Université de Montréal

Optimisation mémoire et exploration architecturale d'applications multimédias sur un réseau sur puce

par

Vincent Gagné

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de M.Sc.
en informatique

mars 2006

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Optimisation mémoire et exploration architecturale d'applications multimédias sur un réseau sur puce

présenté par :

Vincent Gagné

a été évalué par un jury composé des personnes suivantes :

Stefan Monnier
président-rapporteur

El Mostapha Aboulhamid
directeur de recherche

Gabriela Nicolescu
codirectrice

Frédéric Rousseau
membre du jury

Mémoire accepté le 16 mai 2006

Résumé et mots clés

De par leur nature, les applications multimédias bénéficient d'une implémentation sur un réseau sur puce. Deux applications sont considérées: un encodeur MPEG-4 et un algorithme de détection de contours. Elles sont distribuées et leur exécution est simulée par l'environnement de simulation de réseaux sur puce StepNP. Dans le premier cas (MPEG-4), nous optimisons la taille de la mémoire requise en développant un modèle de programmation avec pipeline fonctionnel. Elle passe de 40 Mo pour un encodeur hautement parallèle à 1 Mo pour notre version avec pipeline. Dans le deuxième cas (détection de contours), nous étendons les concepts de base des transformations de boucles (comme la fusion et le pavage) pour les utiliser dans un contexte multiprocesseur. Nous démontrons que la méthodologie proposée est extensible : les taux d'échecs des mémoires cache n'augmentent pas lorsque le nombre de processeurs utilisés augmente.

Mots clés : encodeur MPEG-4, fusion, mémoires cache, MPSoC, pavage, pipeline fonctionnel, StepNP, SystemC, transformations de boucles.

Summary and keywords

By their nature, multimedia applications take advantage of a network-on-chip (NoC) implementation. Two of them are considered: a MPEG-4 video encoder and a cavity detection algorithm. Both are distributed and simulated using the NoC system-level simulation tool StepNP. We present a functional pipeline programming model that helps us reduce the size of the memory required by the MPEG-4 encoder. Compared to the 40 MB of memory needed by a highly parallel version of the encoder, our implementation uses 1 MB. We also extend the basic concepts of loop transformations (like fusion and tiling) and use them in a multiprocessor context. We find that the proposed methodology is scalable: the miss ratios of the caches are stable when increasing the number of simulated processors.

Keywords : cache memories, functional pipeline, fusion, loop transformations, MPEG-4 encoder, MPSoC, StepNP, SystemC, tiling

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et abréviations

CASSE : **CA**mellia **S**ystem-on-chip **S**imulation **E**nvironment

CE : **C**oncurrency **E**ngine

DCT : **D**iscrete **C**osine **T**ransform

DSOC : **D**istributed **S**ystem **O**bject **C**omponent

DSM : **D**istributed **S**hared **M**emory

DSP : **D**igital **S**ignal **P**rocessor

DST : **D**ata **S**pace-oriented **T**iling

FPGA : **F**ield-**P**rogrammable **G**ate **A**rray

HIBI : **H**eterogeneous **IP** **B**lock **I**nterconnection

IDCT : **I**nverse **D**iscrete **C**osine **T**ransform

IP : **I**nternet **P**rotocol

IQ : **I**nverse **Q**uantization

ISS : **I**nstruction **S**et **S**imulator

Ko : **K**ilo-**o**ctet

LISA : **L**anguage for **I**nstruction **S**et **A**rchitectures

M : **M**illion

ME : **M**otion **E**stimation

Mescal : **M**odern **E**mbedded **S**ystems, **C**ompilers, **A**rchitectures and **L**anguages

MESH : **M**odeling **E**nvironment for **S**oftware and **H**ardware

MILAN : **M**odel based **I**ntegrated simu**LA**tio**N** framework

Mo : **M**égao**c**tet

MPEG : **M**oving **P**icture **E**xperts **G**roup

MPSoC : **Multi**processor-**S**ystem-**o**n-a-**C**hip

ms : **M**illiseconde

NoC : **N**etwork-**o**n-**C**hip

ns : **N**anoseconde

OCP : **O**pen **C**ore **P**rotocol

ORB : **O**bject **R**equest **B**roker

Q: **Q**uantization

RISC : **R**educed **I**nstruction **S**et **C**omputer

SAD : **S**um of **A**bsolute **D**ifferences

Sesame : **S**imulation of **E**mbedded **S**ystem **A**rchitectures for **M**ultilevel **E**xploration

