ne se met en place.

Les résultats de ce micro-noyau sont présentés sur deux graphes : la figure 6.15(a) contient quatre courbes pour l'écriture simultanée (spin locks ou transactions, contigu ou non-contigu), tandis que la figure 6.15(b) contient les mêmes courbes pour l'écriture différée.



(a) Écriture simultanée

(b) Écriture différée

FIG. 6.15 – Résultats du second micro-noyau

Les résultats montrent que les temps des transactions sont proches de ceux des spin locks, pour la version contigüe comme pour la version non-contigüe, montrant que les transactions sont capables d'exploiter le parallélisme inhérent de l'application. En fait, les programmes avec des spin locks sont même plus lents que les transactions pour LightTM-WB pour les deux versions, à cause de requêtes de prise et de relâche de verrous supplémentaires. Cela est plus facile à voir pour la version non-contigüe : une fois qu'une ligne est dans l'état **M** en cache, les transactions s'enchainent rapidement sans accéder à la mémoire principale (les commits sont locaux dans ce cas). Pour la version contigüe, un cache a le temps de faire plusieurs transactions avant de recevoir une requête d'invalidation, c'est pourquoi le nombre de ré-écritures et de récupérations de la ligne en cache est inférieur au nombre de transactions, tandis que pour le programme avec des verrous, 2 accès non cachés sont faits pour chaque transaction.

Avec LightTM-WT cependant, les commits plus lents n'arrivent pas à compenser les requêtes de verrous. Néanmoins, pour les deux architectures, les courbes des transactions montre un comportement globalement identique à celui des spin locks.

Enfin, la comparaison entre l'écriture simultanée et l'écriture différée montre que le LightTM-WB a des meilleurs résultats que LightTM-WT pour les deux versions, bien que ce ne soit pas une surprise étant donné la nature du micro-noyau.

6.6.3 Résultats des applications pour l'approche écriture simultanée vs. écriture différée

TAB. 6.4 – Paramètres des applications

Application	**Données d'entrée**
Mandelbrot	Resol. : 300x200, max iter : 3000 centre : (-0.7436447860 ;0.1318252536) largeur : 0.0000029336
Cholesky	tk14
Ocean	contiguous partitions, 258
Raytrace	teapot (256 x 256)
Water N-Squared	512 molécules
Water Spatial	512 molécules

Les charges de travail sélectionnées pour effectuer la comparaison entre les systèmes sont prises des benchmarks SPLASH-2, excepté Mandelbrot, présentée au chapitre 4, qui est une application consistant à calculer et à dessiner une image de l'ensemble de Mandelbrot, et parallélisée à l'aide d'une technique de vol de travail. La table 6.4 montre la configuration des entrées pour les différentes applications utilisées.

Les accès atomiques sont faits en utilisant des transactions : les macros "(UN)LOCK" et "A(UN)LOCK" de SPLASH ont été définies avec les instructions `begin_transaction();` et `end_transaction();`. La synchronisation entre threads - i.e. les barrières - est faite avec un mécanisme traditionnel à base de verrous, comme dans LogTM [MBM+06a]. Pour la configuration avec les transactions, une initialisation a été ajoutée au début des routines esclaves de chaque application. Par ailleurs, bien que le papier se concentre sur les résultats

des deux systèmes LightTM, nous fournissons aussi pour information les temps d'exécution correspondant avec des spin locks ; ces temps ne sont pas réellement pris en compte dans l'analyse des résultats, mais nous considérons important de toujours situer les performances des transactions.

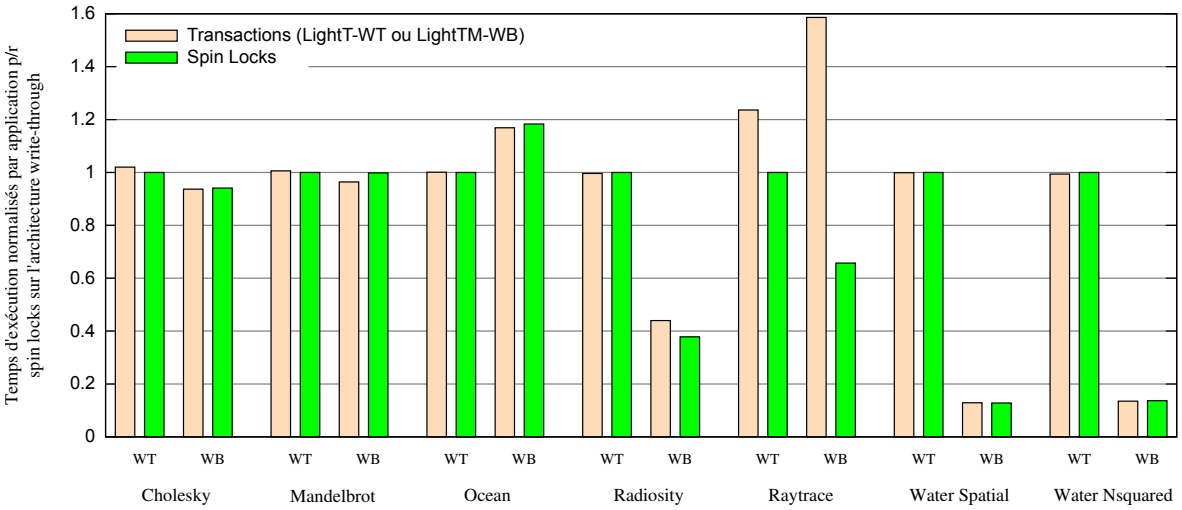


FIG. 6.16 – Temps d'exécution des applications normalisés par application p/r aux temps spin locks en WT

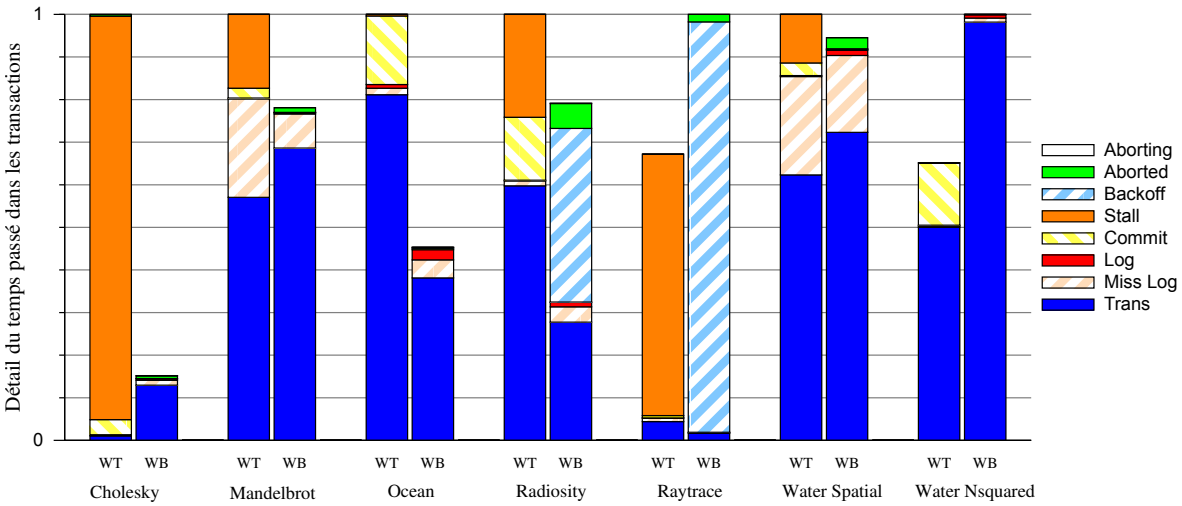


FIG. 6.17 – Détail du temps passé dans les transactions, normalisé par application p/r au temps le plus long (WT ou WB)

6.6.3.1 Temps d'exécution globaux

La figure 6.16 illustre les performances des transactions LightTM et des spin locks pour les sept applications sur les 2 architectures. Pour chaque application, les temps sont normalisés par rapport au temps d'exécution avec des verrous sur l'architecture à écriture simultanée.

TAB. 6.5 – Données transactionnelles des applications pour LightTM-WT

Application	(2) # Trans.	(3) % Aborts	(4) # Read/Trans.	(5) # Write/Trans.	(6) # Lignes/Trans.	(7) # Max ligne/Trans.	(8) % Trans. gelées	(9) # Gel/Trans gel.	(10) % Trans. Deb.	(11) # Deb./Trans. Deb.	(12) % Requêtes opt.
Cholesky	22950	0.06	4.39	4.46	3.57	10	38.9	1.19	0.58	1.04	35.7
Mandelbrot	4885	0	5.16	1.65	2.36	8	0.35	1	0	–	67.6
Ocean	1888	0	3.00	0.42	2.02	3	15.4	1	0	–	98.6
Raytrace	74411	0	28.0	4.49	9.93	30,439	49.0	1.00	0.31	1.01	81.5
Radiosity	1612564	< 0.01	3.88	3.44	2.40	201	10.8	1.00	0.27	2.42	43.6
Water Spatial	609	0	9.02	4.20	1.98	83	6.50	1	0.00	–	69.3
Water NSquared	35360	0	39.4	17.7	7.44	21	0.13	1	0.97	2.77	78.7

TAB. 6.6 – Données transactionnelles des applications pour LightTM-WB

Application	(2) # Trans.	(3) % Aborts	(4) # Read/Trans.	(5) # Write/Trans.	(6) # Lignes/Trans.	(7) # Lignes écrites/Trans.	(8) # Max ligne/Trans.	(9) # Max ligne écr./Trans.	(10) % Trans. Deb.	(11) # Deb./Trans. Deb.
Cholesky	23002	4.32	4.37	3.81	3.56	2.02	10	8	0.55	1.11
Mandelbrot	4715	0.28	5.18	1.44	2.36	0.71	8	5	0	–
Ocean	1888	2.65	3.00	0.44	2.02	0.23	3	1	0	–
Raytrace	74411	55.1	23.1	3.44	8.75	2.43	30,379	4	0.39	1.19
Radiosity	1787478	11.1	3.63	2.74	2.28	1.38	106	46	0.28	2.31
Water Spatial	609	12.8	9.02	4.17	1.98	1.03	83	19	0	–
Water NSquared	35360	0.42	39.4	17.7	7.40	2.97	8	3	0.97	2.77

On peut remarquer sur cette figure que si aucun système n'est clairement meilleur que l'autre, LightTM-WB a un temps d'exécution significativement meilleur pour 3 des 7 applications, alors qu'il est un peu plus lent pour une application (Ocean), et significativement plus lent pour une autre application (Raytrace).

En regard des résultats pour ces benchmarks, la meilleure solution pour une application donnée ne peut pas être connue a priori, le choix de l'architecture pouvant mener à des performances très différentes. Si l'ensemble des applications est défini au moment de la conception, nous pensons que la nature du système TM peut être prise en compte durant l'exploration de l'espace de conception des futurs MPSoCs. Dans le cas contraire, la solution LightTM-WB sera préférée, car requérant un surcout matériel spécifique plus faible, et des résultats sensiblement meilleurs.

TAB. 6.7 – Distribution de la taille des transactions. Les lignes en italiques montrent la taille des transactions pour les écritures

Application	% ≤ 4	% ≤ 8	% ≤ 16	% ≤ 32	% ≤ 64	% ≤ 128
Cholesky	65.1	97.4	100	100	100	100
	88.91	*100*	*100*	*100*	*100*	*100*
Mandelbrot	85.7	100	100	100	100	100
	99.98	*100*	*100*	*100*	*100*	*100*
Ocean	100	100	100	100	100	100
	100	*100*	*100*	*100*	*100*	*100*
Raytrace	62.9	98.7	98.7	98.8	98.9	99.0
	100	*100*	*100*	*100*	*100*	*100*
Radiosity	99.4	99.4	99.7	99.7	99.9	100
	99.7	*99.7*	*99.8*	*100*	*100*	*100*
Water Spat.	99.8	99.8	99.8	99.8	99.8	100
	99.84	*99.84*	*100*	*100*	*100*	*100*
Water NSq.	1.63	99.4	99.9	100	100	100
	100	*100*	*100*	*100*	*100*	*100*

Comparer les temps d'exécution entre transactions et verrous montre qu'en dehors de Raytrace, qui semble être un cas pathologique, les différences de temps sont minimes lors du passage aux transactions. Ceci vient notamment du fait que le pourcentage de temps passé dans les transactions pour ces applications n'est pas très élevé. Ainsi, le choix d'un type d'architecture pour une application doit principalement se faire sur la base du temps total d'exécution. Néanmoins, nous pensons que la comparaison entre ces 2 systèmes transactionnels a du sens, notamment parce que certaines applications peuvent passer un temps beaucoup plus important dans les sections critiques, et que le choix du système peut révéler ou non des cas pathologiques en fonction de l'application.

6.6.3.2 Répartition du temps à l'intérieur des transactions

De manière à mieux comprendre ces résultats, nous étudions la répartition du temps à l'intérieur des transactions pour chaque application. La figure 6.17 montre le détail du temps passé à l'intérieur des transactions pour les deux systèmes. Sur ces histogrammes, Trans représente le temps utile des transactions, Miss Log le temps passé à attendre que le log soit en cache, Log le temps passé à écrire dans le log, Commit le temps passé dans les commits, Stall le temps durant lequel le processeur est en attente d'accès à une ligne dans une transaction (LightTM-WT seulement), Backoff le temps passé à attendre après un abort et avant que la transaction ne recommence (LightTM-WB seulement), Aborted le temps passé à faire du travail plus tard annulé par un abort et Aborting le temps passé dans les aborts.

Encore une fois, si globalement LightTM-WB s'en sort mieux, aucun système n'est toujours plus efficace que l'autre. LightTM-WT passe moins de temps dans les transactions pour deux applications (Raytrace, Water-Nsquared), tandis que LightTM-WB passe moins de temps dans les transactions pour les cinq autres applications. Ce graphe montre qu'en dehors de quelques cas pathologiques (Cholesky WT, Raytrace, Radiosity WB), la plupart

du temps est passé à faire du travail utile. Si le temps passé en commit pour LightTM-WT ne représente pas une portion négligeable (contrairement au cas de l'écriture différée), ce pourcentage reste raisonnable (toujours inférieur à 20%). En écriture simultanée comme en écriture différée, le temps passé dans les aborts ainsi qu'à faire du travail qui sera annulé par un abort est toujours presque négligeable. En effet, les conflits résultent pour les deux systèmes en une perte de temps passée soit en attente (LightTM-WT), soit en backoff (LightTM-WB).

Cholesky met en évidence un cas pathologique en écriture simultanée, dû au fait que de nombreuses données ne sont accédées qu'en lecture, provoquant ainsi la mise en attente de nombreux processeurs, ce qui n'arrive pas en écriture différée.

Par ailleurs, Raytrace est le seul cas où moins de 75% du temps est passé à faire du travail utile (4 et 2% respectivement). Cela s'explique par le fait que les conflits amènent les processeurs à passer la grande majorité de leur temps en gel ou en backoff, et montre qu'écrire des programmes parallèles qui ne sont pas adaptés aux mémoires transactionnelles peut mener à des cas pathologiques. Ces mesures expliquent pourquoi Raytrace a des performances aussi mauvaises comparées à celles des programmes avec des verrous.

6.6.3.3 Données supplémentaires

De manière à caractériser chaque application sur les systèmes LightTM, nous fournissons des données supplémentaires dans les tables 6.5 et 6.6.

La colonne 2 de la table 6.5 donne le nombre de transactions qui ont commitées, tandis que les colonnes 4 à 6 donnent le nombre moyen de lectures, écritures et lignes accédées par transaction. Elles montrent que la taille des transactions n'est pas un facteur déterminant pour les performances, puisque pour les petites transactions (Ocean) comme pour les plus grandes (Water Nsquared), la différence de temps d'exécution entre les programmes à base de verrous ou de transactions n'est pas significative. Les colonnes 10 et 11 suggèrent que même avec de longues transactions exécutant beaucoup de requêtes, très peu débordent du cache, et que donc, pour ces applications, les transactions ne font intervenir qu'un nombre réduit de variables. Cela est corroboré par la colonne 7, montrant que le nombre maximal de lignes accédées dans une transactions est relativement petit ; en dehors de Raytrace qui contient une transaction accédant plus de 30 000 lignes, le plus grand nombre de lignes accédées dans une transaction est de 201 (Radiosity).

La table 6.6 présente les données transactionnelles pour LightTM-WB. Si les données sont relativement proches de celles pour LightTM-WT, deux colonnes relatives aux écritures sont ajoutées, montrant que le nombre effectif de lignes ayant été modifiées reste faible (moyenne et max, colonnes 7 et 9). Cela a de l'importance puisque les écritures sont moins désirables dans les transactions dans ce cas, d'une part car elles résultent plus souvent en des conflits, et d'autre part car elles augmentent la taille du log.

Enfin, la colonne 3 de ces deux tables permet de mettre en évidence la différence de conception entre les deux systèmes : il y a en effet très peu d'aborts dans LightTM-WT qui favorise les gels, puisque seule Cholesky a un pourcentage significativement non nul de transactions ayant subi un abort, mais il y a en contrepartie un pourcentage très haut de gels (colonnes 8 et 9), jusqu'à 49% pour Raytrace. A l'inverse, les transactions LightTM-WB ne peuvent pas être gelées mais font plus d'aborts et passent du temps en backoff (jusqu'à 55% du temps pour Raytrace).

6.6.3.4 Taille des transactions

La table 6.7 représente la distribution approximative en termes de nombre de lignes accédées par transaction, donnant plus de détail qu'une simple moyenne. Les lignes en italique indiquent le pourcentage pour le nombre de lignes écrites à l'intérieur des transactions. Ces valeurs ne changeant presque pas entre les deux architectures, une moyenne des deux est utilisée. En particulier, on peut voir que si les transactions restent relativement petites, 3 applications contiennent des transactions se déroulant sur plus de 64 lignes. Par ailleurs, les transactions accèdent plus de lignes en lecture qu'en écriture, puisque très peu de transactions modifient plus de 4 lignes.

6.6.4 Évaluation de l'impact de la répartition mémoire

Cette section vise à mesurer l'influence de la distribution mémoire avec et sans transactions. La figure 6.18 montre les temps d'exécution pour les sept applications précédentes, sur les deux architectures définies.

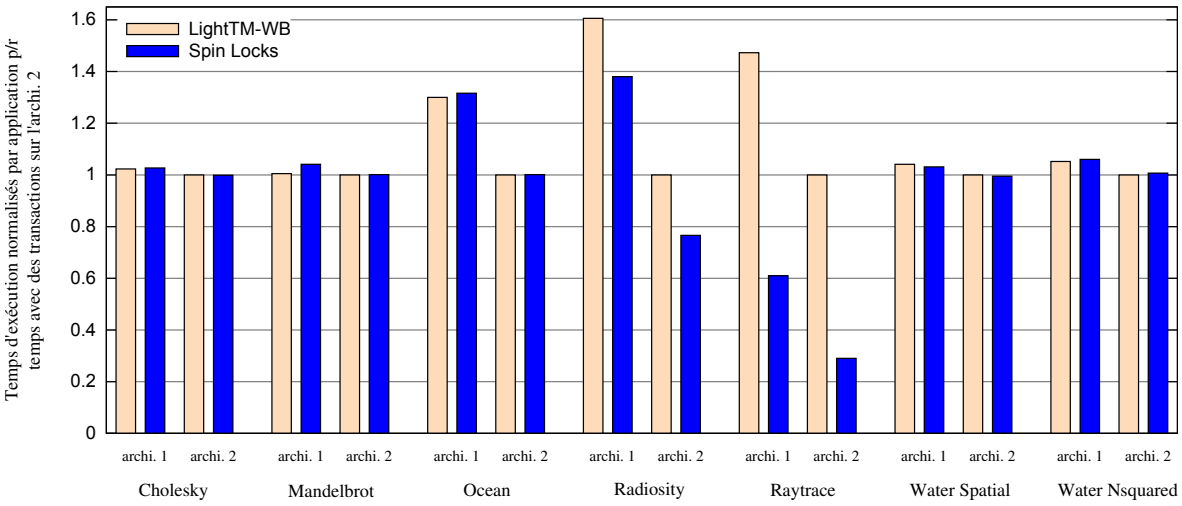


FIG. 6.18 – Temps d'exécution pour l'écriture différée, avec mémoire centrale (archi. 1) ou distribuée (archi. 2)

Ces résultats montrent que la répartition de la mémoire n'est pas cruciale pour toutes les applications : sur les sept considérées, quatre ont des résultats similaires que la mémoire soit centralisée ou distribuée. Néanmoins, pour les trois autres applications (Ocean, Radiosity et Raytrace), les améliorations de performances liées à la répartition mémoire sont significatives, et vont jusqu'à un facteur 3 pour cette dernière application avec les spin locks (facteur de 2 avec les transactions).

Étant donné que la répartition mémoire ne résulte jamais en une perte de temps, mais peut mener à de larges gains pour les applications possédant un niveau de congestion élevé, la distribution physique de la mémoire nous apparait être un point crucial.

On remarque enfin que les transactions ont globalement les mêmes performances que les spin locks, sauf pour les deux applications ayant le niveau de congestion le plus élevé. Cela suggère que les transactions augmentent le niveau de sensibilité à la congestion mémoire. Pour cette raison, LightTM-WB nous semble être une meilleure solution que LightTM-WT comme système TM répondant à notre problème initial.

6.7 Impact des paramètres architecturaux

Cette section vise à étudier l'impact de trois paramètres architecturaux sur les performances des systèmes présentés, pour les spin locks ou les transactions, et sur les deux architectures considérées. Les trois paramètres en question sont le nombre de processeurs, la latence du réseau et la taille des caches. La table 6.8 résume les valeurs utilisées pour ces trois paramètres, ainsi que les valeurs par défaut, c'est-à-dire les valeurs utilisées lors de la variation des autres paramètres.

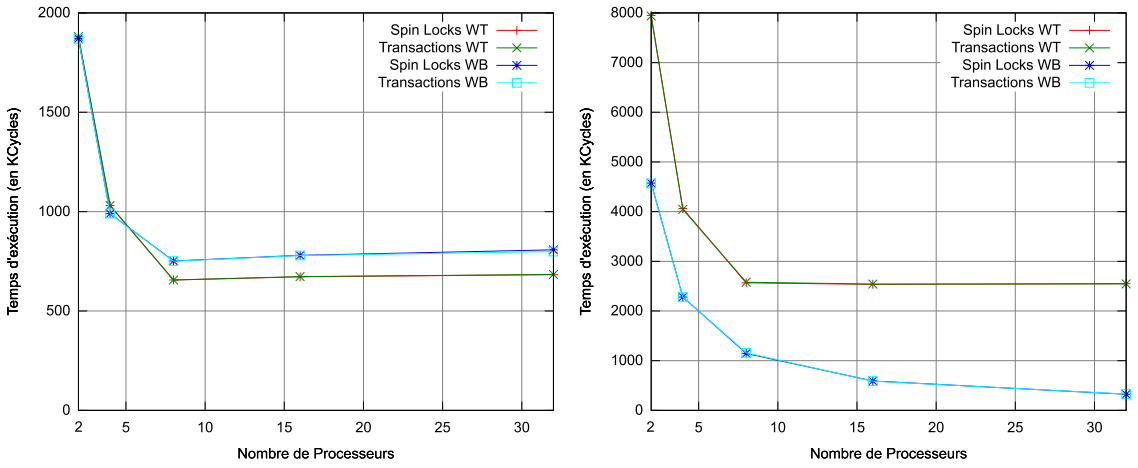
TAB. 6.8 – Valeurs des paramètres considérées

Paramètre	**Valeurs**	**Valeur par défaut**
Nombre de processeurs	2, 4, 8, 16, 32	16
Latence du réseau (cycles)	4, 7, 10, 13, 16	10
Taille des caches (Inst. et Données, en Ko)	2, 4, 8, 16, 32	16

Pour chaque paramètre, deux applications seulement sont considérées (Water-Spatial et Ocean).

6.7.1 Nombre de processeurs

Les résultats des simulations sont présentés figure 6.19.



(a) Ocean

(b) Water-Spatial

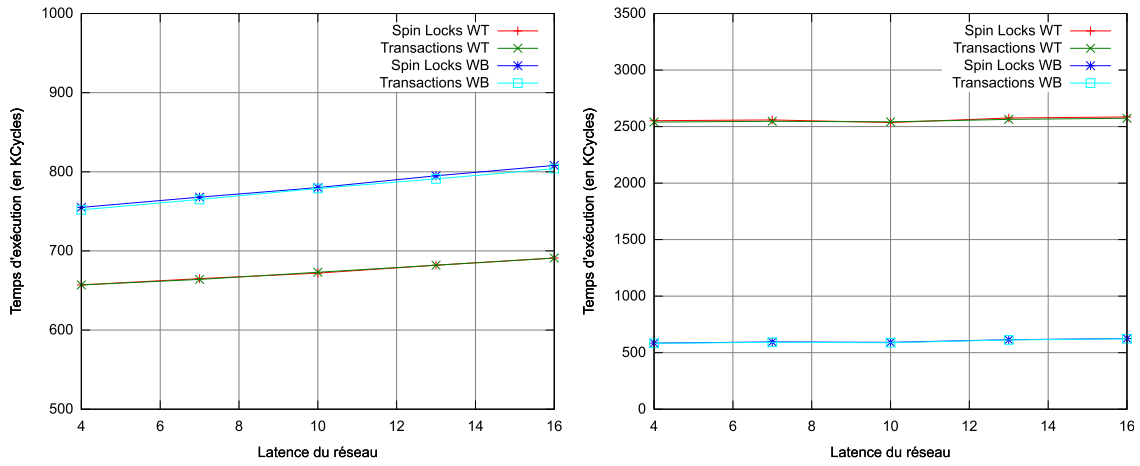
FIG. 6.19 – Impact du nombre de processeurs sur les systèmes LightTM-WT et LightTM-WB et leurs équivalents à base de verrous

Ces résultats montrent que pour les applications considérées, le choix entre transactions et verrous mènent à des temps identiques, et ce quelque soit le nombre de processeurs. Comme discuté précédemment, cela vient du fait que les applications SPLASH-2 n'ont pas des sections critiques très importantes. En revanche, le choix entre écriture simultanée et écriture différée mène à des performances différentes.

Pour Ocean, les temps sont très proches pour 2 et 4 processeurs, puis un écart quasi-constant se forme à partir de 8 processeurs en faveur de l'écriture différée. Pour Water-Spatial, le nombre de processeurs est plus critique et montre que l'écriture simultanée a plus de mal à passer à l'échelle : jusqu'à 8 processeurs, les temps en écriture simultanée ne sont environ que deux fois plus lents, tandis qu'au-delà de 8 processeurs, l'écriture différée arrive à continuer à tirer parti de tous les processeurs, à l'inverse de l'écriture simultanée. Cela mène à un rapport de plus de 7 en faveur de l'écriture différée pour 32 processeurs.

6.7.2 Latence du réseau

Les résultats des simulations avec différentes valeurs de latence du réseau sont présentés figure 6.20.



(a) Ocean

(b) Water-Spatial

FIG. 6.20 – Impact de la latence du réseau sur les systèmes LightTM-WT et LightTM-WB et leurs équivalents à base de verrous

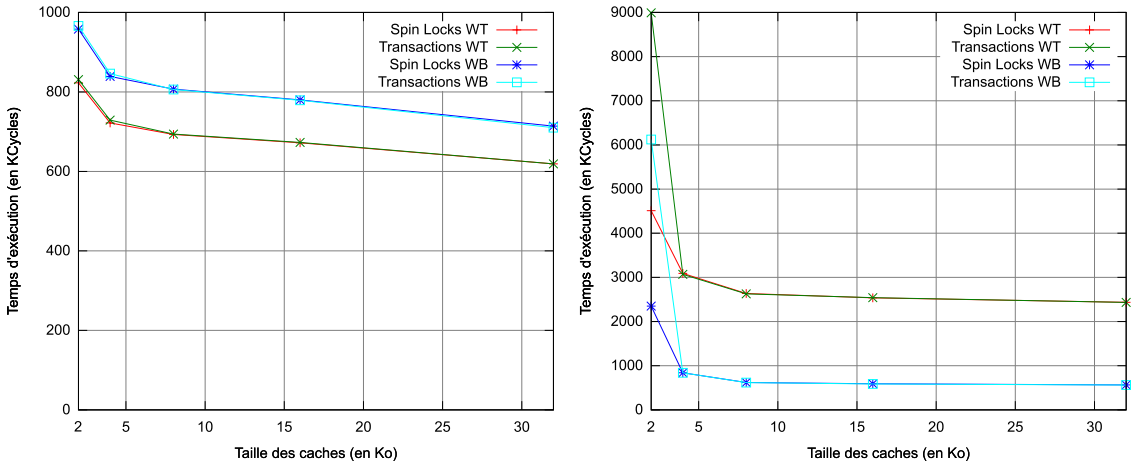
Ces graphes montrent que la latence du réseau n'a qu'un impact très faible sur les performances. En effet, pour les deux applications et quelque soit la configuration, le temps d'exécution croît linéairement avec la latence mais très lentement. Cela se voit clairement uniquement sur les résultats d'Ocean, du fait des échelles différentes. Ainsi, le passage d'une latence de 4 à 16 cycles résulte dans ces expérimentations en au plus 5% de temps d'exécution supplémentaire (Ocean écriture simultanée, spin locks et transactions).

Par ailleurs, on voit que transactions et verrous ont encore une fois les mêmes performances, indépendamment de la latence du réseau.

6.7.3 Taille des caches

La figure 6.21 montre les temps d'exécution pour des tailles de cache différentes.

Comme attendu, le temps d'exécution décroit à mesure que la taille des caches augmente, fortement au début (passage de 2 à 4 Ko), puis plus lentement ensuite. Sur ce point, on peut noter que la taille minimale des caches est plus critique sur Water-Spatial (spin locks et transactions). Enfin, le dernier point à noter est que le temps d'exécution des transactions explose lorsque les caches ont une taille en-dessous d'un certain seuil. Ce seuil est corrélé à



(a) Ocean

(b) Water-Spatial

FIG. 6.21 – Impact de la taille des caches sur les systèmes LightTM-WT et LightTM-WB et leurs équivalents à base de verrous

l'application et plus particulièrement à la taille (spatiale) des transactions dans l'application. Ainsi, pour Ocean, ce seuil n'est pas atteint puisque les temps entre spin locks et transactions sont quasi-identiques, même pour des caches de 2 Ko (le nombre maximum de lignes accédées dans une transaction est de 3). En revanche, Ce seuil est atteint pour Water-Spatial (le nombre maximum de lignes accédées dans une transaction est de 177), puisque les temps sur cette application pour des caches de 2 Ko sont de 2 (WT) et 2.6 (WB) fois plus lents pour les transactions.

6.7.4 Conclusion

Les deux applications choisies ayant un nombre de sections critiques relativement faible, la différence entre verrous et transactions est négligeable devant le choix du protocole de cohérence de cache. Les performances relatives à ce dernier semblent dépendantes de l'application, mais ni du nombre de processeurs, ni de la latence du réseau, ni de la taille des caches. Le seul point à noter concernant les transactions est qu'en dessous d'un certain seuil de taille de cache, dépendant de la taille des transactions, les performances se dégradent très vite.

6.8 Conclusion

Dans ce chapitre, nous avons présenté un système de mémoires transactionnelles implanté en matériel et basé sur un protocole de cohérence de cache à écriture simultanée, dans lequel la détection des conflits est faite au niveau du répertoire. Nous avons comparé ce système TM à un autre, basé sur un protocole de cohérence à écriture différée.

Si aucune réalisation ne surpasse l'autre, laissant ainsi le choix du type de protocole comme un paramètre de l'espace de conception, les résultats sont néanmoins en faveur du système basé sur l'écriture différée. Le couplage de ces résultats à ceux des expérimentations

relatives à la répartition de la mémoire, possible seulement dans le cas de l'écriture différée, corroborent cette conclusion.

Chapitre 7

Étude d'autres systèmes TM matériels

Nous avons vu dans le chapitre 6 une comparaison entre deux systèmes TM matériels basés sur des protocoles de cohérence de cache différents. Comme il s'avère que la politique à écriture différée est plus adaptée à notre problème, on se propose dans ce chapitre d'explorer quelques variations possibles d'un système TM matériel basé sur un protocole à écriture différée.

7.1 Introduction

Jusqu'à présent, nous n'avons considéré que des systèmes ayant des politiques précoces. Peu de travaux dans la littérature ont cherché à comparer différentes politiques. Citons néanmoins [BMV+07] et [SD09] qui ont apporté des éléments de réponse à cette question. Or, il s'avère que cette question est essentielle en vue de la définition d'un système efficace et robuste. Comme les deux familles de systèmes TM utilisées sont les systèmes EE et les systèmes LL, nous nous proposons dans le contexte de notre étude de comparer le système EE présenté dans le chapitre 6 à un système LL, en conservant les mêmes hypothèses architecturales. Cela nécessite dans un premier temps de définir un tel système.

De plus, comme les politiques de résolution des conflits en EE peuvent être multiples, nous comparons celle par défaut à deux autres politiques : une introduite dans [RG02] consistant à utiliser des estampilles pour numéroter les transactions, et une inédite consistant à favoriser les transactions ayant déjà fait aborter d'autres transactions. Nous étudions par ailleurs l'influence du backoff sur les stratégies EE et LL.

Il est difficile de dire si une stratégie est meilleure qu'une autre, notamment à cause du fait que la performance n'est pas le seul élément à prendre en compte. En effet, une autre mesure importante est la robustesse de la solution, ces systèmes étant facilement sujets à des comportements dits pathologiques.

Une pathologie, dans le cadre des systèmes TM, est un comportement imprévu du système résultant d'une interaction avec des entrées particulières et menant soit à une baisse de performances, soit à une étreinte active. Nous nous intéresserons ici principalement à ce dernier cas. En ce sens, l'approche envisagée consiste à considérer le "pire des cas" pour un système, du point de vue de sa progression globale. Nous discuterons donc les avantages et inconvénients de chaque système en fonction de cette caractéristique.

Dans un premier temps, nous allons présenter un système TM matériel basé sur une politique LL.

Étant donné que dans ce chapitre nous nous concentrons sur différents systèmes reposant tous sur un protocole à écriture différée, la terminologie s'en trouve ainsi modifiée :

- *LightTM-WB tel que présenté au chapitre 6 sera désormais dénoté par* ***LigthTM-EE***
- *La variante de LightTM-WB basée sur une politique LL sera quant à elle appelée* ***LightTM-LL***

7.2 LightTM-LL

Nous présentons dans cette section un système TM matériel de type LL – i.e. ayant une détection des conflits et une gestion de version paresseuse – basé sur la même architecture et les mêmes hypothèses que LightTM-EE. Ce système étend le protocole MESI standard en ajoutant un cinquième état, **T**, qui signifie qu'une ligne est dans une transaction. On appelle ce protocole MESIT.

7.2.1 Machine d'état du protocole MESIT

La machine d'état du protocole MESIT de haut-niveau est donnée figure 7.1. Avant lecture ou écriture d'une ligne dans une transaction, un cache doit obtenir la ligne dans l'état **T**. Si la ligne est actuellement dans l'état **M**, il faut alors la recopier en mémoire. Une fois la ligne dans l'état **T**, elle peut être partagée par d'autres caches, mais pour des accès transactionnels seulement. Une invalidation standard sur une ligne dans l'état **T** est Nack-ée, c'est-à-dire qu'une réponse négative est renvoyée et la ligne n'est pas invalidée. Lors d'un commit, une invalidation d'un type particulier (invalidation CO, pour **CO**mmit) est envoyée aux processeurs ayant une copie de la ligne. Après le commit d'une ligne modifiée ou à la réception d'une invalidation CO sur une ligne dans l'état **T**, cette ligne est invalidée. Si une ligne qui est commitée n'a pas été modifiée dans la transaction, elle passe dans l'état **S**.

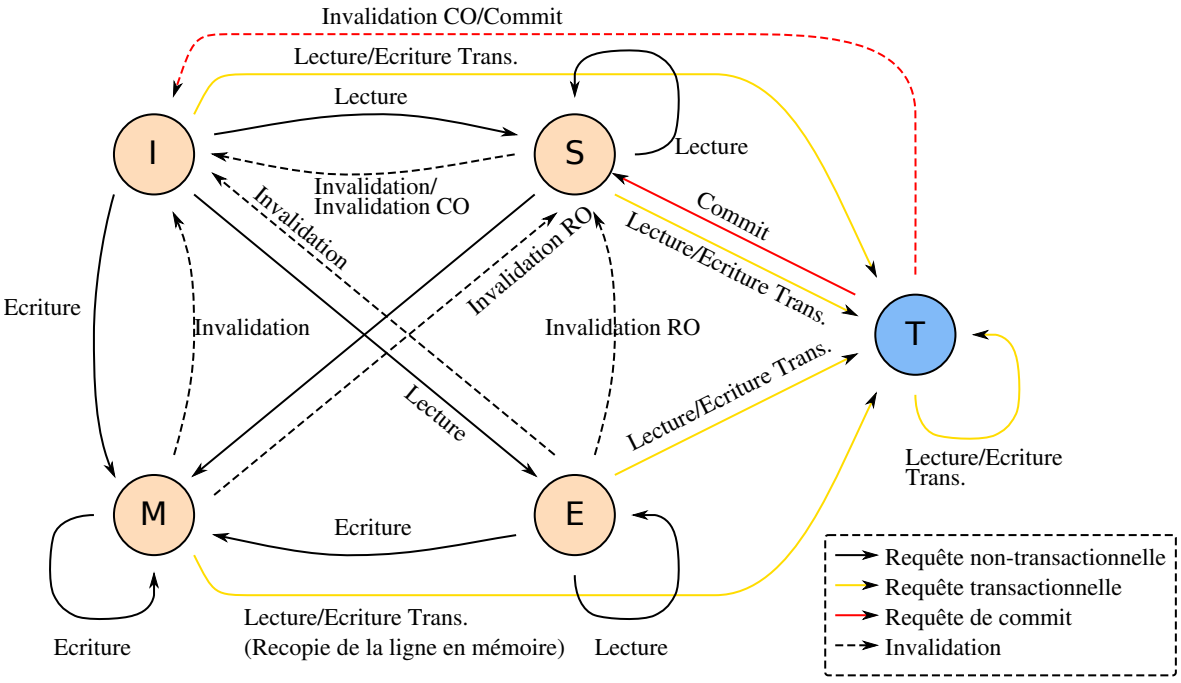


FIG. 7.1 – Automate du protocole MESIT de haut-niveau

7.2.2 Détection et résolution des conflits

LightTM-LL détecte les conflits au moment du commit : lorsqu'une transaction souhaite commiter, elle doit acquérir le jeton de commit. Pour chaque écriture faite dans la transaction, une requête est envoyée à la mémoire, qui envoie un requête d'invalidation CO à tous les caches ayant une copie de la ligne, qui ne peut alors être que dans l'état **T** ou **S**. Le caches recevant cette invalidation prennent la décision de s'aborter si la ligne est dans l'état **T**. Dans le cas où la ligne est dans l'état **S**, la ligne est alors simplement invalidée. Les figures 7.2 et 7.3 illustre le mécanisme de détection des conflits pour ce système, et un résumé des actions prises à la réception d'une requête d'invalidation est donné table 7.1.

TAB. 7.1 – **Actions prises par un cache dans le système LightTM-LL lorsqu'une invalidation est reçue**
'-' indique que le bit correspondant est à 0, tandis que 'X' indique que le bit correspondant a une valeur indifférente. Les configurations qui ne sont pas listées ne sont pas possibles. L'état **I** indique que la ligne est invalide ou pas présente dans le cache. Dans ce dernier cas, la ligne en cache à la place peut néanmoins être valide.

État de la ligne	**État transactionnel de la ligne**	**Type d'invalidation**	**Action(s) prise(s)**
M,E,S,I,T	**Ov**erflowed (Débordé)	**C**omplète, **CO**mmit, ou **R**ead-**O**nly	Ack, Nack, abort local Inval, Inval-RO
I	-	CO	Ack
I	Ov	CO	abort local, Ack
S	-	CO	Inval, Ack
T	X	RO,C	Nack
T	X	CO	Inval, abort local, Ack

On remarque en particulier qu'il n'y a pas de conflit détecté pour les lignes seulement lues par la transaction qui commite, puisque aucune requête n'est envoyée vers la mémoire pour ces lignes. S'il s'avère que ces lignes ont été modifiées par une autre transaction, la cohérence est garantie par le fait que la transaction qui commite est ordonnée logiquement avant cette autre transaction grâce à la sérialisation temporelle des commits. [GAC09] cherche à généraliser cette approche pour d'autres entrelacements de requêtes, et [TPK+09] combine une détection des conflits précoce avec une résolution paresseuse dans ce but.

D'un point de vue implantation, deux bits transactionnels sont donc nécessaires par ligne de cache (au lieu de 3 pour LightTM-EE) : un pour indiquer si la ligne a été accédée dans la transaction, et un pour indiquer un débordement.

L'état **T** est quant à lui implanté avec l'invariant suivant : une ligne est dans l'état **T** si elle est dans l'état **S** et si au moins un de ses deux bits transactionnels est à 1. Cela permet de changer l'état de toutes les lignes transactionnelles qui n'ont pas été modifiées durant la transaction, de **T** vers **S**, lors de la phase de remise à 0 de tous les bits transactionnels. Cette phase a lieu à la fin de la phase de commit, comme pour le système LightTM-EE.