SMT : **S**imultaneous **M**ulti**T**hreading

SoC : **S**ystem-**o**n-a-**C**hip

SOCP : **S**ystemC **O**pen **C**ore **P**rotocol

SOPC : **S**ystem-**o**n-a-**P**rogrammable-**C**hip

SPACE : **S**ystemC **P**artitioning of **A**rchitectures for **C**o-design of **E**mbedded systems

StepNP : **S**ys**T**em-level **E**xploration **P**latform for **N**etwork **P**rocessors

TLM : **T**ransaction **L**evel **M**odeling

UML : **U**nified **M**odeling **L**anguage

VLC : **V**ariable **L**ength **C**oding

VLIW : **V**ery **L**ong **I**nstruction **W**ord

XML : E**X**tensible **M**arkup **L**anguage

Remerciements

Plusieurs personnes ont contribué aux travaux qui ont mené à ce mémoire. Je voudrais d'abord remercier mon directeur de recherche, Dr El Mostapha Aboulhamid, Professeur titulaire à l'Université de Montréal, et ma codirectrice de recherche, Dr Gabriela Nicolescu, Professeure adjointe à la l'École Polytechnique de Montréal, pour leurs conseils, leurs contacts et leur grande connaissance du domaine.

Je voudrais également remercier Dr Youcef Bouchebaba pour sa précieuse aide avec la section sur le formalisme mathématique du modèle de programmation avec pipeline fonctionnel et pour notre fructueuse collaboration pour le chapitre sur l'optimisation de l'application de détection de contours. À ce sujet, je tiens à souligner l'obligeance de Bruno Lavigueur qui m'a autorisé à utiliser les résultats de ses tests dans la section 5.5.

L'optimisation de l'encodeur MPEG-4, présentée dans le chapitre 4, a été réalisée dans le cadre d'un projet de stage chez STMicroelectronics. Aussi, je désire remercier Dr Pierre Paulin ainsi que toute l'équipe de Advanced System Technology à Ottawa pour leur accueil, leur générosité et leur aide lors de mon passage chez eux.

Finalement, je tiens à remercier sincèrement mes parents Michel et Louise pour leur soutien indéfectible tant moral que, disons-le, financier!

Chapitre 1 : Introduction

Lorsqu'ils ne travaillent pas sur le développement de nouvelles technologies, les architectes et programmeurs ont souvent la tâche d'optimiser les technologies et logiciels existants. Par exemple, un programmeur peut avoir comme mandat d'optimiser une application donnée afin qu'elle puisse être exécutée par un processeur moins performant, ceci dans le but de réduire la consommation d'énergie du système. En particulier, les applications multimédias nécessitent une grande puissance de calcul. Il est donc souhaitable de bien les étudier afin de développer des systèmes multimédias peu coûteux et consommant peu d'énergie. Notre travail s'inscrit dans cette catégorie : l'optimisation d'applications. Nous tenterons d'optimiser des applications multimédias :

- pour qu'elles s'exécutent efficacement sur un réseau sur puce (aussi appelé NoC, pour *Network-on-Chip*).
- pour qu'elles utilisent efficacement la hiérarchie mémoire du réseau sur puce.

Les réseaux sur puce sont constitués de plusieurs processeurs (possiblement hétérogènes, c'est-à-dire de modèles différents), de mémoires partagées, de blocs matériels dédiés, etc. Toutes ses composantes sont intégrées sur une même puce de silicium et reliées entre elles par un réseau d'interconnexion qui permet à toute paire de composantes de communiquer à haut débit. Puisque de tels systèmes possèdent une grande puissance de calcul répartie sur plusieurs processeurs et fonctions matérielles dédiées, il est naturel de les utiliser pour exécuter des applications multimédias.

Ces applications servent à transformer ou à compresser/décompresser des photos, de la vidéo ou encore un extrait sonore. Elles nécessitent une grande puissance de calcul car elles sont souvent liées à des contraintes de temps réel (par exemple, un décodeur vidéo doit produire trente images par seconde afin d'assurer une bonne qualité visuelle à l'utilisateur). Elles sont souvent distribuées, c'est-à-dire construites de telle sorte que l'on puisse utiliser plusieurs processeurs fonctionnant en parallèle pour leur

exécution, ceci afin d'augmenter leur rapidité. Historiquement, le fait de "distribuer" une application impliquait que les ressources utilisées par cette application (les processeurs par exemple) ne se trouvaient pas sur la même machine et étaient possiblement séparées par de grandes distances (Internet par exemple). Par contre, l'émergence de réseaux sur puce comportant plusieurs ressources hétérogènes a fait en sorte qu'il est maintenant fréquent de parler d'applications "distribuées" dans ce contexte (ce que nous ferons pour la suite de ce document).

Supposons l'application de traitement d'images suivante : quatre fonctions doivent être exécutées sur chaque pixel de l'image en entrée. Si le développeur de cette application souhaite la distribuer afin de l'exécuter sur un réseau sur puce comprenant quatre processeurs, il pourrait par exemple :

1) diviser l'image en quatre, puis assigner à chaque processeur une partie de l'image. Chaque processeur exécute alors les quatre fonctions sur chacun des pixels contenus dans sa section.
2) utiliser chaque processeur pour exécuter une fonction donnée. Le premier processeur exécute la première fonction sur tous les pixels de l'image, le deuxième exécute la deuxième fonction sur tous les pixels, etc.

S'il s'aperçoit qu'une fonction en particulier prend plus de temps à s'exécuter que les autres, il pourrait également choisir de l'implémenter en matériel, puis de connecter ce bloc matériel au réseau d'interconnexion du réseau sur puce afin que les processeurs puissent l'utiliser. Cette façon de faire augmenterait la rapidité d'exécution de l'application, mais diminuerait la flexibilité du système et augmenterait, dans le cas où le développeur ne possède pas d'implémentation du bloc matériel, son temps de développement.

Tous ces choix sont non triviaux et demandent aux développeurs d'applications distribuées une grande connaissance de l'architecture des réseaux sur puce. En particulier, ils doivent tenir compte de leur hiérarchie mémoire, c'est-à-dire des configurations de leurs mémoires cache, de leurs mémoires blocs-notes et de leurs

mémoires externes. Celles-ci influenceront grandement la manière de programmer des développeurs, le but du jeu étant de maximiser le taux de succès d'accès aux mémoires cache afin de réduire le plus possible les accès à la mémoire externe, très coûteux en termes de temps et de puissance dissipée.

Il existe une autre façon de voir le problème : en diminuant la taille de la mémoire requise par l'application, il devient possible d'utiliser des mémoires blocs-notes pour sauvegarder une partie ou même la totalité des instructions et des données sur la puce même, évitant ainsi plusieurs accès à la mémoire externe.

Bien que tout cela soit déjà bien complexe en soi, un autre facteur peut venir ajouter au désordre : la possibilité de construire et configurer son propre réseau sur puce avant d'y exécuter une application donnée. Les environnements de simulation de réseaux sur puce possèdent une telle souplesse, rendant le travail du développeur encore plus complexe. Il ne s'agit plus de développer une application pour un réseau sur puce donné, mais de développer une application tout en modifiant le réseau sur puce afin que la combinaison logiciel/matériel donne les meilleurs résultats possibles. Cette exploration architecturale demande encore une fois aux développeurs une compréhension aiguë de l'architecture des réseaux sur puce.

Ce travail intègre les trois concepts présentés plus haut que sont l'utilisation de réseaux sur puce, l'optimisation d'applications multimédias ainsi que l'exploration architecturale. Un aperçu plus détaillé du contenu de ce document est présenté dans les prochaines sections. La problématique du sujet, les objectifs du projet, les contributions apportées par ce travail ainsi que le plan du présent document sont présentés dans les sections 1.1 à 1.4.