Le transfert du jeton de commit entre les caches est fait par l'intermédiaire d'un réseau dédié avec une topologie en anneau, constitué de deux fils : un indiquant la présence du jeton sur l'interface et un pour notifier la réception du jeton. En effet, il n'est pas possible de transférer le jeton via l'interface VCI sans créer d'étreintes mortelles potentielles. La latence de ce réseau est de 1 cycle.

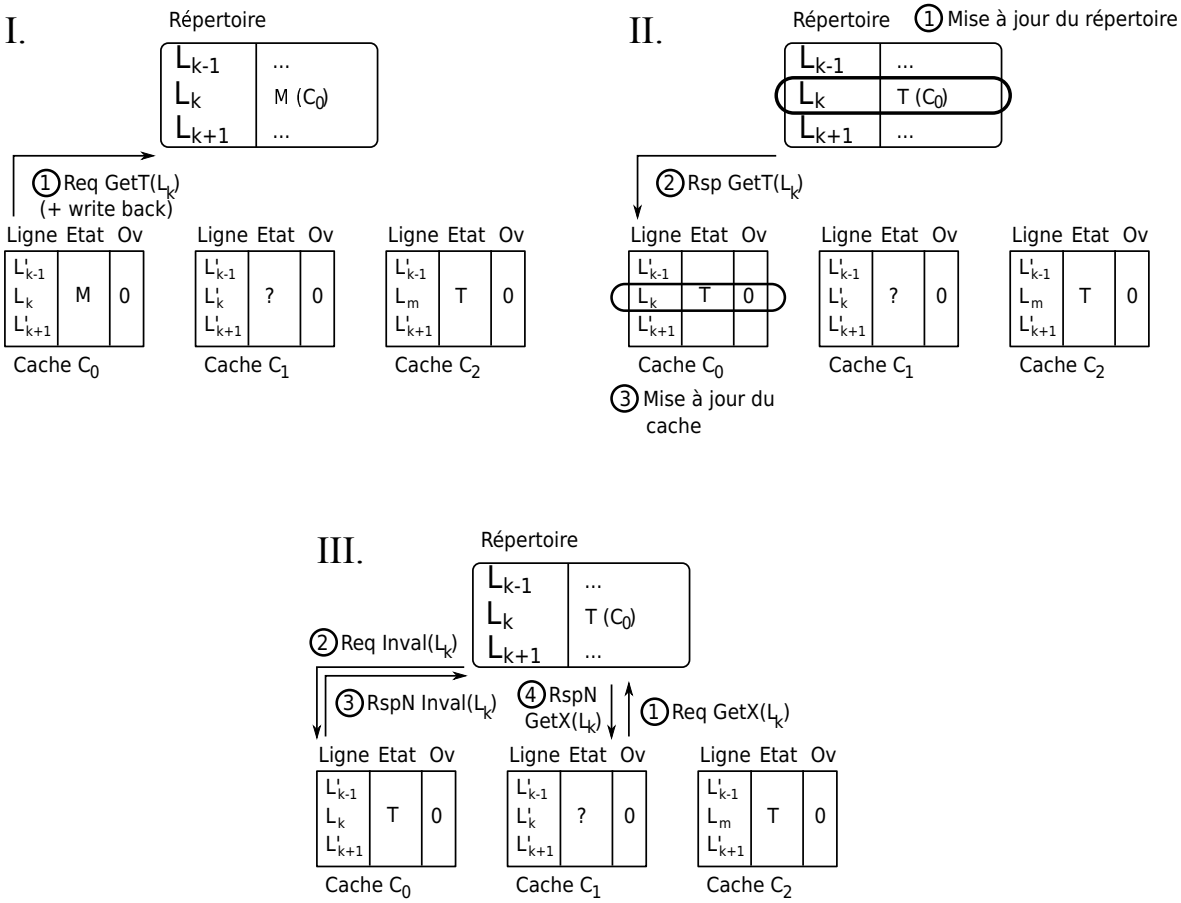


FIG. 7.2 – **Mécanisme de détection des conflits sur LightTM-LL (1/2)**
Cet exemple illustre la détection des conflits sur LightTM-LL. La ligne L_m est une ligne mémoire placée en cache au même endroit que la ligne L_k. Initialement, les caches C_0 et C_2 sont dans des transactions, tandis que ce n'est pas le cas pour les caches C_1 puis C_3.
I. Le cache C_0 possède initialement la ligne L_k dans l'état **M**. (1) Il souhaite accéder cette ligne dans une transaction, et émet pour cela une requête de type GetT, après avoir recopié la ligne modifiée en mémoire.
II. (1) Le répertoire est mis à jour selon cette requête. (2) Une réponse à la requête est envoyée. (3) À la réception de la requête, le cache peut mettre à jour l'état de la ligne, puis faire les accès transactionnels sur cette ligne.
III. (1) Le cache C_1 souhaite faire une écriture non-transactionnelle sur un mot de la ligne L_k. Il envoie donc une requête GetX vers la mémoire. (2) À la réception de cette requête, la mémoire envoie une requête d'invalidation au processeur ayant une copie de la ligne. En effet, même si une ligne est dans l'état **T** au niveau du répertoire, la mémoire ne peut répondre directement sans risquer de créer une étreinte mortelle (à cause des lignes qui ne sont que lues dans les transactions). (3) Le cache C_0 détecte un conflit pour cette invalidation et répond négativement. (4) Le répertoire n'est pas mis à jour et une réponse négative est envoyée à la requête initiale. Suite à cette réponse, le cache C_1 va retenter d'émettre sa requête, jusqu'à ce que la ligne soit commitée.

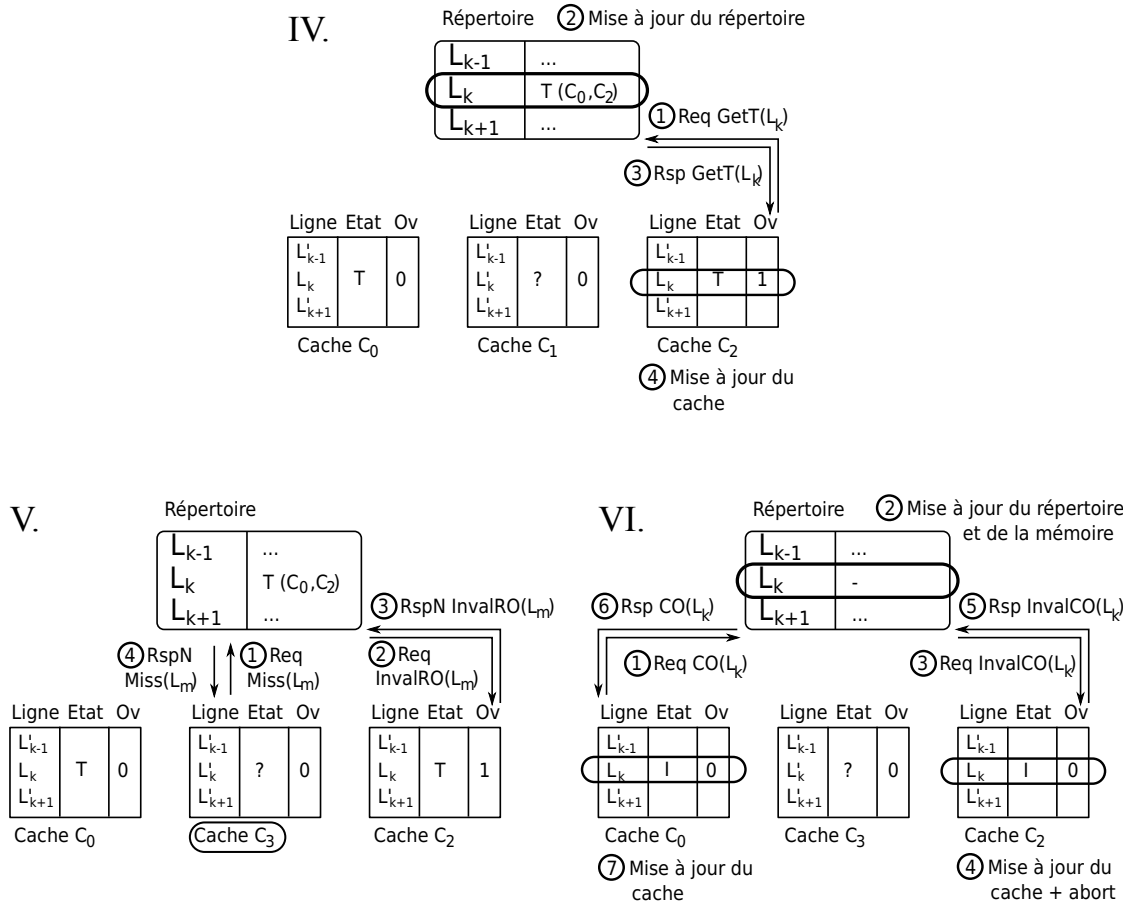


FIG. 7.3 – **Mécanisme de détection des conflits sur LightTM-LL (2/2)**
IV. (1) Le cache C_2 veut accéder à la ligne L_k dans une transaction, alors que la ligne en cache à cet endroit, L_m, a déjà été accédée durant la transaction. Le bit de débordement est mis à 1, puis une requête GetT est envoyée vers la mémoire. (2) La répertoire est mis à jour en ajoutant C_2 à la liste des caches ayant une copie de la ligne, car la ligne est déjà dans l'état **T**. (3) Pour cette même raison, la réponse à la requête peut être faite de manière directe. (4) La ligne est mise à jour dans le cache, de même que le TAG.
V. (1) Le cache C_3, qui n'est pas dans une transaction, veut lire un mot de la ligne L_m, qui a déjà été accédée dans la transaction courante. (2) La mémoire détecte que cette ligne est dans l'état **T** (non représenté), et envoie une requête d'invalidation au cache C_2 qui en possède une copie. (3) Le cache C_2 ne possède pas la ligne, mais contient à la place une ligne dans l'état **T** qui a fait un débordement, il détecte donc un conflit dans le doute (ici, à raison). Une réponse négative est envoyée à la requête d'invalidation. (4) Suite à cette réponse, une réponse négative est envoyée à la requête de C_3, qui va donc retenter sa requête.
VI. (1) Le cache C_0 commite la ligne L_k. Il envoie une requête de type Commit vers la mémoire, avec les nouvelles valeurs. (2) Le répertoire est mis à jour – la ligne est marquée à jour en mémoire, sans copie – ainsi que la mémoire elle-même avec les nouvelles valeurs. (3) Une requête de type d'invalidation C_O est envoyée au cache C_2 qui possède une copie de la ligne, afin de l'invalider. (4) Le cache C_2 invalide la ligne et entre alors dans une phase d'abort. (5) Une réponse positive est envoyée à la requête d'invalidation. (6) Une fois la réponse reçue par la mémoire, une réponse à la requête de commit est envoyée au cache C_0. (7) La ligne est invalidée dans le cache C_0.

7.2.3 Gestion de version

LightTM-LL utilise une gestion de version paresseuse : lorsqu'une écriture a lieu dans une transaction, celle-ci est mise dans un tampon. Au moment du commit, le tampon est vidé et toutes les écritures sont envoyées vers la mémoire. Pour des raisons de facilité d'implémentation et de temps, nous avons fait l'hypothèse que ce tampon avait une taille non bornée. Aussi, les résultats issus de LightTM-LL sont à pondérer par un surcout temporel et/ou matériel, relatif aux débordements de ce tampon. Enfin, lors de la lecture d'une ligne transactionnelle, il est nécessaire, si cette ligne a été modifiée, de lire la valeur présente dans le tampon.

7.3 Autres variantes de systèmes utilisés

Si LightTM-LL définit de par sa conception de manière assez directe les actions à prendre lors de la résolution des conflits, LightTM-EE laisse plus de libertés quant à cet aspect. Nous présentons ici ces variantes, ainsi que l'utilisation du backoff pour les deux systèmes.

L'étude présentée ici suit dans une certaine mesure celle faite dans [SD09]. Néanmoins, nous n'avions pas connaissance de ces travaux lorsque nous l'avons menée, car ces derniers ont été publiés dans la même période.

7.3.1 Utilisation du backoff

Le backoff est une technique qui consiste à ajouter un temps d'attente après un abort, de manière à éviter la congestion engendrée par une transaction qui aborte et recommence continuellement tandis qu'une transaction longue s'exécute ; ou pire, un cas pathologique d'aborts mutuels successifs entre plusieurs transactions se mettant en place du fait qu'aucune transaction n'arrive à commiter avant d'entrer en conflit avec une autre et de devoir aborter.

Pour le rendre plus efficace, on ajoute en général deux propriétés au backoff :

- L'attente d'un temps non fixe, mais pseudo-aléatoire. Chaque processeur est initialisé avec une graine différente (telle que son identifiant), et attend lors d'un abort un nombre de cycles tiré uniformément entre 1 et une valeur maximale, obtenu à partir d'un générateur pseudo-aléatoire. Un générateur aléatoire efficace peut être obtenu simplement en matériel à partir d'un xor et d'un décalage sur une valeur 32 bits. Dans notre implémentation, nous avons utilisé les fonctions `rand()` et `srand()` de la libc.
- Le rallongement du temps d'attente en cas d'aborts successifs. Celui-ci peut-être soit linéaire, soit exponentiel. Dans notre cas, il sera exponentiel, i.e. que le temps d'attente maximum est multiplié par deux à chaque nouvel abort successif. Lors d'un commit, ce dernier est réinitialisé.

7.3.2 Résolution des conflits avec des estampilles

Au lieu de résoudre les conflits avec la stratégie de base, cette variante consiste à utiliser des estampilles pour la résolution des conflits. Ce concept a été introduit dans [RG02] et appliquée à un système TM matériel dans [MBM+06a].

Dans cette approche, toutes les transactions sont ordonnées de manière absolue. Cela requiert d'avoir une horloge commune pour le système, accessible en peu de cycles. Dans l'implémentation, nous avons utilisé le nombre de cycles simulés depuis le lancement du système. En cas d'égalité, le choix se fait en fonction de l'identifiant du processeur.

La décision d'aborter ne se peut se prendre que lors de la réception d'une réponse négative, mais pas à la réception d'une invalidation, comme c'est le cas pour LightTM-EE de base.

Plus précisément, lors de la réception d'une invalidation, on commence par regarder si un conflit est détecté. Si c'est le cas, une réponse négative est alors renvoyée à cette requête. De plus, un flag `NACKED_EARLIER_TRANS` propre au cache est levé si l'estampille de la transaction ayant généré la requête d'invalidation est antérieure à l'estampille de la transaction courante.

À la réception d'une réponse négative, la décision d'aborter est prise seulement si le flag `NACKED_EARLIER_TRANS` est levé. Dans le cas contraire, la transaction recommence seulement la requête qui vient d'échouer.

L'absence d'étreinte mortelle liée à un cycle de dépendances est ainsi garantie, selon le schéma de preuve suivant.

Soit un graphe G dont les sommets sont ordonnés de manière totale. On essaie de construire un cycle dans ce graphe, en respectant les contraintes suivantes :

- Un arc allant d'un nœud n_i à un nœud n_j positionne le flag du nœud n_j si et seulement si $j > i$ (condition de levée du flag)
- On ne peut pas avoir d'arc d'un nœud n_i vers un nœud n_j si n_i a son flag positionné et $i > j$ (condition d'abort)

Soit un cycle dans G, et n_{i1} le nœud du cycle ayant le plus petit numéro (estampille). n_{i1} possède un arc incident vers un nœud n_{i2} ayant un numéro plus grand, donc le flag de n_{i2} est levé. Or, n_{i2} ne peut pas avoir un arc allant vers n_{i1} (condition d'abort), et a donc un arc vers un nœud n_{i3} ayant un numéro plus grand. Le flag de ce nœud est donc levé.

On montre donc par récurrence qu'aucun cycle ne peut se former.

7.3.3 Résolution des conflits basé sur le nombre de conflits gagnés

Une autre idée pour résoudre les conflits consiste à considérer le nombre de conflits gagnés. En effet, plus une transaction a gagné de conflits, plus on a envie qu'elle aboutisse, pour éviter la perte de travail.

Ainsi, cette stratégie consiste à comparer, à la réception d'une requête d'invalidation, le nombre de conflits gagnés par les deux transactions. Si la transaction locale a gagné autant ou plus de conflits que la transaction responsable de l'invalidation, une réponse négative est renvoyée, et fera aborter la transaction responsable de l'invalidation. Dans le cas contraire, une réponse positive est renvoyée et la transaction locale aborte. Dans les deux cas, le nombre de conflits gagnés par la transaction qui n'aborte pas est incrémenté du nombre de conflits gagnés par la transaction qui va devoir aborter (ce dernier est remis à 0 lors de la phase d'abort). Les transactions prennent ainsi du poids au cours de leur exécution.

Si cette stratégie ne garantit pas à chaque instant qu'une des transactions en cours dans le système va commiter, on a néanmoins la propriété suivante. Soit T_1 une transaction. On a :

- T_1 va commiter, ou
- T_1 va se faire aborter par une transaction T_2

Dans ce dernier cas, on a :

- T_2 va commiter, ou
- T_2 va se faire aborter par une transaction T_3

et ainsi de suite. En posant M le nombre maximum d'accès à une variable dans une transaction (i.e. toutes les transactions font au plus M accès à des variables), on peut montrer qu'il existe $i \leq M$ tel que T_i commite.

7.3.4 Conclusion

Au total, 6 systèmes TM matériels seront comparés dans cette section :
- Le système LightTM-EE de base sans backoff
- Le système LightTM-EE de base avec backoff
- Le système LightTM-LL sans backoff
- Le système LightTM-LL avec backoff
- Le système LightTM-EE avec estampille des transactions pour la résolution des conflits (avec backoff)
- Le système LightTM-EE avec résolution des conflits basée sur le nombre de conflits gagnés (avec backoff)

Dans un premier temps, l'utilisation du backoff sera évaluée pour les systèmes EE et LL ; dans un second temps, les systèmes EE et LL seront comparés l'un à l'autre ; enfin, dans un troisième temps, la version de base de LightTM-EE sera comparée à ses deux variantes.

7.4 Évaluation

Cette section compare les performances des différentes variantes des systèmes TM matériels présentés et discute de leurs résultats. Les paramètres architecturaux utilisés pour les simulations sont résumés dans la table 7.2. Ils sont globalement identiques à ceux utilisés dans le chapitre 6, une différence étant toutefois le nombre de processeurs fixé à 16.

TAB. 7.2 – Caractéristiques des plateformes de simulation

Nombre de processeurs	16
Nombre de bancs mémoire	19
Modèle du processeur	SPARC-V8 avec FPU, in order
Taille du cache de données	$16Ko$
Taille d'une ligne (données)	8 mots (32 octets)
Taille du cache d'instructions	$16Ko$
Taille d'une ligne (instructions)	8 mots
Associativité du cache	Correspondace directe
Taille du tampon d'écriture	8 mots
Topologie du NoC	Mesh 2D
Latence du NoC	10 cycles
Déf. d'un cycle sur les graphes	200 cycles simulés

Comme nous ne nous concentrons ici que sur les systèmes TM, nous avons modifié les programmes utilisés pour les benchmarks : nous avons choisi de ne garder que les applications SPLASH-2 exhibant un taux élevé de conflits dans les transactions (Barnes, Cholesky, Raytrace et Radiosity), et nous avons ajouté des programmes des benchmarks STAMP [MCKO08] (Intruder, Genome, SSCA2 et Vacation), plus un programme de Btree.

La suite de programmes STAMP vise ouvertement le domaine des TMs, mais ne propose pas d'implémentation alternative à base de verrous. Le programme Btree a été crée pour l'occasion, et effectue des requêtes en parallèle sur un Btree d'arité 6 (20% d'insertions, 80% de recherches).

7.4.1 Impact du backoff

Cette partie évalue l'impact de l'ajout d'un backoff après les aborts sur les systèmes LightTM-EE et LightTM-LL, pour les 9 programmes présentés au-dessus.

7.4.1.1 Sur LightTM-EE

La figure 7.4 montre les temps d'exécution pour les neufs applications considérées et les deux systèmes.