1.1 Problématique

Les applications multimédias, et plus particulièrement les applications de traitement d'images, nécessitent de puissants processeurs, une mémoire de grande taille et une bande passante élevée. Ceci est d'autant plus vrai lorsqu'on doit respecter des contraintes de temps réel. Afin de répondre à ces exigences, les concepteurs de systèmes embarqués[1] peuvent emprunter deux approches : développer un système spécifique à l'application, plus performant et qui consomme moins d'énergie, ou utiliser un réseau sur puce. La deuxième approche favorise la réutilisation de composantes matérielles et permet de réduire le temps d'accès au marché. Cette solution est de plus en plus adoptée, dans un marché toujours très compétitif.

Pour des raisons de dissipation de puissance, les concepteurs de réseaux sur puce favorisent l'utilisation de plusieurs processeurs simples sur celle de processeurs superscalaires, plus complexes. Ceci force les programmeurs à développer des versions distribuées de leur application multimédia. Cette tâche est souvent complexe : du fonctionnement et de la performance finale de l'application dépendent plusieurs facteurs dont la taille de la mémoire requise et l'attention particulière portée à la localité des données pour assurer un bon taux de succès des mémoires cache.

Ces deux facteurs sont non triviaux et nous ont poussés à compléter deux études de cas pour mieux les comprendre. La première étude considère le problème de la taille de la mémoire. Nous optimisons d'abord une application d'encodage vidéo MPEG-4 afin qu'elle réponde aux critères fixés par des concepteurs de réseaux sur puce. Ensuite, nous testons sa validité sur plusieurs configurations de réseaux sur puce et donnons les résultats. Dans la deuxième étude, nous modifions une version distribuée d'une application de détection de contours afin d'améliorer le taux de succès des mémoires cache.

---

[1] Un système embarqué est un système électronique dédié à une application dont la définition inclue une limitation des ressources disponibles. Cette limitation est généralement d'ordre spatiale (taille limitée), et énergétique (consommation restreinte). (source : http://fr.wikipedia.org)

1.2 Objectifs

Ce travail s'inscrit dans la recherche continue de systèmes plus performants, plus génériques, consommant moins d'énergie, faciles à programmer et peu coûteux. Nous nous concentrons sur la hiérarchie mémoire, grande consommatrice d'énergie et goulot d'étranglement dans les systèmes embarqués. Notre but est d'aider à la mise en place de modèles de programmation parallèle[1] tenant compte des contraintes de mémoire et utilisant efficacement la hiérarchie mémoire disponible. Par exemple, nous voulons implémenter un modèle de programmation avec pipeline fonctionnel (décrit dans la section 4.3) qui réduira grandement la taille de la mémoire requise par l'encodeur MPEG-4.

Parallèlement à l'étude des modèles de programmation se fait celle de la hiérarchie mémoire, plus précisément des mémoires cache de niveau 1 et 2, de la mémoire cache du coprocesseur ainsi que des mémoires blocs-notes. Nous voulons mettre en évidence l'importance d'étudier la hiérarchie mémoire afin de choisir une plateforme matérielle efficace. Nous simulerons l'exécution de l'encodeur MPEG-4 avec pipeline à l'aide d'un environnement de simulation de réseaux sur puce afin d'étudier l'impact d'une hiérarchie mémoire donnée sur la rapidité d'exécution de l'algorithme.

Nous souhaitons aussi montrer comment on peut étendre les techniques connues d'optimisation de boucles telles la fusion et le pavage à un contexte multiprocesseur. Nous allons modifier une application de traitement d'images typique, soit la détection de contours, afin d'utiliser la fusion, le pavage, le décalage de boucles et autres dans un contexte multiprocesseur et ainsi montrer qu'il est possible de conserver un haut degré de parallélisme tout en bénéficiant des avantages de ces techniques. Nous utiliserons un environnement de simulation de réseaux sur puce pour comparer les performances des versions optimisées et non optimisées.