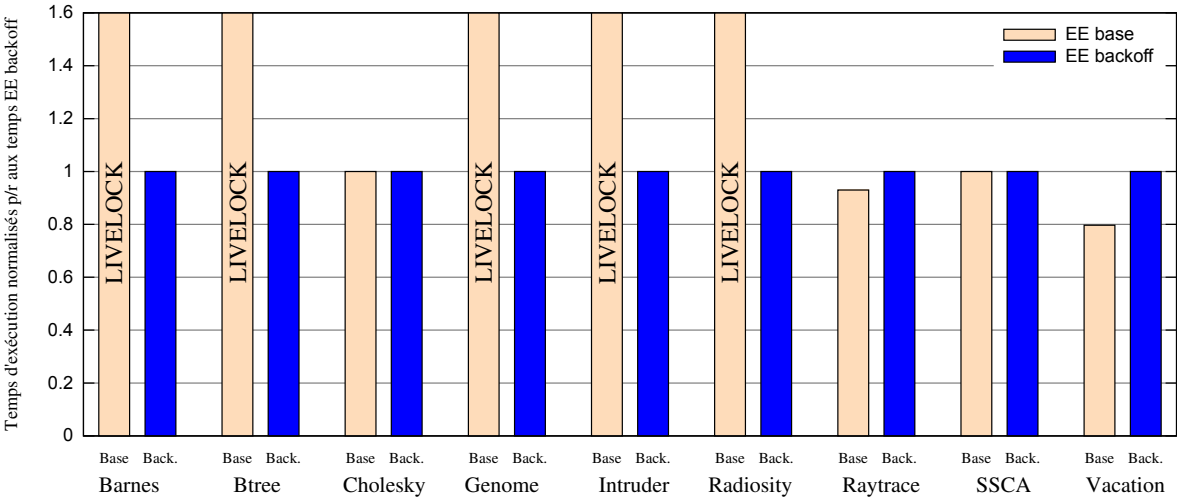


FIG. 7.4 – Impact de l'ajout du backoff dans un système EE (LightTM-EE)

Si l'ajout du backoff a un léger impact négatif sur les performances, il permet de résoudre tous les cas pathologiques d'étreintes actives pour les applications testées. Ces résultats concordent avec ceux présentés dans [SD09].

Pour cette raison, nous concluons que la stratégie EE avec une résolution des conflits basique n'est pas viable sans l'ajout d'un backoff, car il en résulte de nombreux cas d'étreintes actives. *A contrario*, l'ajout d'un backoff permet de supprimer en pratique ces cas d'étreintes actives, et nous préconisons donc qu'une résolution des conflits basique devrait disposer d'un mécanisme de backoff.

7.4.1.2 Sur LightTM-LL

La figure 7.5 montre les temps d'exécution pour les applications considérées et les deux systèmes. Pour la version avec backoff, un processeur en backoff transmet bien évidemment le jeton de commit lorsqu'il le reçoit.

Cette fois, l'ajout du backoff n'impacte les performances que de manière négative : en effet, la politique LL se suffit d'elle-même pour garantir le commit de transactions. Ainsi, pour les applications Raytrace et Intruder, les performances sont respectivement dégradées d'un facteur 2 et 4. Pour les autres applications, les temps sont globalement identiques.

Ces deux cas pathologiques s'expliquent par le fait que pour des transactions de taille équivalente qui s'enchainent, la transaction qui commite va aborter toutes les autres et a de grandes chances de commiter au coup suivant car elle aura probablement commencé en premier la transaction suivante (pendant que les autres processeurs sont encore en backoff). Si cela se répète, tous les autres processeurs seront encore en attente pour un certain temps

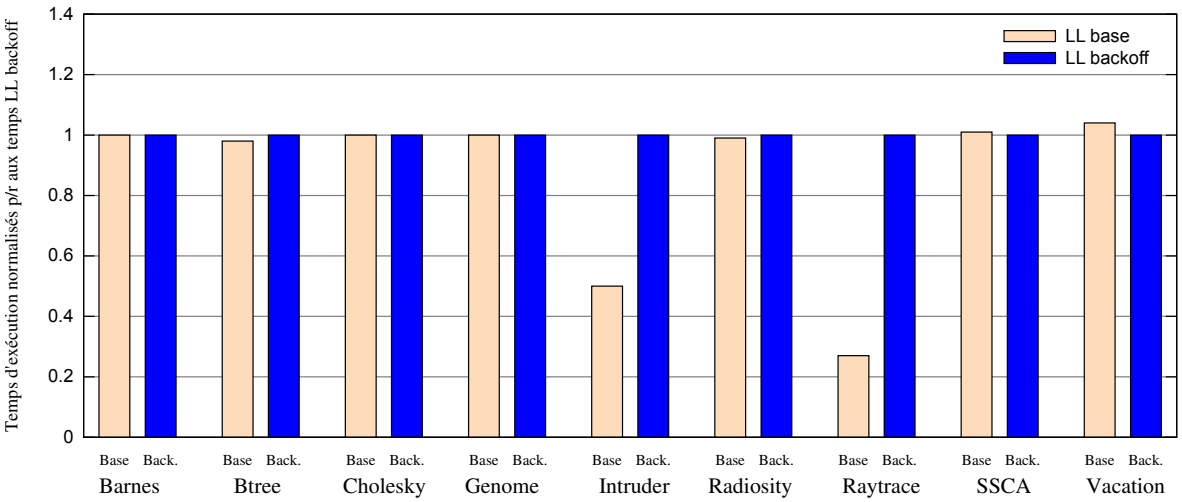


FIG. 7.5 – Impact de l'ajout du backoff dans un système LL (LightTM-LL)

quand ce processeur aura atteint un point de synchronisation, faisant perdre du temps inutilement. Ce schéma peut se répéter pour chacun des processeurs jusqu'à ce que tous aient atteint le point de synchronisation.

Nous en concluons qu'un système LL ne devrait pas inclure un mécanisme de backoff, car en plus du cout relatif à son implantation (même si ce dernier est faible), l'impact sur les performances est négatif, et peut présenter des cas pathologiques résultant en une baisse importante des performances.

7.4.2 Comparaison des politiques EE et LL

Au vu des résultats de la section 7.4.1, nous considérerons la version de LightTM-EE avec backoff et la version de LightTM-LL sans backoff pour les expérimentations à venir. Les temps d'exécution pour ces deux politiques sont présentés figure 7.6.

Ces résultats semblent donner un avantage très net à la politique EE. En effet, pour 8 des 9 applications, le EE est plus efficace, le temps en LL étant en moyenne 1.28 plus lent que le EE, et jusqu'à 2.34 fois plus lent (SSCA2). La politique LL ne se montre quant à elle plus efficace qu'avec l'application Intruder. Ainsi, l'approximation des performances liée à la taille du tampon pour ce système n'est pas critique au regard de ces résultats, puisque cette solution n'a de toutes façons pas des résultats comparables à ceux de la politique EE.

D'après ces résultats, la stratégie EE semble donc être un meilleur choix que la politique LL. Nous comparons cette stratégie à deux variantes dans la section suivante.

7.4.3 Résultats des différentes politiques de résolution de type EE

Si la politique EE présentée jusqu'à présent reste conceptuellement simple et possède un cout d'implantation assez réduit, nous la comparons dans cette section à deux variantes un peu plus élaborées : une basée sur l'estampillage des transactions, et l'autre sur le nombre de conflits gagnés. Les résultats sont donnés figure 7.7.

Si aucune politique n'est systématiquement meilleure ou moins bonne que les autres, la résolution des conflits basée sur le nombre de conflits gagnés s'avère être un peu en-dessous des deux autres : c'est la plus lente sur 5 des applications, et son temps d'exécution moyen

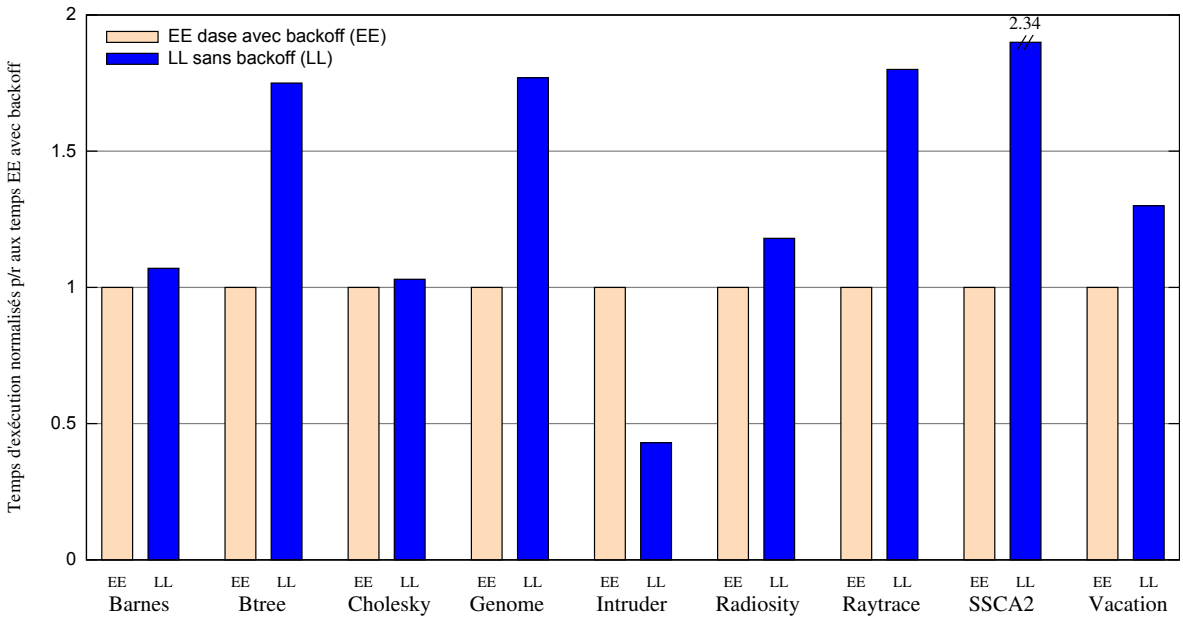


FIG. 7.6 – Comparaison des performances des politiques EE et LL

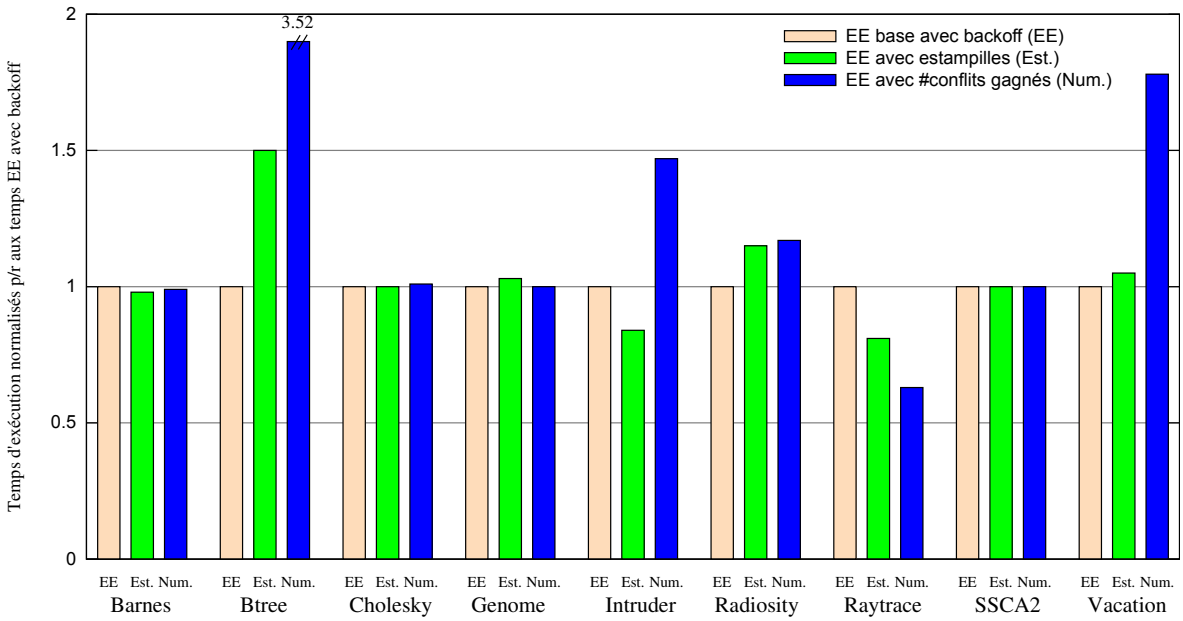


FIG. 7.7 – Comparaison des performances des différentes politiques EE de résolution des conflits

est 1.18 fois celui de la configuration EE de base. Enfin, les deux dernières configurations donnent des résultats sensiblement similaires.

D'après ces résultats, il semblerait donc que le choix à effectuer d'un point de vue performances se porte sur une solution à base d'estampilles, ou plus simplement celle de LightTM-EE présentée dans le chapitre 6.

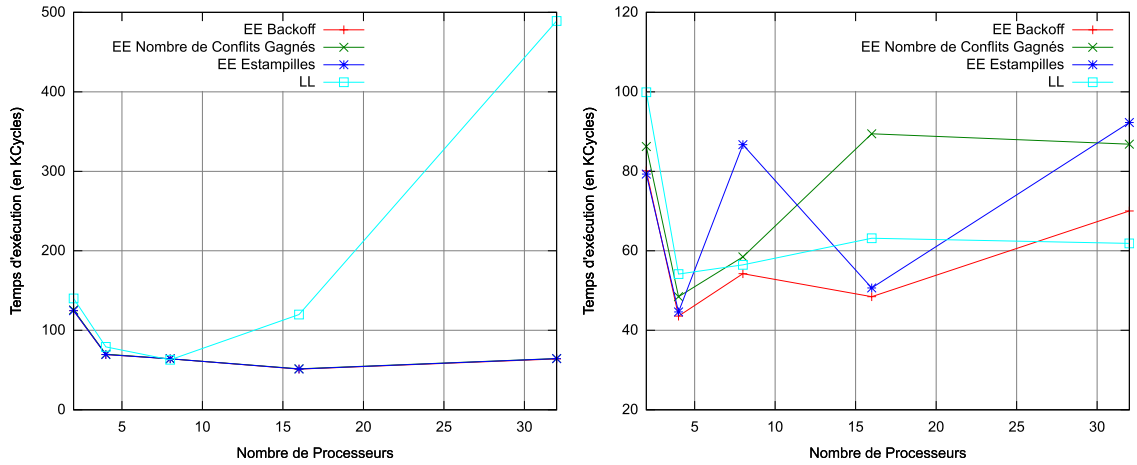
7.5 Impact des paramètres architecturaux

De même que dans le chapitre 6, cette section vise à étudier l'impact de trois paramètres architecturaux (nombre de processeurs, latence du réseau et taille des caches) sur les performances des différentes solutions présentées. Les valeurs utilisées pour les paramètres, ainsi que les valeurs par défaut, sont les mêmes que dans le chapitre 6.

Encore une fois, seules deux applications sont considérées pour ces simulations : il s'agit de SSCA2 et Vacation. SSCA2 mène à relativement peu de conflits, tandis que Vacation possède un grand nombre de sections critiques conflictuelles.

7.5.1 Nombre de processeurs

Les résultats des simulations sont présentés figure 7.8.



(a) SSCA2

(b) Vacation

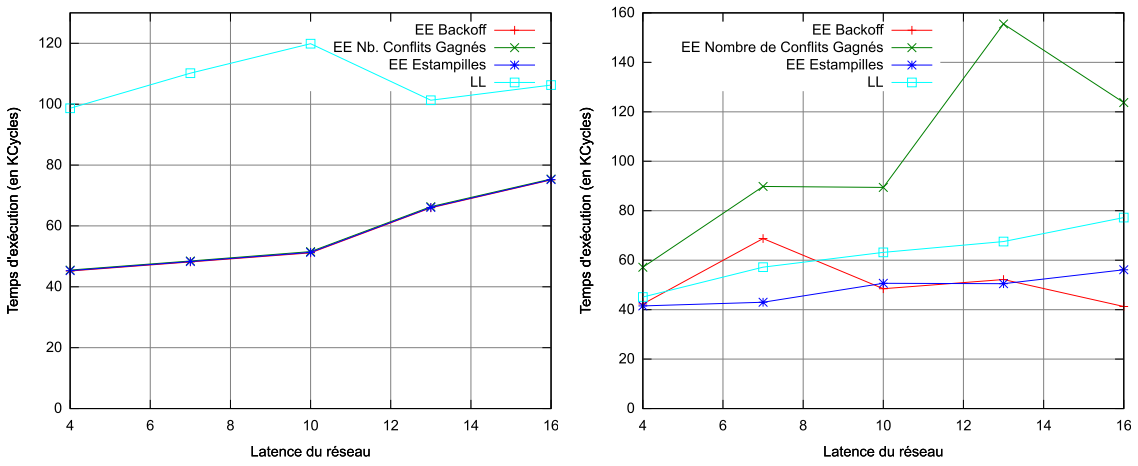
FIG. 7.8 – Impact du nombre de processeurs sur les différentes politiques de résolution des conflits

Pour SSCA2, ces résultats mettent en avant le fait que les 3 politiques EE (courbes superposées en bas) se comportent exactement de la même manière en présence de peu de conflits quel que soit le nombre de processeurs. On pouvait s'attendre à des tels résultats dans le sens où les différences entre les trois systèmes concernent justement la résolution des conflits. D'autre part, ces résultats montrent que la politique LL ne passe pas bien à l'échelle comparée aux autres politiques sous ces conditions. Le surcout apporté par la nécessité d'attendre le jeton devient prohibitif pour 32 processeurs.

Pour Vacation, les tendances sont moins claires, mais on peut néanmoins voir que pour cette application, les conflits sont tels qu'au-delà de quatre processeurs, les temps d'exécution s'allongent. Aucune politique n'est clairement meilleure que les autres, mais celle de type EE backoff semble être celle qui s'en sort le mieux.

7.5.2 Latence du réseau

Les résultats des simulations avec différentes valeurs de latence du réseau sont présentés figure 7.9.



(a) SSCA2 (b) Vacation

FIG. 7.9 – Impact de la latence du réseau sur les différentes politiques de résolution des conflits

Les résultats pour SSCA2 apportent peu d'informations si ce n'est que la politique LL semble moins impactée par une latence de réseau élevée que les autres politiques. Il est à noter que l'anneau utilisé pour le jeton de commit est un réseau dédié et que ses performances temporelles ne varient pas dans ces expérimentations.

Pour Vacation, les temps pour les politiques LL et EE avec estampilles croissent à peu près linéairement. Pour les deux autres politiques, les effets de la latence semblent quelque peu arbitraires, mais ont plus d'impact pour la politique EE basée sur le nombre de conflits gagnés, où les variations sont grandes. L'impact de la latence sur la politique EE avec backoff en présence de conflits est plus modéré.

7.5.3 Taille des caches

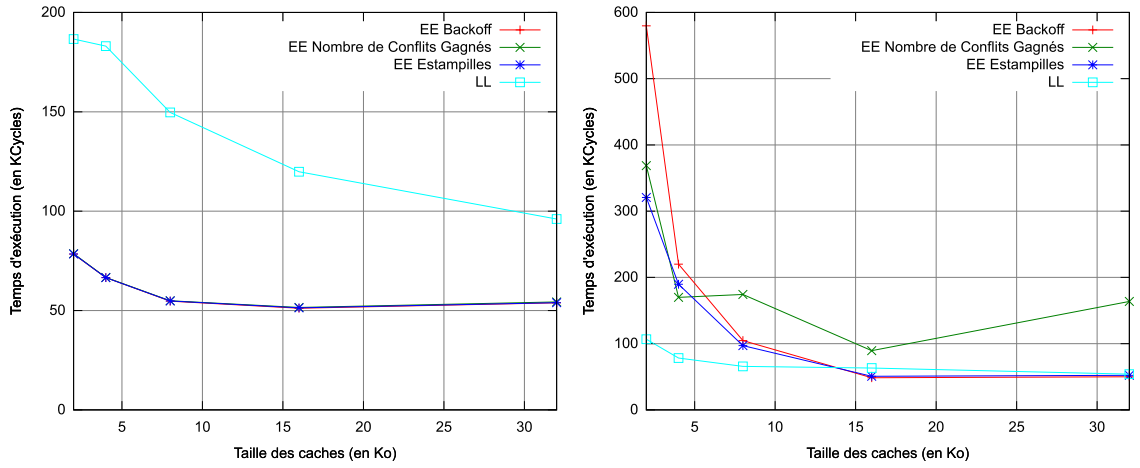
La figure 7.10 montre les temps d'exécution pour des tailles de cache différentes.

Ces résultats montrent que pour SSCA, la taille des caches n'a que peu d'impact pour les trois politiques de type EE. Il est presque surprenant que le passage de caches de 2 Ko à 32 Ko résulte en un gain aussi faible. Pour la politique LL, le gain est plus grand à mesure que les caches grandissent, mais cette politique est toujours largement plus lente que les trois autres.

Pour Vacation, une petite taille de cache couplée à la grande quantité de conflits résultant de l'application provoque une explosion des temps pour les politiques EE, en particulier EE avec backoff. À l'inverse, la politique LL reste modérément impactée par la taille des caches, et a un temps d'exécution près de 6 fois inférieur au temps EE backoff pour des caches de 2 Ko. La politique EE basée sur le nombre de conflits gagnés montre encore une fois qu'elle est plus volatile, c'est-à-dire sensible à la variation du paramètre en présence de conflits.

7.5.4 Conclusion

Ces résultats montrent qu'en présence d'un nombre de conflits faible (SSCA2), les trois politiques EE implémentées ont les mêmes résultats, et ce indépendamment du nombre de



(a) SSCA2 (b) Vacation

FIG. 7.10 – Impact de la taille des caches sur les différentes politiques de résolution des conflits

processeurs, de la latence, ou de la taille des caches. Par ailleurs, ils confirment que sous cette hypothèse, la politique LL est la moins performante. En particulier, cette politique passe très mal à l'échelle comparée aux autres.