---

[1] Techniques logicielles qui permettent d'exprimer concrètement des algorithmes parallèles et de les rendre compatibles avec les systèmes parallèles sous-jacents. (source : http://www.wikipedia.org traduction libre)

1.3 Contributions

Ce travail apporte deux contributions principales au domaine de la recherche sur les systèmes multiprocesseurs :

1) Nous décrivons un modèle de programmation avec pipeline fonctionnel (voir section 4.3) appliqué à un encodeur vidéo MPEG-4. L'exécution de ce modèle sur un réseau sur puce comprenant plusieurs processeurs ARM, une hiérarchie mémoire à trois niveaux et plusieurs fonctions matérielles dédiées est simulée par l'environnement StepNP. L'utilisation d'un pipeline pour implémenter un encodeur vidéo est une pratique courante. Notre démarche se démarque parce que nous utilisons un pipeline fonctionnel (c'est-à-dire que le pipeline est implémenté en logiciel). À notre connaissance, aucun autre modèle de pipeline fonctionnel dont l'unité de base est le macroblock (c'est-à-dire un bloc de 16×16 pixels) n'a été étudié dans le contexte d'un réseau sur puce multiprocesseur.

2) Nous proposons une méthodologie qui permet d'appliquer les concepts de base des transformations de boucles dans un contexte multiprocesseur afin de réduire le taux d'échec des mémoires cache (voir le chapitre 5). Ces concepts ont traditionnellement été étudiés pour des systèmes monoprocesseurs. Notre méthodologie est extensible (indépendante du nombre de processeurs utilisés) et produit des résultats similaires à ceux obtenus dans un contexte monoprocesseur.

1.4 Plan du document

La suite de ce document est divisée en six chapitres. Le chapitre 2 contient une revue des travaux publiés dont le sujet est relié au nôtre. Les principaux sujets touchés sont l'optimisation d'algorithmes d'encodage vidéo et les transformations de boucles.

Le chapitre 3 présente brièvement plusieurs environnements de simulation de réseaux sur puce et détaille celui qui a été retenu pour tester les optimisations proposées et obtenir les résultats de simulation présentés dans les chapitres 4 et 5. Il propose également un ajout à la plateforme de simulation, soit l'introduction d'un protocole de cohérence de mémoires cache.

Le chapitre 4 présente la première étude de cas : l'optimisation de la taille de la mémoire requise par une application d'encodage MPEG-4 ainsi que l'exploration architecturale de la plateforme simulée sur laquelle elle s'exécute. L'application est présentée, puis les détails des optimisations effectuées sont donnés avec les conséquences qu'elles provoquent et les résultats obtenus. Un résumé des changements apportés à l'architecture de la plateforme simulée et les résultats de performance et les taux d'échecs des mémoires cache sont ensuite donnés.

Le chapitre 5 présente la deuxième étude de cas : la généralisation des concepts de base des transformations de boucles (dont l'exécution est limitée à un contexte monoprocesseur) afin de les utiliser dans un contexte multiprocesseur, puis la distribution des calculs sur les processeurs disponibles. Une application de traitement d'images, la détection de contours, est utilisée pour valider les méthodes proposées. Un rappel des concepts de base des transformations de boucles est présenté, puis les détails de la distribution de l'application sont donnés, suivis des résultats.

Le chapitre 6 donne un aperçu des travaux futurs et des idées à développer.

Le chapitre 7 présente la conclusion.

Chapitre 2 : État de l'art

Notre première étude de cas porte sur l'optimisation mémoire d'un encodeur vidéo MPEG-4. La norme MPEG-4 est décrite en [19] et présentée très brièvement dans la section 4.1. La complexité des calculs qu'elle décrit a fait en sorte que plusieurs chercheurs l'ont choisie comme étude de cas. Chaque trame de la vidéo à encoder est divisée en blocs de 16×16 pixels, appelés macroblocks, puis encodée séquentiellement. Les macroblocks passent par une série de fonctions qui utilisent les principes de la redondance spatiale et temporelle pour produire un fichier compressé qui devra être lu par un décodeur avant d'être affiché à l'écran.

Denolf et al. [20] ont démontré comment l'optimisation d'un algorithme d'encodage MPEG-4 (réduire le nombre d'accès mémoire ainsi que le nombre d'instructions exécutées) décrit dans un langage de haut niveau d'abstraction facilitait la sélection d'une architecture lors de son raffinement vers une plateforme matérielle. En particulier, les tests effectués sur la version fonctionnelle de l'algorithme ont permis de développer une hiérarchie mémoire efficace avant même d'implémenter tout le système sur un FPGA. Cette méthodologie est présentée en détails en [21, 22].

Plusieurs architectures ont été proposées pour accueillir des applications d'encodage et de décodage MPEG-4. Par exemple, Wang et al. [23] ont proposé une plateforme matérielle pour l'exécution d'un décodeur vidéo MPEG-4. Les macroblocks traversent un pipeline, dont certains étages sont implémentés en logiciel et d'autres en matériel, avant de créer les images en sortie. Le pipeline augmente le débit du décodeur en divisant le processus de décodage des macroblocks en plusieurs étapes dont l'exécution s'effectue en parallèle (par exemple, l'étape 1 du macroblock n et l'étape 2 du macroblock n-1 s'effectuent simultanément).

Stolberg et al. [24] ont proposé M-PIRE, une plateforme programmable optimisée pour l'encodage et le décodage vidéo MPEG-4. Elle comprend trois processeurs programmables : un processeur MIPS qui exécute les opérations de contrôle, un

processeur VLIW qui exécute les fonctions d'estimation de mouvement, de compensation de mouvement, de DCT/IDCT, de quantisation et de quantisation inverse sur les macroblocks et un processeur audio comprenant deux DSP.

Salminen et al. [25] ont présenté un SOPC (*System-on-Programmable-Chip*) qui contient huit processeurs Nios et un contrôleur de mémoire externe reliés par le réseau d'interconnexion HIBI [26]. Implémenté sur un Altera Stratix, le système utilise 36 402 éléments logiques et atteint une fréquence de 78 MHz. En se basant sur cette architecture, Lehtoranta et al. [27] ont implémenté un encodeur vidéo MPEG-4 parallèle. Les trames sont divisées en trois fenêtres horizontales, puis encodées simultanément par les processeurs esclaves. Le processeur maître reçoit les trames à encoder de l'ordinateur hôte via une connexion Ethernet, configure les trois processeurs esclaves via la mémoire partagée, effectue la concaténation des fenêtres encodées par les processeurs esclaves et envoie les trames encodées à l'ordinateur hôte. Puisque la mémoire partagée devient rapidement un goulot d'étranglement lorsque le nombre de processeurs esclaves augmente, les auteurs ont proposé l'utilisation de tampons locaux afin de diminuer son utilisation.

. . . . . .

Afin d'améliorer le taux de succès des mémoires cache dans les applications destinées aux systèmes embarqués et d'ainsi diminuer leur consommation d'énergie, Cathoor et al. [21] ont suggéré l'utilisation des transformations de boucles. Depuis, elles ont été grandement étudiées. En particulier, la fusion, une technique combinant plusieurs nids de boucles en un seul, a été beaucoup étudiée [28, 29, 30, 31, 32].

Marchal et al. [32] ont développé un modèle qui utilise une configuration de la hiérarchie mémoire passée en paramètre pour sélectionner quels nids de boucles fusionner afin d'optimiser les accès mémoire. Fraboulet et al. [29] ont développé un algorithme optimal qui minimise la taille des tableaux intermédiaires.

D'autres ont proposé d'éliminer ces tableaux intermédiaires [33, 34], notamment en remplaçant les variables indicées par des variables scalaires temporaires dont l'utilisation est plus sujette à être optimisée par les compilateurs qui utilisent alors des registres pour sauvegarder leur valeur. Une autre technique, appelée *storage order optimization* [35] ou optimisation de l'allocation de la mémoire, consiste à faire pointer certains tableaux ou certaines parties de tableaux vers le même espace mémoire, à condition que leur utilisation ne se chevauche pas.

Les travaux sur le pavage, une technique visant à diviser l'espace d'itérations d'une boucle en tuiles dont l'encodage s'effectue séquentiellement, se concentrent surtout sur l'optimisation d'un seul nid de boucles à la fois [36, 37, 38]. Il serait cependant intéressant de pouvoir analyser une séquence de plusieurs nids de boucles avant de déterminer de quelle façon il est préférable de former les tuiles.

Kandemir et al. [39] ont donc proposé de modifier la fonction d'itération de plusieurs nids de boucles dans le but d'améliorer la localité temporelle des données. Kadayif et Kandemir [40] ont introduit DST, pour *data space-oriented tiling* ou pavage orienté sur les données. Ce type de pavage divise l'espace d'adressage en tuiles de données. L'encodage d'une tuile nécessite donc l'exécution de toutes les itérations de tous les nids de boucles accédant aux données contenues dans celle-ci.