En présence d'un nombre de conflits importants, la politique LL ne semble plus aussi inefficace, notamment lorsque la capacité des caches est très petite. Dans les autres cas, elle n'est toutefois pas meilleure que les autres.

Ces résultats sont à confirmer sur d'autres applications.

7.6 Terminaison des programmes à base de transactions

7.6.1 Introduction

Si plusieurs travaux ont étudié des pathologies dans le cadre des mémoires transactionnelles [BMV+07, SD09], très peu se sont intéressés aux garanties portant sur l'absence de famine et la terminaison des programmes dans le cadre des systèmes HTM, en particulier utilisant un NoC. Pour cette section, nous reconsidérons le système LightTM-LL puisque nous nous intéressons ici davantage aux garanties qu'aux performances.

Plusieurs travaux relatifs aux systèmes STM s'intéressent aux problématiques liées aux propriétés de vivacité [GK09a, GK09b], mais ont un cadre qui s'applique mal aux systèmes HTM.

[WS07] propose une solution permettant d'éviter la famine dans un système HTM de type LL, en n'autorisant le commit uniquement pour les transactions ayant été abortées un nombre de fois égal au nombre maximum d'aborts ayant été fait par une transaction en cours dans le système. Le surcout matériel de la solution présentée est important, et le surcout temporel très élevé. Une solution alternative est proposée, dans laquelle on autorise à faire commiter les transactions qui ne vérifient pas cette condition, tant qu'elles ne sont pas en conflits avec une des transactions ayant le nombre maximum d'aborts. Si cette variante permet de limiter le surcout temporel, elle augmente encore de manière non négligeable le

surcout matériel. Comme LightTM-LL a déjà des résultats en-dessous des autres systèmes, appliquer une telle stratégie mènerait sans nul doute à la solution la plus lente et la plus chère.

Enfin, [RG02] a introduit l'idée des estampilles dans le but de favoriser les transactions les plus anciennes, mais cette stratégie ne marche pas avec nos hypothèses (voir section 7.6.3.1), notamment en présence d'un NoC.

Nous présentons dans un premier temps les garanties apportées par les solutions présentées du point de vue du commit des transactions et des implications au niveau de la terminaison du programme. Dans un second temps, nous évoquons les difficultés liées à la mise en place d'un protocole garantissant l'absence de famine. Dans un troisième temps, nous proposons une solution EE qui apporte une telle garantie, si on ignore le problème de débordement des caches. Cette solution a des résultats comparables à LigthTM-EE de base avec backoff.

Pour le reste de la section, on suppose pour des raisons de simplicité d'explications que les programmes considérés ont un nombre de transactions qui n'est pas infini, mais qui peut être non borné (i.e. dans une exécution qui se passe sans problème, le programme s'arrête au bout d'un moment). Dans ce cas, l'absence de famine résulte en la terminaison du programme.

7.6.2 Problèmes posés par les solutions présentées

7.6.2.1 LightTM-EE de base avec backoff

La stratégie EE de base avec backoff, en dépit de ses bons résultats, a l'inconvénient de reposer sur des probabilités. En effet, on ne peut avoir aucune garantie quant au temps avant lequel une transaction va commiter. La seule garantie que l'on a, dans le cas où le nombre de transactions par processeur est fini, est que la probabilité qu'une transaction *donnée* commite est de 1 avec un temps infini. Dans le cas contraire, la seule garantie que l'on a est qu'*une* transaction du système va commiter avec une probabilité de 1 avec un temps infini. Un schéma de preuve est donné ci-après pour deux transactions.

Soient T_1 et T_2 deux transactions en conflit. On pose $Back_n$ la suite des valeurs maximales de backoff pour les transactions : $Back_n = U_0 \times 2^n$, où U_0 est le temps d'attente initial.

Soit U_n le temps d'attente de T_1 après un abort : c'est une suite de variables aléatoires discrètes sur l'intervalle $[1; Back_n]$. On note t_2 le nombre de cycles maximum que prend T_2 pour s'exécuter en l'absence de conflits.

Soit k tel que $Back_k > t_2$. La probabilité que $U_k > t_2$ est $\frac{Back_k - t_2}{Back_k} = p$.

Or avec une loi de Bernoulli, la probabilité pour qu'un évènement de probabilité $p > 0$ ne se réalise pas après n essais est $(1-p)^n$ et $\lim_{n \to +\infty} (1-p)^n = 0$.

En d'autres termes, la probabilité pour que l'évènement se réalise pour un k' donné est de 1.

On a donc qu'il existe k' tel que $U_{k'} > t_2$. De plus ici, comme $Back_n$ croît de manière exponentielle avec n, p se rapproche très vite de 1, donc il faut en moyenne peu d'essais pour atteindre ce k'.

Cela signifie que le temps durant lequel T_1 va attendre en backoff est inférieur au temps maximum requis pour que T_2 se complète sans conflit. T_2 va donc arriver à terme. Si le nombre de transactions dans le système est fini, on a trivialement que toutes les transactions vont arriver à leur terme. On remarque aussi dans cette preuve qu'en présence de 2 transactions, il suffit qu'une seule ait un système de backoff pour que le système soit sans étreinte active.

Pour un nombre de transactions quelconque, la preuve est un peu plus complexe mais suit le même principe : au bout d'un moment, toutes les transactions sauf une vont attendre

suffisamment de temps pour que la dernière transaction puisse aller à son terme.

Ainsi, la garantie apportée par cette solution est qu'une transaction quelconque va commiter, i.e. qu'à un moment, une transaction du système va commiter.

7.6.2.2 LightTM-EE avec estampilles

La stratégie utilisant les estampilles pose elle aussi un problème si on ne suppose pas d'ordre fifo entre le canal de réponse aux requêtes et le canal d'envoi des requêtes d'invalidation entre une mémoire et un cache. En effet, si l'ordre fifo n'est pas garanti entre ces deux canaux, le cache est obligé, lorsqu'il attend la réponse à une requête sur la ligne L, et qu'il reçoit une requête d'invalidation sur L, de se souvenir que la ligne L ne sera peut-être plus à jour lors de la réception de la réponse (dans le cas où la requête d'invalidation a effectivement doublé la réponse).

Pour traiter ce problème, il faut donc dans ce dernier cas retenter la requête. Or, le fait de retenter cette requête combiné à la politique de résolution basée sur les estampilles peut mener à une étreinte active, selon le schéma expliqué figure 7.11.

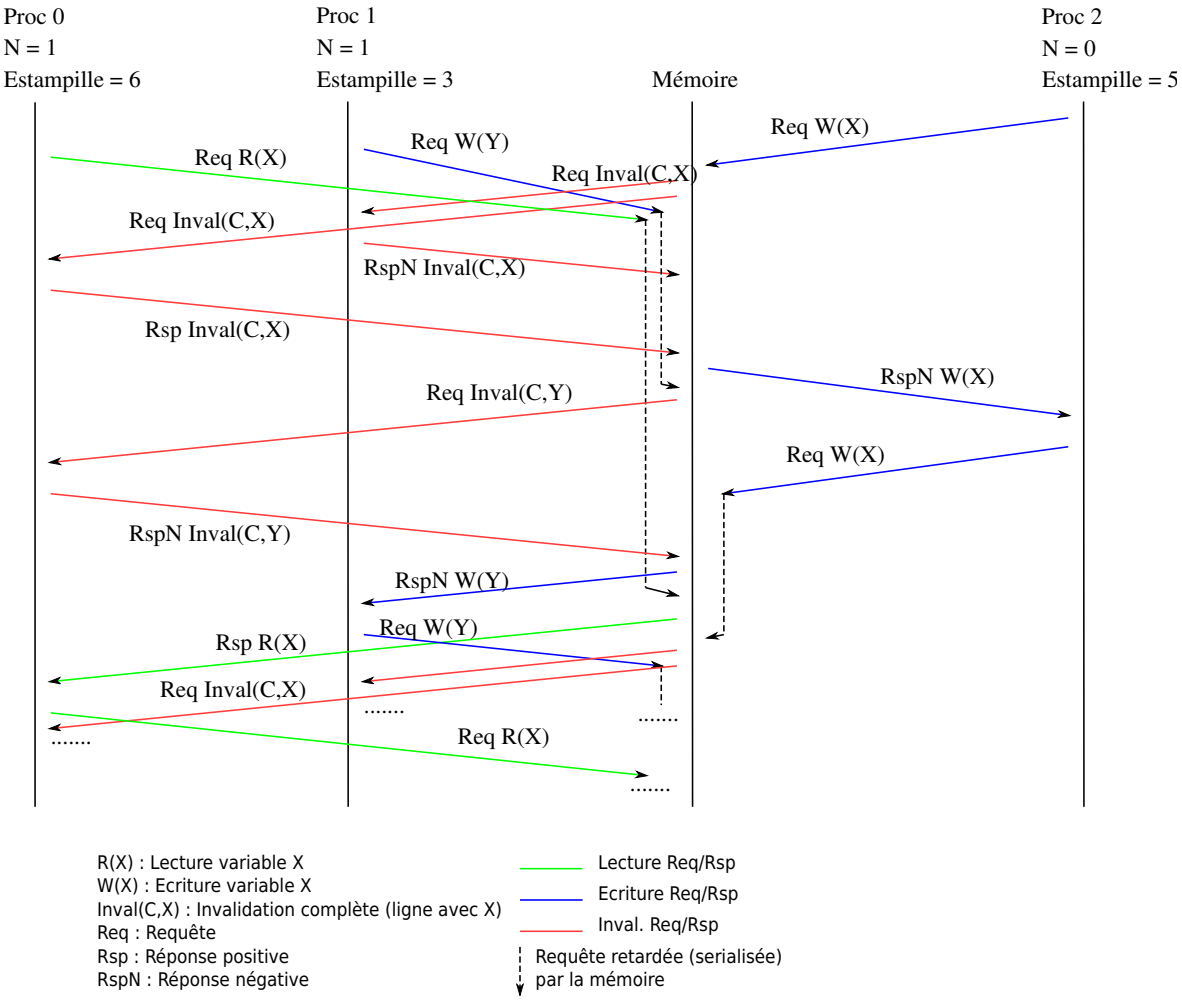


FIG. 7.11 – Étreinte active engendrée par la résolution des conflits par estampilles

En pratique, sans l'hypothèse de fifo, une étreinte active s'est produite pour les applications Barnes, Btree, Intruder et Radiosity. Ce problème limite donc l'applicabilité de cette

solution, ce qui la place en-dessous de la solution LightTM-EE avec backoff.

En dehors de ce problème, nous n'avons pas réussi à montrer de garantie de commit d'une transaction. L'idée derrière cette solution est que la transaction possédant l'estampille la plus ancienne ne peut pas être abortée, et va donc finir par commiter, mais le fait qu'une requête puisse être retentée rend très difficile l'analyse, comme l'illustre l'exemple au-dessus.

7.6.2.3 LightTM-EE avec nombre de conflits gagnés

Cette solution apporte globalement les mêmes garanties que la solution LightTM-EE de base avec backoff, soit qu'une transaction du système va commiter à un moment donné.

7.6.2.4 LightTM-LL Infinité du nombre de transactions et famine

La stratégie LL possède quant à elle la garantie qu'une transaction en cours dans le système va commiter, puisqu'une transaction ne peut se faire aborter que par une transaction qui commite.

7.6.2.5 Résumé

Avec les deux types de propriétés vues (commit d'une transaction quelconque et commit d'une transaction en cours), on a trivialement que toutes les transactions vont arriver à leur terme dans le cas d'un programme avec un nombre fini de transactions. Les différents systèmes sont résumés dans la table 7.3.

TAB. 7.3 – Résumé des garanties pour les différents systèmes. *CTC* : Commit d'une requête en cours. *CTQ* : commit d'une requête quelconque. *P* : Repose sur les probabilités

Système	Garanties	Nombre de transactions	
		Borné	Non borné
LL	CTC	Fin du programme	-
EE Backoff	CTQ (P)	Fin du programme (P)	-
EE Estampilles	??	??	-
EE Nb. Conflits Gagnés	CTQ (P)	Fin du programme (P)	-

Néanmoins, contrairement à ce qui est énoncé dans [SD09], même la garantie de commit d'une transaction en cours (plus forte) ne garantit pas l'absence d'étreinte active : c'est en particulier le cas quand le système possède un nombre non borné de transactions, car il peut se produire une famine.

Par exemple, supposons un système producteur/consommateur avec un producteur et un consommateur synchronisés par des transactions. Supposons que le thread consommateur vérifie la présence d'éléments à traiter dans une transaction qui fait systématiquement aborter la transaction du thread producteur car il y a une sorte de "résonance" entre le moment du commit de cette transaction et le passage du jeton de commit. Dans ce cas, même si le nombre d'éléments à produire et à consommer est fini, le système peut aller dans un état d'étreinte active.

Il est à noter que ce même exemple de producteur consommateur peut mener encore plus facilement à une étreinte active dans le cas des autres types de résolution des conflits.

Par exemple, dans le cas d'une résolution des conflits basée sur une stratégie EE avec back-off, on aura une étreinte active si la transaction du consommateur est plus rapide et fait systématiquement aborter la transaction du producteur.

Pour résoudre ce problème de famine, il faut garantir qu'une transaction échouée soit prioritaire lors des conflits faisant suite à ces échecs.

7.6.3 Obstacles à un protocole garantissant une absence de famine

Cette section essaie de faire un bilan des difficultés relatives à la conception d'une solution garantissant une absence de famine, en explorant d'un point de vue conceptuel trois solutions alternatives à celles proposées précédemment.

En effet, garantir une absence de famine n'est pas une chose facile, car les hypothèses à considérer se doivent d'être réduites : en particulier, on n'a pas la garantie qu'une transaction qui aborte va entrer en conflit avec la même transaction que précédemment, même si cette dernière n'a pas encore commité (ou aborté). Il est possible qu'à chaque recommencement, les données accédées soient différentes, par exemple si le résultat d'un test `if` au début de la transaction alterne entre les valeurs vrai et faux. De plus, il n'est pas forcé que l'écriture responsable de ce changement se fasse dans une transaction, sans que la synchronisation soit incorrecte pour autant.

7.6.3.1 Solution basée uniquement sur les estampilles

On pourrait penser à une solution très simple basée uniquement sur les estampilles pour garantir l'absence de famine : lors d'un conflit, la transaction la plus ancienne gagne toujours.

Néanmoins, cette solution ne marche pas pour la raison suivante : lorsqu'un cache reçoit une invalidation d'une transaction ayant une estampille plus ancienne alors que la ligne de cache sur laquelle porte l'invalidation a fait un débordement, le cache en question ne peut pas répondre positivement et s'aborter, ni répondre négativement.

Il ne peut pas répondre positivement et s'aborter car la valeur qui serait alors prise en mémoire pour la réponse à la requête initiale serait une valeur intermédiaire à la transaction abortée. Il faut donc que ce cache aborte avant de répondre, de manière à récupérer la valeur en log. Or, cela n'est pas possible car l'abort engendrerait des requêtes vers la mémoire, qui ne pourraient pas être traitées, car cette dernière attendrait la réponse à la requête d'invalidation (on aurait donc une situation d'étreinte mortelle).

Enfin, répondre dans ce cas de manière négative serait contraire à la politique, et pourrait donc facilement mener à une étreinte active, par exemple dans le cas où deux transactions débordent sur une ligne puis essaient d'accéder dans l'autre cache la ligne qui a débordée.

7.6.3.2 Solution basée sur le nombre de conflits perdus

Afin de favoriser le commit des transactions ayant dû aborter suite à de nombreux conflits perdus, la solution présentée ici consiste à reprendre l'idée d'un score propre à chaque transaction, et à l'incrémenter non pas après avoir gagné un conflit, mais après en avoir perdu un. Ainsi, une transaction s'étant faite aborter un grand nombre de fois aura un score très élevé et finira par commiter.

Malheureusement, cette solution telle quelle ne tient pas, dans la mesure où elle peut mener à une étreinte active dans le cas de deux transactions en conflit : la transaction qui se fait aborter recommence, gagne cette fois le conflit, puis se fait aborter par l'autre transaction, etc.

Le problème réside dans le fait que la décision pour un même conflit change au cours du temps. Pour y remédier, on peut associer à chaque transaction un identifiant, comme une estampille, et faire qu'une transaction ayant perdu un conflit se souvienne de l'identifiant de la transaction ayant gagné, et perde ensuite tous les conflits face à cette transaction. Lorsque l'identifiant de la transaction en conflit est différent, la décision d'aborter ou non est prise en fonction du score. L'identifiant d'une transaction abortée ne doit pas être changé.

Néanmoins, malgré cette modification et malgré sa complexité, cette solution n'offre pas la garantie d'absence d'étreinte active, des cas pathologiques pouvant se mettre en place avec trois transactions ou plus, par exemple dans le cas de conflits cycliques (figure 7.12).

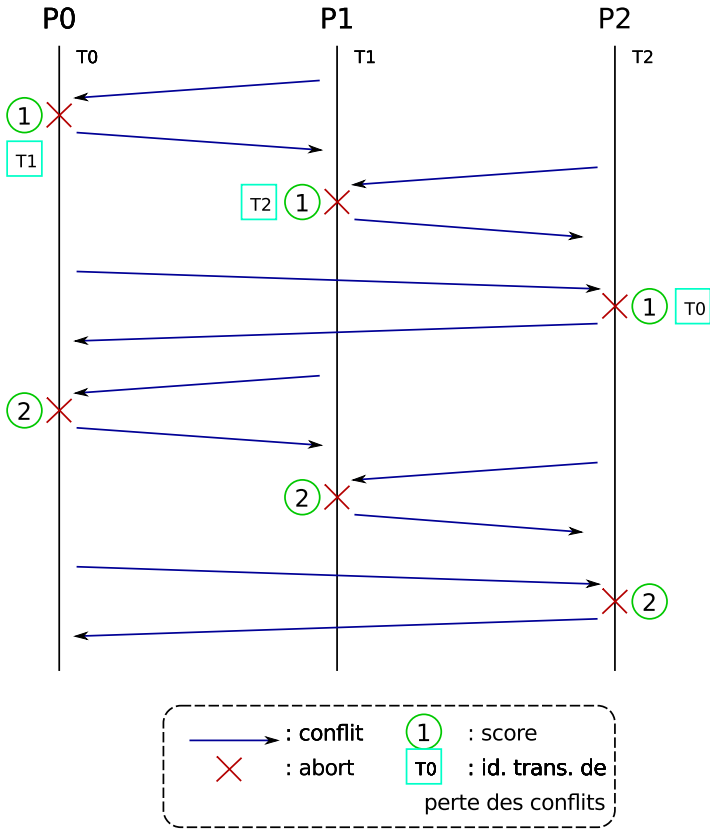


FIG. 7.12 – Exemple d'étreinte active simple dans la solution basée sur le nombre de conflits perdus

7.6.3.3 Solution basée sur le nombre de conflits gagnés et perdus (CGP)

Cette solution tire parti de deux solutions : celle présentée juste au-dessus, et celle à base du nombre de conflits gagnés. Le principe est le suivant : lors d'un conflit, la transaction qui gagne voit son score augmenté de 2 plus le nombre de transactions gagnées par la transaction qui perd, et la transaction qui perd voit son score augmenté de 1.

L'idée qui sous-tend cette solution est de favoriser les transactions qui ont perdu de nombreux conflits, mais de favoriser encore plus les transactions qui les gagnent. Ainsi, dans le cas d'un conflit entre seulement deux transactions, la première transaction à gagner le conflit va gagner tous les conflits suivants, mais le score de la transaction qui perd systématiquement va lui aussi augmenter. Par conséquent, lorsque la transaction gagnante

va commiter, son score redescendu à 0 lui fera perdre tout conflit contre la transaction perdante. Ainsi, cette solution permet de résoudre le problème du producteur/consommateur, comme illustré table 7.4. De plus, elle ne requiert pas d'associer un numéro aux transactions.

TAB. 7.4 – Résumé des garanties pour le système CGP

Système	Garanties	Nombre de transactions	
		Borné	Non borné
EE-CGP	CTD (P)	FP (P)	FP (P)

La figure 7.13 illustre les résultats préliminaires de cette solution face à la politique EE de base. Si cette dernière est toujours un peu plus efficace, les temps proposés par la nouvelle solution sont acceptables.

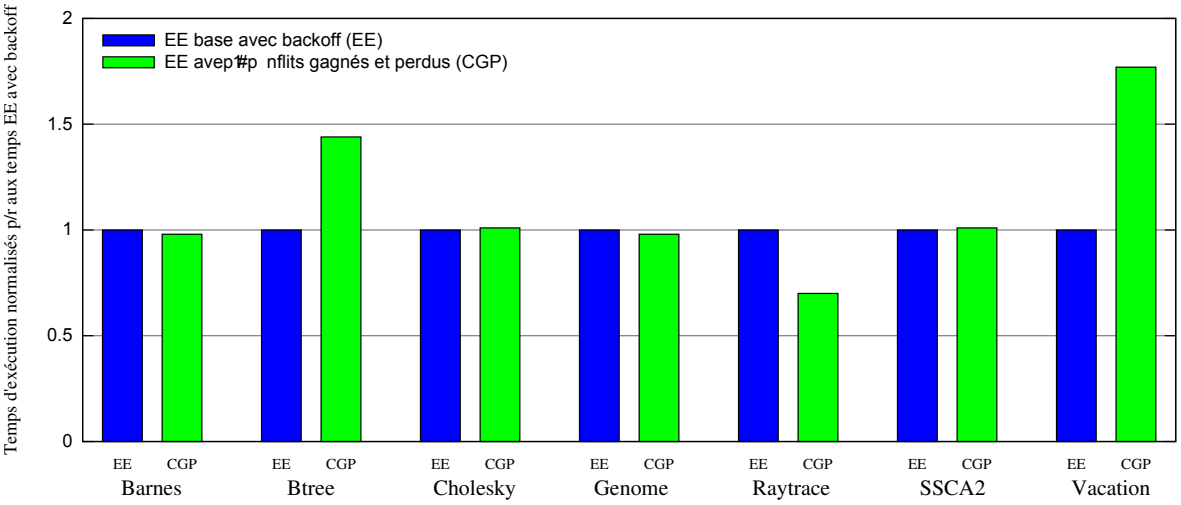


FIG. 7.13 – Performances de la solution basée sur le nombre de conflits gagnés et perdus

Néanmoins, cette solution pose le problème que le score des transactions croît de manière beaucoup plus rapide, et que pour un score codé sur n bits, le nombre de conflits successifs pouvant se produire sans provoquer de débordements est de l'ordre de $\sqrt{2^n}$ (contre 2^n dans les solutions précédentes). Cela pourrait devenir problématique pour des transactions très longues ou un n trop petit.

Par ailleurs, cette solution ne garantit pas complètement l'absence d'étreinte active et requiert l'ajout d'un backoff après abort : en effet, lorsqu'une transaction reçoit une invalidation alors qu'elle a débordé, elle répond systématiquement de manière négative, et ce même si son score est plus faible. Cela est nécessaire, car il se produirait sinon le même scénario que dans le cas présenté section 7.6.3.1, menant à une étreinte mortelle. Cependant, contrairement à la solution basée uniquement sur les estampilles, le problème de débordement de cache ici ne peut pas mener à une étreinte active, mais juste ne plus assurer l'absence de famine.

7.6.4 Conclusion

Pour toutes les solutions basées sur le EE se pose le problème des débordements, en particulier lorsqu'une transaction ayant débordée reçoit une requête d'invalidation créant

un conflit. Ce cas oblige, indépendamment de la politique choisie, à répondre par un abort, rendant très compliqué la garantie d'absence de famine.

Néanmoins, nous avons proposé une solution traitant les autres cas à problèmes, apportant en particulier une garantie probabiliste de terminaison avec un nombre non borné de transactions. Les performances de cette solution sont comparables à celles de la solution EE sans backoff, même si a priori un peu en-dessous.

7.7 Conclusion

Nous avons présenté dans ce chapitre la deuxième partie de notre travail autour des mémoires transactionnelles. Comme il n'existe pas de politique universellement reconnue supérieure pour les systèmes HTM, nous avons étudié le problème avec nos hypothèses. Nous avons implanté un système de type LL et nous sommes intéressés à la comparaison de ce système avec trois systèmes EE ayant des politiques de résolution de conflits différentes. Dans un premier temps, nous avons aussi montré que le backoff n'était pas nécessaire pour un système de type LL, alors qu'un système EE n'était pas viable sans.

Les résultats des comparaisons montrent que parmi les systèmes étudiés, la meilleure solution d'un point de vue performances et avec nos hypothèses est LightTM-EE de base avec backoff.

Nous avons ensuite apporté un peu de précision à ces résultats en étudiant l'influence de trois paramètres architecturaux sur les performances : le nombre de processeurs, la latence du réseau et la taille des caches. Ces résultats confirment entre autres que la politique LL est la moins efficace en présence de peu de conflits, et passe très mal à l'échelle. Dans le cas inverse, cette politique est comparable aux autres, et montre un avantage si en plus les caches sont de petites taille (4 Ko ou moins).

Enfin, nous nous sommes intéressés aux garanties d'absence d'étreinte active dans un système HTM, et avons proposé une solution garantissant une absence de famine entre transactions, mais ne garantissant pas d'absence d'étreinte active autrement que de manière probabiliste.

Chapitre 8

Limitations et travaux futurs sur les mémoires transactionnelles

Ce chapitre présente d'une part les limitations de l'étude menée sur les mémoires transactionnelles, et d'autre part des axes de recherche visant à la compléter.

8.1 Limitations de l'étude sur les systèmes LightTM

8.1.1 Passage à l'échelle

Dans les expérimentations que nous avons faites, nous avons utilisé des architectures ayant au plus 32 processeurs. Or, les architectures du futur posséderont un nombre bien plus élevé de processeurs : les ordres de grandeur de la centaine ou du millier ne sont plus très loin. Or, le passage à l'échelle d'un système TM matériel pour un nombre de processeur aussi élevé n'est pas garanti.

Pour les systèmes de type LL, le problème qui se pose vient de l'unicité du jeton. En effet, l'attente pour obtenir le jeton de commit peut être dans ce cas très longue. Il peut alors se mettre en place des pathologies où un processeur aborte en attendant le jeton de commit, puis est en cours de transaction lorsqu'il reçoit le jeton, et ce plusieurs fois de suite (voire indéfiniment, si cela arrive de manière systématique). En plus des temps de latence augmentés pour les commits, on peut donc avoir des cas de famine qui ne se produisent pas avec un nombre réduit de processeurs car le jeton circule suffisamment rapidement pour masquer le problème.

Pour les systèmes de type EE, la solution basée sur l'écriture simultanée n'est pas viable puisque d'une part la détection des conflits au niveau du répertoire devient couteuse avec un nombre de processeurs élevé, mais surtout la contrainte de n'avoir qu'un seul banc mémoire n'est pas tenable. La solution basée sur l'écriture différée est la plus plausible, mais connaitrait les mêmes difficultés qu'un tel système sans mémoire transactionnelle : le nombre de requêtes émises en écriture différée lorsque le nombre de processeurs est élevé impacte fortement les performance de cette politique de cohérence de cache.

La définition d'un système TM pour une architecture contenant un nombre très élevé de processeur reste donc un réel défi.

8.1.2 Variation des paramètres architecturaux

Dans toutes les expérimentations effectuées, certains paramètres architecturaux ont été fixés à des valeurs arbitraires. Ces paramètres sont la topologie des réseaux, la taille des fi-

fos, ainsi que la hiérarchie mémoire. Ces variations n'ont pas été explorées pour des raisons de temps, et car elles dépassent un peu le cadre de notre étude, mais nous sommes conscients du fait que dans l'optique d'implanter un système TM sur une puce, ces points sont essentiels.

Nous avons choisi comme taille de fifos des valeurs qui nous semblaient réalistes au vu de nos hypothèses. La topologie des réseaux est fortement couplée à nos hypothèses, tandis que la hiérarchie mémoire aurait pu être un peu plus poussée, comme posséder des nœuds reliés entre eux par un réseau global, et possédant chacun quatre cœurs connectés sur un réseau local, sur lequel est aussi connecté un banc mémoire. Néanmoins, l'architecture choisie est réaliste même si elle reste simple.

8.2 Axes de recherche en vue de travaux futurs

Notre but dans cette thèse n'était pas de concevoir un prototype qui puisse être fondu, mais d'étudier les propriétés de ces systèmes à un niveau supérieur à celui de l'architecture. En ce sens, les travaux futurs qui en découlent sont de deux ordres : il faut d'une part poursuivre l'étude menée sur les mémoires transactionnelles et les pathologies d'un point de vue conceptuel, et d'autre part poursuivre l'étude à un niveau inférieur en vue d'une intégration, qui seule peut valider complètement l'approche.

8.2.1 Travaux futurs d'un point de vue conceptuel

8.2.1.1 Caractériser les systèmes en fonction des types d'applications

Si dans nos expérimentations, nous avons essayé de choisir un spectre assez large d'applications, dans l'espoir qu'il soit représentatif, il reste néanmoins très difficile de dire quelles sont les applications pour lesquelles les résultats sont les plus pertinents. Dans les moyennes données (ex : "Le système A est 1.25 fois plus lent que le système B"), nous avons considéré que tous les systèmes avaient la même importance, mais c'est une hypothèse pour faire face au manque de critères plus pertinents.

Bien qu'il soit très difficile de caractériser les applications et de les ranger en catégories, une mesure intéressante à notre sens est la taille et la fréquence des transactions dans un programme, ainsi que la fréquence des conflits (définissant la congestion du système). Ainsi, le travail viserait à définir des métriques telles que le pourcentage du temps passé dans une transaction pour un seul thread ou le nombre de lignes mémoire différentes accédées sur le nombre d'accès total. Cela est difficile car il faut que les métriques trouvées soient suffisamment robustes pour mener à des résultats qui aient du sens. S'il n'est pas sûr que ces métriques existent, nous pensons qu'il est possible d'en trouver qui caractérisent la majorité des systèmes. La validation de ces métriques passerait par la production (manuelle et automatique) d'une grande quantité de programmes différents ayant des métriques identiques. De bonnes métriques seraient dans ce cas caractérisés par l'obtention de résultats similaires pour chacun des systèmes.

Ainsi, le but de ce travail serait de pouvoir par exemple conclure qu'en présence d'un programme ayant une métrique de congestion inférieure à une valeur donnée, un système est meilleur, tandis que dans le cas inverse, c'est un autre système qui est le plus performant.

En outre, il serait possible de caractériser les applications des benchmarks afin de voir celles qui sont similaires voire redondantes.

8.2.1.2 Poursuivre le travail sur les pathologies

S'il est vrai que nous avons attaché de l'importance uniquement aux pathologies de type étreinte active, le travail commencé dans [BMV+07] visant à définir les pathologies (y compris celles n'affectant que les performances) selon les types de systèmes TM a une grande valeur. Il nous parait important de le poursuivre en l'étendant à tous les types de systèmes TM existants.

8.2.1.3 Définir des critères de résistance

Dans la poursuite de ce qui a été vu au chapitre 7, il nous parait important de définir des critères de résistance des systèmes face aux pathologies. Nous avons défini deux critères concernant la quantité maximale (finie ou non) de transactions dans un système ainsi que le fait qu'une garantie soit probabiliste ou non, mais cela n'est pas suffisant.

Nous pensons qu'il est possible de définir des critères plus fins en fonction des pathologies n'affectant pas que la fonctionnalité mais aussi les performances (i.e. pas seulement le fait que des transactions vont commiter un jour), ou prenant en compte les limitations matérielles : en effet, nous n'avons jusqu'à présent pas considéré les cas de débordement des registres contenant les identifiants des transactions ou la valeur de backoff maximale.

8.2.1.4 Système intégrant plusieurs politiques de résolution

Avant d'avoir connaissance des travaux faits dans [SD09], nous avons conçu un système qui devait combiner les politiques EE et LL, en s'adaptant dynamiquement à l'application et en choisissant la politique la plus efficace. Nous avons implanté ce système dans le sens EE vers LL à l'aide d'un protocole basé sur un jeton pouvant prendre plusieurs valeurs, mais suite aux mauvais résultats du LL, nous n'avons pas poursuivi l'implantation dans le sens inverse. Le but initial était de garantir le commit de transactions de manière non probabiliste en passant du EE vers LL après un certain nombre d'abort de la part de tous les processeurs dans une transaction. De plus, l'espoir portait sur le fait que le système puisse s'adapter à la solution la plus efficace à un instant donné dans le programme.

Néanmoins, la réalisation de certains des travaux discutés au-dessus pourrait montrer que certains types de systèmes sont plus efficaces avec certaines applications, ou ont des garanties plus élevées à l'aide de politiques en moyenne plus lentes, ce qui pourrait montrer l'avantage de réaliser des systèmes s'adaptant de manière dynamique. Cette piste n'a encore jamais été explorée, et le système actuel le plus souple nécessite une configuration logicielle statique [SD09].

8.2.2 Travaux futurs en vue d'une intégration

8.2.2.1 Adaptation dynamique de la taille du log

Notre solution actuelle suppose que le log alloué avant la première transaction sera assez grand pour l'exécution de toutes les transactions. En cas de débordement, cela est pour l'instant détecté mais pas corrigé. Une solution est d'aborter, libérer la zone actuelle et allouer une zone plus grande, par exemple de taille double.

8.2.2.2 Support du système d'exploitation

Même si notre approche est basée sur une solution TM entièrement matérielle, il est nécessaire d'avoir un support minimal de la part du système d'exploitation afin d'avoir une

solution pérenne. Les points ciblés sont entre autres la gestion des débordements décrite au-dessus, ainsi que l'appel à une fonction après un abort, permettant de faire le backoff en logiciel plutôt qu'en matériel (ce qui permet par exemple de faire un appel à la fonction `rand` sans ajouter de matériel). Néanmoins, d'autres points sont aussi à prévoir, en particulier la définition claire d'une interface pour les transactions, indépendante du type de processeur, et l'enrichissement du paramétrage de la solution matérielle actuelle.

8.2.2.3 Support de la part du compilateur

Actuellement, le codage des instructions permettant de spécifier le début et la taille du log est fait à la main dans le script d'éditions de lien. À terme, il est nécessaire que la chaine d'outils utilisée prenne en compte les nouvelles instructions nécessaires.

8.2.2.4 Faire une étude de consommation

Dans les travaux présentés, nous ne nous sommes jamais intéressés au problème de la consommation. Or, cela est une contrainte en général élevée des systèmes embarqués. Une telle étude est donc nécessaire en vue d'une intégration. Nous n'en avons pas fait d'une part pour une raison de temps, et d'autre part car les outils utilisés (et notamment la bibliothèque SoCLib) ne fournissent pas de support pour une étude de la sorte. Nous gardons néanmoins à l'esprit que l'aspect consommation peut retarder pour un certain temps la venue des systèmes TM dans le domaine de l'embarqué.

8.2.2.5 Analyse du cout en surface de la solution

Afin de valider la solution présentée, il est nécessaire d'en faire une implantation à un niveau de description matériel, par exemple en VHDL, afin de permettre d'évaluer le cout en surface d'un tel système TM. En particulier, cela permettrait de comparer ces performances à celles d'un système sans support pour les transactions, mais de surface équivalente, en comblant la différence de surface avec de la mémoire cache.

Chapitre 9

Conclusion

Les travaux présentés dans cette thèse visent à fournir un support matériel et une interface matériel/logiciel, tous deux adaptés à la programmation parallèle, plus particulièrement du point de vue du cache et de la mémoire. Cela a été fait d'une part par l'intermédiaire de l'étude d'une bibliothèque de vol de travail d'un point de vue architectural et d'autre part par l'implantation de plusieurs systèmes de mémoire transactionnelle.

Nous avons soulevé un certain nombre de questions spécifiques à ces deux points. Nous les reprenons maintenant, en y répondant de manière au moins partielle à la lumière des travaux effectués dans le cadre de cette thèse.

Le vol de travail est-il un moyen d'écriture de programmes parallèles adapté pour le domaine de l'embarqué ?

Comme vu au chapitre 4, le paradigme du vol de travail permet, grâce à un ordonnanceur décentralisé, d'adapter dynamiquement la charge de travail sur les cœurs de calcul disponibles. La contrainte au niveau du programmeur est que le parallélisme doit être décrit de manière à correspondre à l'interface, mais la garantie en est une synchronisation correcte entre les threads. Ces avantages en font un moyen adapté à l'écriture de programmes parallèles, y compris dans le domaine de l'embarqué.

Nous avons présenté AWS, une bibliothèque de vol de travail que nous avons utilisée sur des systèmes embarqués multiprocesseurs. Les gains en performance de cette bibliothèque par rapport à une parallélisation statique peuvent s'avérer significatifs. En effet, le vol de travail permet de n'avoir qu'un surcout faible dans l'exécution des programmes parallèles par rapport à une parallélisation statique, tout en pouvant mener à de larges gains en performance liés à l'équilibrage dynamique de la charge de travail.

Le plus grand problème posé par l'application de ce modèle à l'embarqué concerne le respect de garanties liées au temps-réel, en particulier dans le cas des systèmes critiques, puisque le vol de travail rend très difficile voire impossible l'analyse statique et la preuve de respect de contraintes temporelles.

Néanmoins, nous pensons que son application au domaine de l'embarqué reste possible, notamment car un grand nombre d'applications embarquées ne sont pas critiques. Nous avons utilisé AWS dans la parallélisation de deux applications, démontrant ainsi que cette approche est viable en pratique, y compris pour l'embarqué.

Quelles sont les caractéristiques matérielles permettant d'avoir une exécution efficace d'un algorithme basé sur le vol de travail ?

Notre étude de l'utilisation d'un algorithme basé sur le vol de travail sur un système embarqué s'est concentrée sur les deux points architecturaux suivants : premièrement, sur l'utilisation de mémoires adressables de manière implicite (caches) ou explicite (par des DMAs) pour les calculs à effecteur localement par les nœuds ; deuxièmement, sur la distribution des structures propres aux nœuds.

Les résultats montrent qu'il est préférable pour un algorithme basé sur le vol de travail d'utiliser un adressage implicite à base de caches plutôt que des mémoires locales, même dans le cas où les données accédées sont connues à l'avance : en effet, les caches sont le meilleur choix, d'une part d'un point de vue performances, mais aussi d'un point de vue modèle de programmation puisque leur gestion est entièrement matérielle.

Est-il envisageable de concevoir un système HTM basé sur un protocole de cohérence à écriture simultanée dans une architecture à base de NoCs ? Pour quelles performances ?

Nous avons la conviction que le modèle de programmation TM sera utilisé dans le futur, quel que soit le domaine, du fait la facilité d'écriture des programmes à base de transactions, ainsi que des propriétés liées aux transactions. De fait, une question primordiale pour les systèmes embarqués concerne le choix du protocole de cohérence de cache, puisqu'un système HTM efficace est fortement lié à ce protocole.

Nos travaux du chapitre 6 ont montré qu'il était possible de concevoir un tel système dans une architecture à base de NoCs. Si le choix du protocole de cohérence dans un système TM fait donc partie de la phase de conception, nous pensons toutefois qu'un protocole à écriture différée est plus adapté, et ce pour les raisons suivantes.

Premièrement, les performances, bien que globalement similaires, ont l'air de se situer en peu en dessous en écriture simultanée. Deuxièmement, si les complexités totales des deux systèmes sont proches, le surcout pour supporter les transactions est bien plus réduit en écriture différée. Or, il est toujours préférable de choisir la solution qui nécessite le moins de modifications par rapport à un système connu ou existant, puisqu'elle implique moins de conception et moins de validation. Enfin, la solution basée sur l'écriture simultanée possède une limitation de taille puisqu'elle nécessite de n'avoir qu'un seul banc mémoire dans l'architecture, ce qui est une contrainte d'autant plus grande que le nombre de cœurs augmente. Nous n'argumentons pas qu'une solution en écriture simultanée supportant plusieurs bancs mémoire n'est pas possible, mais dans ce cas, le surcout relatif à la cohérence aurait des conséquences importantes en termes de cout matériel et de cout temporel.

Ainsi, notre réponse est qu'une telle implantation est possible, mais qu'on lui préférera une approche classique basée sur l'écriture différée, pas pour des raisons de performances, mais plutôt pour son cout et ses limitations.

Quelles politiques des systèmes HTM donnent les meilleurs résultats ? À quels couts matériels ?

La principale contribution de ces travaux, présentée dans le chapitre 7, a été d'implanter et de comparer différents systèmes HTM. Les résultats montrent qu'entre les systèmes EE et LL, ceux donnant les meilleurs résultats sont les systèmes de type EE avec backoff. Par ailleurs, ce sont aussi ces systèmes qui requièrent le moins de surcout matériel, car ils ne nécessitent notamment pas l'ajout d'un deuxième réseau pour la circulation d'un jeton, contrairement aux systèmes LL.

Une deuxième partie de notre étude a consisté à comparer trois systèmes EE ayant des politiques de résolution différentes : celle de base, une à base d'estampilles et une nouvelle

politique basée sur le nombre de transactions gagnées. La politique EE de base semble donner les meilleurs résultats. D'un point de vue cout d'implantation, c'est aussi la plus légère puisqu'elle ne nécessite pas l'ajout de mécanismes particuliers contrairement aux deux autres.

Quelles sont les garanties que l'on peut obtenir en termes de progression du système et d'absence de pathologies pour ces différentes politiques ?

Notre étude sur les mémoires transactionnelles a finalement défini deux critères de robustesse de systèmes TM : la garantie de terminaison en présence d'un nombre de transactions fini et la garantie de terminaison en présence d'un nombre de transactions potentiellement infini – dans le cas d'un système ayant un nombre de transactions non borné, ce critère se traduit par l'absence de famine. Ces deux critères sont à moduler selon que la garantie de terminaison soit probabiliste ou non.

Nous avons montré que les solutions actuellement proposées ne garantissaient pas la terminaison en présence d'un nombre potentiellement infini de transactions. Pour palier ce problème, nous avons défini une nouvelle politique garantissant de manière probabiliste la terminaison de l'application en présence d'un nombre potentiellement infini de transactions. Néanmoins, de nouveaux critères restent à définir.

Il est fort possible qu'une solution avec de la mémoire cache équivalente en surface à un système TM donné et utilisant des verrous soit plus efficace que ce système TM. Néanmoins, si nous n'avons pas la conviction que le modèle TM s'imposera grâce à ses performances – nos résultats étant moins optimistes que les travaux similaires publiés –, nous avons la conviction qu'il le sera de par sa simplicité d'utilisation. Comme évoqué en introduction, la complexité grandissante des programmes parallèles nécessite de nouvelles abstractions. Aussi faut-il que ces abstractions ne dégradent pas significativement les performances des solutions actuelles, d'où l'intérêt de trouver la solution la plus efficace. Mais à nos yeux, ce sont plutôt les garanties concernant l'absence de comportement pathologiques qui sont importantes, et nous pensons que ces pathologies constituent les seuls vrais obstacles au déploiement du modèle TM.

Chapitre 10

Publications

Les travaux réalisés au cours de cette thèse ont donné lieu à plusieurs publications, listées ici.

1. Q. Meunier and F. Pétrot, Lightweight Transactional Memory systems for NoCs based architectures : Design, implementation and comparison of two policies, in *Journal of Parallel and Distributed Computing (JPDC)*, 2010
2. Q. Meunier and F. Pétrot and J.-L. Roch, Hardware/Software Support for Adaptive Work-Stealing in On-Chip Multiprocessor, in *Journal of Systems Architecture (JSA)*, 2010
3. Q. Meunier and F. Pétrot, LightTM : une Mémoire Transactionnelle conçue pour les MPSoCs, in *SympA'09*, *Toulouse*
4. Q. Meunier and F. Pétrot, Lightweight Transactional Memory Systems for Large Scale Shared Memory MPSoCs, in *NEWCAS'09*, *Toulouse*

Annexe A

Validation des modèles TM

A.1 Introduction

La validation des modèles de simulation est un problème très complexe, et mériterait sans doute une place plus importante dans ce manuscrit qu'une simple annexe. Beaucoup de travaux de recherche portent sur la vérification des programmes et la validation, mais cela est resté un peu en marge de notre travail, bien que nous avons pu voir l'utilité, ou plutôt sentir le manque, d'outils adaptés à nos besoins. Dans le cadre de nos travaux, nous avons été confrontés au problème de la validation de composants matériels, qui est un niveau de difficulté au-dessus du logiciel standard : en effet, l'accumulation des couches (code des composants matériels, simulateur, linker-script (mapping mémoire), système d'exploitation, logiciel exécuté sur la plateforme) rendent la mise au point et la correction d'erreurs très difficile, car ces dernières peuvent survenir à tous les niveaux. Pour y faire face, nous avons cherché quelles étaient les méthodes traditionnelles de validation, mais la seule référence qui s'y rapporte est [SPC+02] qui au final apporte très peu. Par ailleurs, les moyens de vérification statiques sont impensables pour les tailles des machines d'état utilisées ; parallèlement à cela, il est difficile d'exprimer des propriétés sur ces machines d'états. Valider un fonctionnement souhaité ne peut donc se faire que par un test intensif.

Nous avons donc opté pour la validation par le test de nos modèles TM écrits. Si écrire des petits tests permet de vérifier la base du comportement sur quelques cas précis, arriver à écrire des tests couvrant un grand nombre de cas est long, car les tests comportant une boucle répétant un code identique ont tendance à converger vers un comportement identique au fil des boucles (dans l'entrelacement des requêtes entre les processeurs). C'est pourquoi, il nous a paru nécessaire de générer ces tests automatiquement.

Par ailleurs, nous avons la chance de pouvoir obtenir un résultat de référence pour ces tests, en lançant une simulation avec des spin locks au lieu des transactions sur les sections critiques.

A.2 Raisonnement et logique de l'approche

Pour pouvoir générer des tests couvrant un maximum de cas, c'est-à-dire efficaces, il est nécessaire d'avoir connaissance de l'architecture du système. Autrement, les tests générés risquent de tester pour beaucoup la même portion de code des composants et les mêmes parties du protocole.

La granularité des transactions pour les différents systèmes étant la ligne de cache, cela en fait un paramètre primordial pour les tests générés. Comme les débordements de cache

sont eux aussi des points critiques pour le système, on distingue au final deux paramètres principaux : le nombre de lignes mémoires contenant les variables qui seront accédées durant la transaction (nb_diff_ML), et le nombre de ligne de cache dans lequel seront placées ces lignes mémoires (nb_diff_CL). Cela permet de définir la contention au niveau du système : si nb_diff_ML = nb_diff_CL, il n'y aura jamais de débordement dans une transaction ; à l'inverse, si nb_diff_CL = 1, toutes les lignes mémoires seront placées dans la même ligne de cache, résultant en de nombreux débordements. Bien entendu, le nombre de lignes de cache ainsi que leur taille doivent être connus pour pouvoir générer les programmes finaux. La stratégie de placement est illustrée figure A.1.

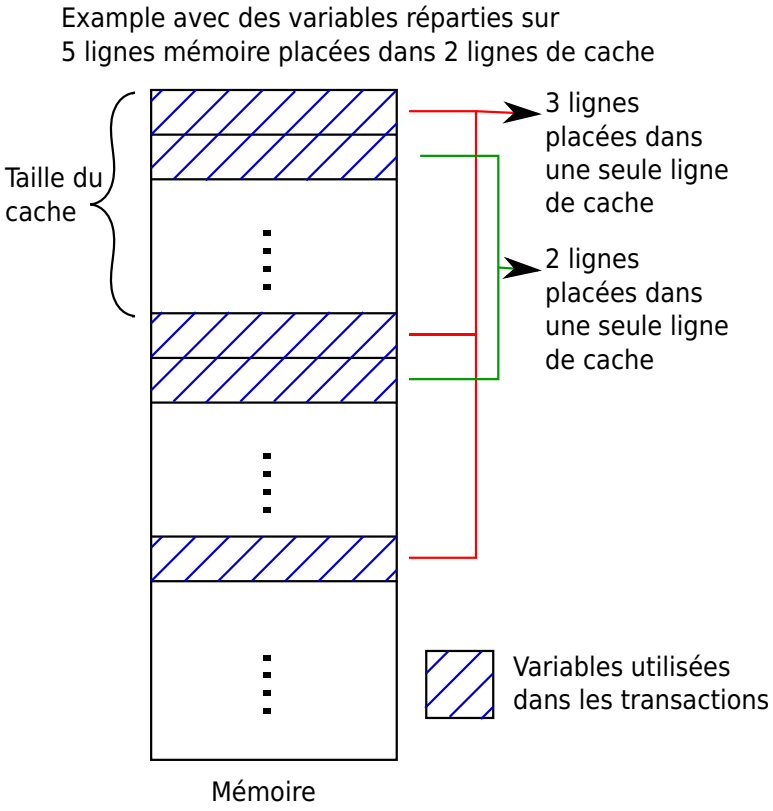


FIG. A.1 – Stratégie de placement des données en fonction du nombre de lignes mémoire et de lignes de cache accédées

Un autre point important pour la validation des systèmes sont le type des requêtes : en effet, si en général une transaction commence par lire puis écrit une variable, il arrive qu'elle ne fasse que des écritures ou que des lectures sur certaines variables, ce qui change la partie du protocole utilisée (ainsi, un bogue rencontré dans LightTM-LL ne se manifestait que lorsqu'une transaction ne faisait que des lectures). Pour cette raison, les variables accédées possèdent un attribut spécifiant le type des requêtes pouvant être effectuées dessus (RW, RO, WO).

Un des points les plus critiques dans la conception des tous les systèmes sont les interactions entre les accès effectués sur une même ligne mémoire par une transaction d'une part, et par une requête standard d'autre part. Pour générer ce type de conflit, les variables peuvent être privée à un thread, et ainsi être accédées en dehors des transactions, tandis que des accès transactionnels se font de manière concurrente sur les autres variables de la ligne.

Les proportions de variables privées, Read-only et Write-only ont été fixé arbitrairement

à une par ligne en moyenne pour les tests générés.

Un des derniers points concerne les accès non cachés au milieu des transactions. En effet, si les accès non cachés outrepassent le mécanisme de transactions, il peut se produire de subtiles interactions lorsqu'un accès non caché est fait au sein d'une transaction. La granularité des accès étant moins cruciale dans ce cas, une seule ligne a été utilisée en pratique pour les variables non cachées. Par ailleurs, pour pouvoir garantir un résultat déterministe, les accès sur les variables non cachées doivent être soit privés, soit toujours RO ou WO.

Enfin, les autres paramètres utiles pour la génération des tests sont le nombre maximum de transactions par processeur (thread) et le nombre maximum d'instructions par transaction.

A.3 Implémentation

Le programme générant les programmes de tests a été implémenté en C++. La figure A.2 montre un diagramme de classes de l'outil.

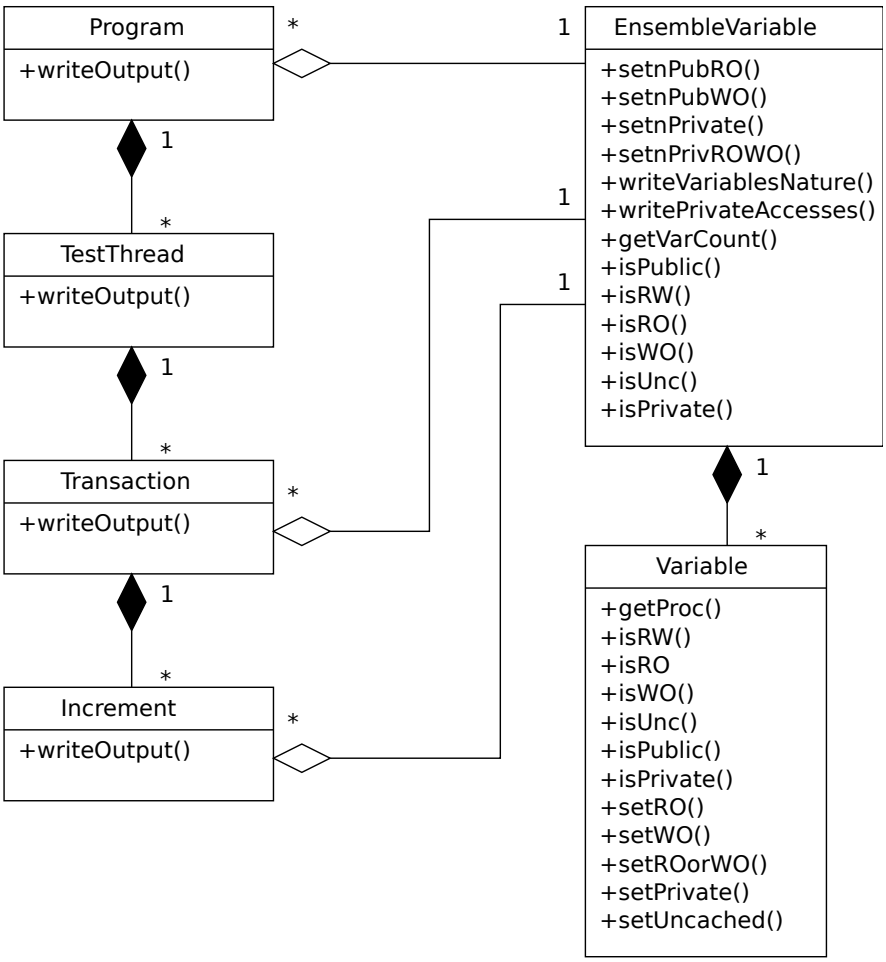


FIG. A.2 – Diagramme de classes de l'outil de génération de tests

L'exemple de code ci-dessous montre une portion d'un petit programme avec la fonction exécutée par un des threads.

```
1 #include "local_defs.h"
```

```
2
3  #define NB_THREADS 2
4
5  /* NB_MAX_TRANS : 4
6   * NB_MAX_INSTS : 8
7   * LINE_SIZE : 4
8   * CACHE_LINES : 512
9   */
10
11 volatile int tab[4104];
12
13 /******************
14 * Variables Nature *
15 ******************/
16 /* [    0]    0 WO     [    1] Pub WO     [    2] Pub RO     [    3] Pub RO
17  * [    4]    0 RW     [    5] Pub WO     [    6] Pub WO     [    7]    1 WO
18  * [2048] Pub RW     [2049] Pub RO     [2050]    1 RW     [2051] Pub RW
19  * [2052] Pub RO     [2053] Pub RW     [2054] Pub RW     [2055] Pub RW
20  * [4096] Pub WO (U) [4097] Pub WO (U) [4098] Pub WO (U) [4099] Pub WO(U)
21  */
22
23
24 #ifdef USE_SPIN
25    pthread_spinlock_t *spin;
26 #endif
27
28 void run0(){
29    int local_var; // variable pour pouvoir avoir des transactions où l'on
         ne fait que lire les variables utiles
30
31    #ifdef USE_TRANS
32       int size;
33       int *mem;
34       size = sizeof(unsigned int)*9*100;
35       mem = local_malloc(size);
36       store_log_size(size);
37       store_log_address(mem);
38    #endif
39
40    tab[0] = 1; // variable privée au proc 0 et en WO
41    tab[4]++; // variable privée au proc 0
42
43    #ifdef USE_TRANS
44       begin_transaction();
45    #endif
46    #ifdef USE_SPIN
47       pthread_spin_lock(spin);
48    #endif
49    local_var = tab[3];
50    __asm__ __volatile__ (
51          "clr %%l6\n"
52          "\tsethi 0x4, %%l6\n"
53          "\tor %%l6, 0x3, %%l6\n"
54          "\tsll %%l6, 2, %%l6\n"
55          "\tadd %0, %%l6, %%l4\n"
56          "\tclr %%l5\n"
```

```
57          "\tor %%l5, 0x1, %%l5\n"
58          "\tsta %%l5, [ %%l4 ] (32)\n"
59       :
60       : "r" (tab)
61       : "%l4","%l5", "%l6"
62    );
63    tab[2055]++;
64    tab[2055]++;
65    local_var = tab[2052];
66    #ifdef USE_TRANS
67       end_transaction();
68    #endif
69    #ifdef USE_SPIN
70       pthread_spin_unlock(spin);
71    #endif
72
73    tab[0] = 1; // variable privée au proc 0 et en WO
74    tab[4]++; // variable privée au proc 0
75
76    #ifdef USE_TRANS
77       begin_transaction();
78    #endif
79    #ifdef USE_SPIN
80       pthread_spin_lock(spin);
81    #endif
82    local_var = tab[2052];
83    __asm__ __volatile__ (
84          "clr %%l6\n"
85          "\tsethi 0x4, %%l6\n"
86          "\tor %%l6, 0x2, %%l6\n"
87          "\tsll %%l6, 2, %%l6\n"
88          "\tadd %0, %%l6, %%l4\n"
89          "\tclr %%l5\n"
90          "\tor %%l5, 0x1, %%l5\n"
91          "\tsta %%l5, [ %%l4 ] (32)\n"
92       :
93       : "r" (tab)
94       : "%l4","%l5", "%l6"
95    );
96    tab[0] = 1; // variable privée au proc 0 et en WO
97    tab[5] = 1;
98    local_var = tab[2052];
99    #ifdef USE_TRANS
100      end_transaction();
101   #endif
102   #ifdef USE_SPIN
103      pthread_spin_unlock(spin);
104   #endif
105
106 }
```

A.4 Résultats

Bien qu'il soit difficile de décrire les résultats à l'aide de métriques, cet outil combiné au script de lancement et de vérification, nous a permis de corriger un grand nombre de bogues dans nos implémentations. La figure A.3 montre comment procède le script de vérification pour valider un système.

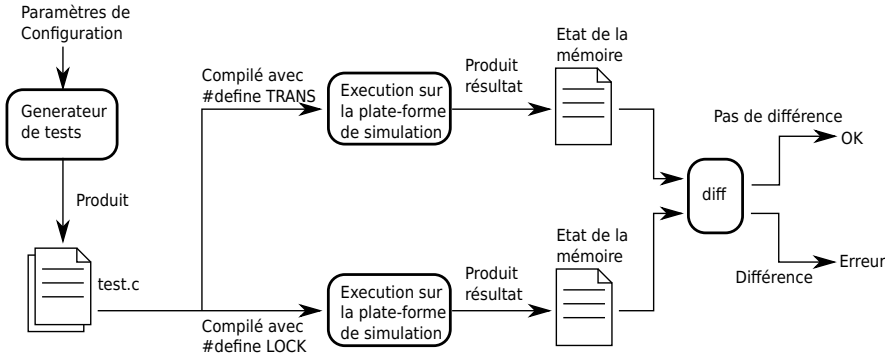


FIG. A.3 – Schéma représentant le mécanisme de validation automatique à partir des tests générés

L'avantage de ces programmes de tests sont qu'une fois un bogue mis en évidence, il est assez facile de le tracer jusqu'à découvrir sa source. Il suffit pour cela de repérer tous les accès à la variable et de regarder les valeurs de la variable après les commits des transactions y faisant accès. Insérer les points d'arrêt correspondants peut se faire à l'aide d'un script (perl par exemple), rendant la tâche beaucoup plus facile que le débogage d'une application réelle.

Nous avions l'intention de vérifier l'efficacité de ce générateur de tests en mesurant la couverture du code des composants. En particulier, le but était de savoir si en un temps raisonnable, on obtenait une couverture complète selon le critère le plus faible (chaque ligne est exécutée au moins une fois). Malheureusement, nous n'avons pas réussi à intégrer correctement gprof dans les modèles SoCLib.

Annexe B

Macros utilisées pour réaliser les instructions CAS

Les instructions CAS simple et double mot n'ont été utilisées que sur des architectures de type x86. Ces architectures possèdent une instruction `cmpchg` pour "compare and exchange", mais qui n'est pas atomique. Il est néanmoins possible de la rendre atomique en préfixant l'instruction du mot-clé `lock`, donnant ainsi une instruction ayant la sémantique du CAS. Pour la version double mot, il existe l'instruction `cmpxchg8b` [1].

La réalisation des macros `cas` et `cas2` sont données respectivement figures B.1 et B.2.

```
#define cas(_a, _o, _n)                                          \
   ({ __typeof__(_o) __o = _o;                                   \
     __asm__ __volatile__(                                       \
          "lock cmpxchg %3,%1"                                   \
          : "=a" (__o), "=m" (*(volatile unsigned int *)(_a)) \
          :  "0" (__o), "r" (_n) );                              \
     __o;                                                        \
   })
```

FIG. B.1 – Macro utilisée pour réaliser l'instruction CAS simple mot

```
#define cas2(ptr,old_value,new_value)                    \
   ({ __typeof__ (*ptr) prev;                            \
     __asm__ __volatile__ ("movl (%3), %%ebx\n\t"        \
                   "movl 4(%3), %%ecx\n\t"               \
                   "lock; cmpxchg8b %1\n\t"              \
                   : "=A" (prev)                         \
                   : "m" (*ptr),                         \
                     "0" (old_value),                    \
                     "r" (&new_value)                    \
                   : "memory", "%ebx", "%ecx");          \
     prev;                                               \
   })
```

FIG. B.2 – Macro utilisée pour réaliser l'instruction CAS double mot

[1] Note : contrairement à la définition du CAS donnée au chapitre 2, ces macros retournent en cas de succès l'ancienne valeur, et non `1`

Annexe C

Allocations dans les transactions

C.1 Pourquoi cela pose un problème

`malloc` est une primitive du système qui utilise une section critique à base de verrous afin de garantir qu'une zone mémoire donnée n'est allouée qu'une seule fois, même lors de l'exécution concurrente d'appels à cette primitive. Si l'on s'autorise à faire des appels à `malloc` dans une transaction sans rien modifier, cela va créer une étreinte mortelle dès qu'un processeur ayant obtenu un verrou dans `malloc` doit faire un abort, puisque lors de dernier, le verrou ne sera pas relâché, menant toutes les tentatives de prise suivantes à un échec.

C.2 Première approche : cacher les accès aux verrous dans les transactions

Cette solution, à la base de l'étude menée dans [HVS08], consiste à considérer les accès vers les verrous comme des accès cachés de manière cohérente vers la mémoire. Lorsque l'on dispose de verrous matériels, cela ne présente que peu d'intérêt, puisque les verrous sont des ressources hautement partagées, et rajoute donc le surcout matériel de la cohérence pour peu ou pas de gains en performance. Adapter notre architecture en vue d'une solution basée sur cette approche pour supporter les allocations à l'intérieur des transactions peut néanmoins se faire sans utiliser la cohérence. L'idée est de différencier les types d'accès non cachés, en fonction de leur nature : lié à une instruction ou à un segment (une requête est propagée du cache à la mémoire si l'instruction spécifie qu'il s'agit d'un accès non caché ou si le segment correspondant à l'adresse de la requête n'est pas caché). Ainsi, on peut faire en sorte dans les transactions de ne considérer que l'attribut de cachabilité d'un segment et non le type de la requête, de manière à cacher les requêtes vers le segment contenant les verrous (qui aurait alors l'attribut caché, mais qui toutes les requêtes seraient propagées en dehors des transactions, du fait de la nature des instructions).

Néanmoins, cette solution est un peu complexe et nécessite de modifier la sémantique de cachabilité, avec le risque de ne plus faire marcher certains codes. C'est pour cette raison que nous ne l'avons pas retenue.

C.3 Deuxième approche : ajouter une primitive d'allocation à base de transactions

Cette approche consiste à ajouter une primitive d'allocation qui contient une section critique définie par une transaction. De cette manière, la ou les variables modifiées par l'allocation retrouveront leur valeur initiale lors d'un abort, ne pouvant ainsi pas mener à une fuite mémoire (zone mémoire non libérée dans le processus). Néanmoins, cette solution pose un gros problème puisque toutes les transactions ayant fait une allocation à un instant donné se retrouvent avec la même zone mémoire allouée, ce qui mène à une baisse de performance terrible et résulte en une multitude d'aborts.

Il est donc nécessaire pour rendre viable cette solution de répartir les allocations sur différents bancs mémoire. La solution employée consiste à faire appel à une primitive d'allocation, dont le banc mémoire dépend du processeur.

Si cela permet de réduire grandement la congestion du système, il subsiste encore un problème, qui est que toutes les adresses de début de ces différentes sections sont par défaut contigües en mémoire. Nous avons placé ces données de manière à n'avoir qu'une adresse de section par ligne, menant ainsi à un gain de 50% en performance.

Enfin, un dernier problème concerne la concurrence entre les appels à la primitive `malloc` standard, et les appels à la primitive `malloc` pour les transactions : il est en effet nécessaire de garantir que les deux n'interviennent pas de manière simultanée, cela pouvant autrement mener à de nombreux problèmes [1]. Il faut donc ajouter des points de synchronisation dès que nécessaire (cela est déjà le cas dans les benchmarks STAMP). Nous avons donc utilisé cette solution pour pouvoir simuler les benchmarks STAMP.

[1] Par exemple si malloc standard lit la valeur de base de la zone allouée, et avant de la modifier reçoit une requête d'invalidation sur la ligne à cause d'une transaction qui fait une allocation au même instant. La transaction va réussir, et à sa fin, l'allocation standard pourra reprendre, allouant la même zone mémoire que la transaction.

Bibliographie

[AAK+05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proc. 11th International Conference on High-Performance Computer Architecture (11th HPCA'05)*, pages 316–327, San Francisco, CA, USA, February 2005. IEEE Computer Society. 5.5.1

[ABP01] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2) :115–144, 2001. 2.4.2, 3.1.3

[AKS06] Adnan Agbaria, Dong-In Kang, and Karandeep Singh. Lmpi : Mpi for heterogeneous embedded distributed systems. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 79–86, Minneapolis, MN, July 2006. IEEE. 2.4.1

[ALHH08] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen-Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems*, 26(3), 2008. 3.1

[BDLM07] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proc. 34th International Symposium on Computer Architecture (34th ISCA'07)*, pages 24–34, San Diego, California, USA, June 2007. ACM SIGARCH. 5.5.3

[BGH+08] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. TokenTM : Efficient execution of large transactions with hardware transactional memory. In *Proc. 35th International Symposium on Computer Architecture (35th ISCA'08)*, Beijing, June 2008. ACM SIGARCH. 5.5.3

[BHHR] Jayaram Bobba, Mark Hill, Tim Harris, and Ravi Rajwar. Transactional memory bibliography. 5.5

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Santa Fe, New Mexico., November 1994. 3.1.2

[BLM06] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted transactional memory : Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr 2006. 5.4.4.1

[Blo70] B. H. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Comm. of the ACM*, 13(7) :422, July 1970. 5.4.2

[BMV+07] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware

transactional memory. In *Proc. 34th International Symposium on Computer Architecture (34th ISCA'07)*, pages 81–91, San Diego, California, USA, June 2007. ACM SIGARCH. 5.3.2, 7.1, 7.6.1, 8.2.1.2

[BNZ08] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. *ACM SIGARCH Computer Architecture News*, 36(3) :115–126, 2008. 2.6

[BR02] Michael A. Bender and Michael O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory of Computing Systems*, 35 :2002, 2002. 2.4.2, 3.1.3, 4.4.4

[BRPS06] Julien Bernard, Jean-Louis Roch, Serge De Paoli, and Miguel Santana. Adaptive encoding of multimedia streams on MPSoC. In *International Conference on Computational Science (4)*, volume 3994 of *Lecture Notes in Computer Science*, pages 999–1006. Springer, 2006. 3.4.1

[BRT08] Julien Bernard, Jean-Louis Roch, and Daouda Traoré. Processor-oblivious parallel stream computations. In *PDP*, pages 72–76. IEEE Computer Society, 2008. 3.2.1, 3.3.3.5

[BvLR+08] Holger Blume, Jörg von Livonius, Lisa Rotenberg, Tobias G. Noll, Harald Bothe, and Jörg Brakensiek. Openmp-based parallelization on an mpcore multiprocessor platform - a performance and power analysis. *Journal of Systems Architecture - Embedded Systems Design*, 54(11) :1019–1029, 2008. 2.4.1

[BZ07] Lee Baugh and Craig Zilles. Analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *TRANSACT '07 : 2nd Workshop on Transactional Computing*, aug 2007. 5.4.4.1

[CBM+08] Calin Cascaval, Colin Blundell, Maged M. Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory : Why is it only a research toy ? *ACM Queue*, 6(5) :46–58, 2008. 2.6.1

[CCM+06] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in transactional memory virtualization. In *ASPLOS-XII : Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM Press, Oct 2006. 5.5.2

[CGK+07] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liakovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilferson. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115, San Diego, June 2007. ACM Press. 3.4.1

[CLP+08] Olivier Certner, Zheng Li, Pierre Palatin, Olivier Temam, Frederic Arzel, and Nathalie Drach. A practical approach for reconciling high and predictable performance in non-regular parallel programs. In *Proceedings of the conference on Design, automation and test in Europe*, pages 740–745, Munich, Germany, March 2008. 3.4.1

[CNV+06] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *ASPLOS-XII : Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 347–358. ACM, 2006. 5.5.2

[CRM07] Michael Chu, Rajiv Ravindran, and Scott Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 369–380, 2007. 3.4.1

[CTTC06] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA*, pages 227–238. IEEE Computer Society, 2006. 5.4.2, 5.5.2

[DGG+07] Vincent Danjean, Roland Gillard, Serge Guelton, Jean-Louis Roch, and Thomas Roche. Adaptive loops with kaapi on multicore and grid : applications in symmetric cryptography. In Marc Moreno Maza and Stephen M. Watt, editors, *Parallel Symbolic Computation, PASCO 2007, International Workshop, 27-28 July 2007, University of Western Ontario, London, Ontario, Canada*, pages 33–42. ACM, 2007. 3.2.1

[DGK+05] El-Mostafa Daoudi, Thierry Gautier, Aicha Kerfali, Rémi Revire, and Jean-Louis Roch. Algorithmes parallèles à grain adaptatif et applications. *Technique et Science Informatiques*, 24 :1–20, 2005. 3.2.1

[DLM+10] David Dice, Yossi Lev, Virendra J. Marathe, Mark Moir, Daniel Nussbaum, and Marek Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the 22nd Symposium on Parallelism in Algorithms and Architectures (22nd SPAA'10)*, Thira, Santorini, Greece, June 2010. ACM. 5.5

[DLMN09] David Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (14th ASPLOS'09)*, pages 157–168, Washington, DC, USA, March 2009. ACM. 5.5

[dMP08] Pierre Guironnet de Massas and Frédéric Pétrot. Comparison of memory write policies for noC based multicore cache coherent systems. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'08)*, pages 997–1002, Munich, Germany, March 2008. IEEE. 2.6.2, 4.4.2

[DTP+05] Andrew Duller, Daniel Towner, Gajinder Panesar, Alan Gray, and Will Robbins. picoarray technology : The tool's story. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 106–111, Munich, Germany, March 2005. 1

[Dur06] Marc Duranton. The challenges for high performance embedded systems. In *Proceedings of the 9th Euromicro Conference on Digital System Design.*, pages 3–7, Dubrovnik, Croatia, September 2006. Keynote address. 2.4.1

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1) :23–53, 1991. 3.3.3.5

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2) :5 :1–5 : ? ?, May 2007. 2.6, 2.6.1

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Programming Language Design and Implementation*, pages 212–223, 1998. 2.4.2, 3.1, 3.1.3, 3.4.1

[FMB+07] Cesare Ferri, Tali Moreshet, R. Iris Bahar, Luca Benini, and Maurice Herlihy. A hardware/software framework for supporting transactional memory in a

MPSoC environment. *SIGARCH Computer Architecture News*, 35(1) :47–54, 2007. 5.7

[GAC09] J. Ruben Titos Gil, Manuel E. Acacio, and Jose M. Garcia Carrasco. Speculation-based conflict resolution in hardware transactional memory. In *Proc. 23rd IEEE International Symposium on Parallel and Distributed Processing (23rd IPDPS'09)*, pages 1–12, Rome, Italy, May 2009. IEEE Computer Society. 7.2.2

[GAG08] J. Ruben Titos Gil, Manuel E. Acacio, and Jose M. Garcia. Directory-based conflict detection in hardware transactional memory. In P. Sadayappan, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *High Performance Computing – (15th HiPC'08), Proceedings 15th International Conference*, volume 5374 of *Lecture Notes in Computer Science (LNCS)*, pages 541–554, Bangalore, India, December 2008. Springer-Verlag (New York). 5.7

[GCPB99] Michael Barry Greenwald, David R. Cheriton, Serge Plotkin, and Mary Baker. Non-blocking synchronization and system design, August 17 1999. 2.5.3.2

[GG00] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In *DATE '00 : Proceedings of the conference on Design, automation and test in Europe*, pages 250–256, New York, NY, USA, 2000. ACM. 2.1.1.3

[Ghe05] Frank Ghenassia, editor. *Transaction Level Modeling with SystemC : TLM Concepts and Applications for Embedded Systems*. Springer, 2005. 3.4.1

[GK09a] Rachid Guerraoui and Michal Kapalka. How Live Can a Transactional Memory Be ? Technical report, 2009. 7.6.1

[GK09b] Rachid Guerraoui and Michal Kapalka. The Semantics of Progress in Lock-Based Transactional Memory. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2009. 7.6.1

[HLMS03] Herlihy, Luchangco, Moir, and Scherer. Software transactional memory for dynamic-sized data structures. In *PODC : 22th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2003. 2.6

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory : Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993. 2.6, 5.5

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7) :33–38, 2008. 2.3.3

[HVS08] Neelam Goyal Haris Volos and Michael M. Swift. Pathological interaction of locks with transactional memory, December 2008. 5.4.4.2, C.2

[HWC+04] Lance Hammond, Vicky Wong, Michael K. Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA*, pages 102–113. IEEE Computer Society, 2004. 2.6, 5.5.1

[HYUY09] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based load balancing. *SIGPLAN Not.*, 44(4) :55–64, 2009. 3.1.3

[Int06] Intel. Intel pentium 4 processor on 90nm process specification update, September 2006. 1

[ITR09] ITRS. International technology roadmap for semiconductors. In *System Drivers*, 2009. 1

[LAS+07] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing memory systems for chip multiprocessors. In *34th International Symposium on Computer Architecture*, pages 358–368, San Diego, California, June 2007. ACM. 3.4.1, 4.5.5

[LMG09] Marc Lupon, Grigorios Magklis, and Antonio Gonzalez. Fastm : A log-based hardware transactional memory with fast abort recovery. In *PACT '09 : Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 293–302, Washington, DC, USA, 2009. IEEE Computer Society. 5.5.3

[Lom77] David B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Language Design for Reliable Software*, pages 128–137, 1977. 2.5.4

[LZL+08] Yi Liu, Xin Zhang, He Li, Mingxiu Li, and Depei Qian. Hardware transactional memory supporting I/O operations within transactions. In *HPCC '08 : Proc. 10th International Conference on High Performance Computing and Communications*, pages 85–92, sep 2008. 5.4.4.1

[MBM+06a] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM : log-based transactional memory. In *HPCA*, pages 254–265. IEEE Computer Society, 2006. 2.6, 5.5.2, 6.6.3, 7.3.2

[MBM+06b] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logTM. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 359–370. ACM, 2006. 5.4.3, 5.4.3.3

[MCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP : Stanford transactional applications for multi-processing. In David Christie, Alan Lee, Onur Mutlu, and Benjamin G. Zorn, editors, *IISWC*, pages 35–46. IEEE, 2008. 7.4

[Mic04] Maged M. Michael. Hazard pointers : Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, PDS-15(6) :491–504, June 2004. 4.10

[MKH91] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation : A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3) :264–280, July 1991. 2.4.2

[MSB+05] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4) :99, 2005. 5.6

[MSS05] Marathe, Scherer, and Scott. Adaptive software transactional memory. In *DISC : International Symposium on Distributed Computing*. LNCS, 2005. 2.6

[MTC+07] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proc. 34th International Symposium on Computer Architecture (34th ISCA'07)*, pages 69–80, San Diego, California, USA, June 2007. ACM SIGARCH. 2.6

[OKK03] Jaegeun Oh, Seon Wook Kim, and Chulwoo Kim. Openmp and compilation issue in embedded applications. In *Proceedings of the International Workshop on OpenMP Applications and Tools*, volume 2716 of *Lecture Notes in Computer Science*, pages 109–121, Toronto, Canada, June 2003. Springer. 2.4.1

[Pap98] Dionysios P. Papadopoulos. *Hood : A User-Level Thread Library for Multiprogramming Multiprocessors*. PhD thesis, The University of Texas at Austin, September 21 1998. 2.4.2

[PDN97] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European conference on Design and Test*, pages 7–11, Paris, France, March 1997. 4.2.1

[PG03] Frédéric Pétrot and Pascal Gomez. Lightweight implementation of the POSIX threads API for an on-chip MIPS multiprocessor with VCI interconnect. In *Proc. of the Design Automation and Test in Europe, Embedded Software Forum*, pages 20051–20056, Munich, Germany, March 2003. IEEE Computer Society. 4.2.2, 6.5.3

[PPB02] Pierre Paulin, Chuck Pilkington, and Essaid Bensoudane. Stepnp : A system-level exploration platform for network processors. *IEEE Design & Test of Computers*, 19(6) :17–26, 2002. 2.4.1

[RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. *ACM SIGPLAN Notices*, 37(10) :5–17, October 2002. 7.1, 7.3.2, 7.6.1

[RHL05] Ravi Rajwar, Maurice Herlihy, and Konrad K. Lai. Virtualizing transactional memory. In *ISCA*, pages 494–505. IEEE Computer Society, 2005. 2.6, 5.5.1

[SD09] Arrvindh Shriraman and Sandhya Dwarkadas. Refereeing conflicts in hardware transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing (23rd ICS'09)*, pages 136–146, Yorktown Heights, NY, USA, June 2009. ACM Press. U. Rochester. 7.1, 7.3, 7.4.1.1, 7.6.1, 7.6.2.5, 8.2.1.4

[SDS08] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008. 5.5.3

[SGMP08] A. Sheibanyrad, A. Greiner, and I. Miro-Panades. Multisynchronous and fully asynchronous nocs for gals architectures. *IEEE Design & Test of Computers*, 25(6) :572–580, Nov.-Dec. 2008. 4.2.1

[SM01] B. Saglam and V. Mooney, III. System-on-a-chip processor synchronization support in hardware. In *Proceedings of the conference on Design, automation and test in Europe*, pages 633–641, Munich, Germany, March 2001. IEEE. 4.2.1

[SPC+02] Daniel J. Sorin, Manoj Plakal, Anne Condon, Mark D. Hill, Milo M. K. Martin, and David A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Trans. Parallel Distrib. Syst*, 13(6) :556–578, 2002. A.1

[SSH+07] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*. Jun 2007. 5.5.3

[ST97] Shavit and Touitou. Software transactional memory. *DISTCOMP : Distributed Computing*, 10, 1997. 2.6

[SVG+08] M.M. Swift, H. Volos, N. Goyal, L. Yen, M.D. Hill, and D.A. Wood. Os support for virtualizing hardware transactional memory, 2008. 5.5.3

[SYHS07] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *MICRO*, pages 123–133. IEEE Computer Society, 2007. 5.4.2

[The08] The Soclib Consortium. Soclib : an open platform for virtual prototyping of multi-processors system on chip. Technical report, [Online]. Available : http ://www.soclib.fr, 2008. 4.5, 6.6.1

[TPK+09] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM : eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 145–155. ACM, 2009. 7.2.2

[TRM+08] Daouda Traoré, Jean-Louis Roch, Nicolas Maillard, Thierry Gautier, and Julien Bernard. Deque-free work-optimal parallel stl algorithms. In *Euro-Par 2008 Parallel Processing*, Lecture Notes in Computer Science, pages 887–897, 2008. 2.4.2, 3.1.3, 3.2.1, 3.3.1

[VHC+08] E. Vallejo, T. Harris, A. Cristal, O. Unsal, and M. Valero. Hybrid transactional memory to accelerate safe lock-based transactions. In *Workshop on Transactional Computing (TRANSACT)*. Citeseer, 2008. 2.6

[WGH+07] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5) :15–31, 2007. 1

[WNySS96] Haigeng Wang, Alexandru Nicolau, and Kai yeung S. Siu. The strict time lower bound and optimal schedules for parallel prefix with resource constraints. *IEEE Trans. Comput*, 45 :1257–1271, 1996. 3.2

[WOT+95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs : Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press. 6.6

[WS07] M. M. Waliullah and Per Stenstrom. Starvation-free transactional memory system protocols. In *Proceedings of the 13th Euro-Par Conference : European Conference on Parallel and Distributed Computing*, pages 280–291. Aug 2007. 7.6.1

[YBM+07] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE : Decoupling hardware transactional memory from caches. In *HPCA*, pages 261–272. IEEE Computer Society, 2007. 5.5.3

Résumé L'avènement des puces multicoeurs repose certaines questions quant aux moyens d'écrire les programmes, qui doivent alors intégrer un degré élevé de parallélisme. Nous abordons cette question par l'intermédiaire de deux points de vue orthogonaux. Premièrement via le paradigme du vol de travail, pour lequel nous effectuons une étude visant d'une part à rechercher quelles sont les caractéristiques architecturales simples donnant les meilleures performances pour une implémentation de ce paradigme ; et d'autre part à montrer que le surcout par rapport à une parallélisation statique est faible tout en permettant des gains en performances grâce à l'équilibrage dynamique des charges. Cette question est néanmoins surtout abordée via le paradigme de programmation à base de transactions – ensemble d'instructions s'exécutant de manière atomique du point de vue des autres coeurs. Supporter cette abstraction nécessite l'implantation d'un système dit TM, souvent complexe, pouvant être logiciel ou matériel. L'étude porte premièrement sur la comparaison de systèmes TM matériels basés sur des choix architecturaux différents (protocole de cohérence de cache), puis sur l'impact d'un point de vue performances de plusieurs politiques de résolution des conflits, autrement dit des actions à prendre quand deux transactions essaient d'accéder simultanément les mêmes données.

Mots-Clés Mémoire transactionnelle, Matériel, Réseau-sur-puce, Protocole de cohérence, Simulation cycle-accurate, Performances

---

Abstract The arrival of multiprocessor chips rises again some questions about the way of writing programs, which must then include a high degree of parallelism. We tackle this problem via two orthogonal approaches. First, via the work-stealing paradigm, for which we perform a study targeting on the first hand to seek for simple architectural characteristics giving the best performances for an implementation of this paradigm ; and on the second hand to show that the overhead compared to a static parallelization is low, while allowing performances improvement thanks to dynamic load balancing. This question is nevertheless especially tackled via the transaction based programming paradigm – sequence of instructions executing atomically from the other cores' point of view. Supporting this abstraction requires the implementation of a system called TM, often complex, either software or hardware. The study focuses first on the comparison between two hardware TM systems based on different architecture choices (cache coherence protocol), and then on the impact on performances of several conflict resolution policies, in other words the actions to be taken when two or more transactions try to access the same pieces of data.

Keywords Transactional Memory, Hardware, Network-on-Chip, Coherence Protocol, Cycle-accurate Simulation, Performances

ISBN : 978-2-84813-159-7
Laboratoire TIMA – 46 Avenue Félix Viallet, 38000 Grenoble