Une autre technique, décrite en [31], propose d'appliquer simultanément la fusion et le pavage sur une séquence de plusieurs nids de boucles. L'avantage de cette technique est qu'elle permet d'effectuer un pavage plus flexible tout en conservant les avantages qu'apporte la fusion. De plus, elle remplace les tableaux intermédiaires du code non fusionné par des tampons de plus petites tailles.

Les techniques décrites plus haut ont une caractéristique commune : elles ont été étudiées et appliquées dans un contexte monoprocesseur. Par contre, l'émergence des MPSoC (*Multiprocessor-Systems-on-a-Chip*) comme plateformes pour les applications multimédias a motivé plusieurs études sur le sujet. Kandemir et al. [41]

ont proposé de diviser l'espace d'adressage en tuiles de données (de la même façon que DST, introduit plus haut) partitionnées sur les processeurs disponibles. Chaque processeur est responsable d'exécuter toutes les itérations qui accèdent aux données contenues dans les tuiles qui lui sont assignées. Chen et Kandemir [42] ont quant à eux présenté une stratégie automatique de réorganisation des itérations des boucles afin que les accès aux données partagées par plusieurs processeurs soient effectués à l'intérieur d'une même tranche de temps, ceci dans le but de minimiser les accès à la mémoire externe. Bouchebaba et al. [43] ont cherché à étendre les concepts de base des transformations de boucles, comme la fusion et le pavage, afin de les utiliser dans un contexte multiprocesseur.

Plusieurs ont proposé l'utilisation de mémoires blocs-notes en remplacement des mémoires cache traditionnelles [44, 45]. Ces mémoires sont gérées par le programmeur ou le compilateur qui cherche généralement à ce qu'elles contiennent les données les plus fréquemment utilisées par les processeurs. Elles sont situées près des processeurs (sur la même puce de silicium) et sont donc très rapides. Leur plage d'adresse est fixe : les concepts de base des mémoires cache comme les étiquettes, l'associativité et les blocs ne sont donc pas utilisés. Les mémoires blocs-notes sont utilisées pour accélérer les accès mémoire et réduire la consommation d'énergie. L'idée de gérer dynamiquement le contenu de telles mémoires afin de minimiser les accès à la mémoire externe est intéressante. Par contre, l'utilisation de telles mémoires nécessite souvent la création d'instructions assembleur spéciales ou de contrôleurs matériels qui sont nécessairement dépendants de l'architecture du système.

Nos travaux diffèrent de ceux présentés dans cette section. L'encodeur vidéo MPEG-4 parallèle que nous présentons utilise un pipeline fonctionnel dont l'unité de base est le macroblock. Ce pipeline permet de réduire considérablement la taille de la mémoire requise par l'encodeur tout en conservant un haut degré de parallélisme. La mise en place du pipeline s'appuie sur un formalisme mathématique, présenté dans la section 4.3.1. Nos travaux portant sur l'optimisation de l'application de détection de

contours, présentés dans le chapitre 5, ont été publiés dans [43]. En se basant sur les concepts de base des transformations de boucles, développés pour des systèmes monoprocesseurs, nous proposons une méthodologie extensible et efficace pour réduire le taux d'échecs des mémoires cache dans un contexte multiprocesseur.

Chapitre 3 : Présentation de l'environnement de simulation des réseaux sur puce

Durant les dernières années, plusieurs environnements de simulation de réseaux sur puce ont été développés. Nous avons choisi StepNP [1, 2, 3], développé par la compagnie STMicroelectronics, comme environnement de base pour toutes nos simulations. Brièvement, ce choix a été motivé par cinq facteurs. StepNP, dont la description détaillée est donnée plus loin dans le chapitre :

- supporte le modèle TLM (*Transaction Level Modeling*) [4].
- supporte les processeurs à contextes d'exécution multiples.
- propose un ordonnanceur matériel contrôlé par des primitives logicielles simples et efficaces.
- supporte la création de modules matériels dédiés ainsi que les communications logiciel/matériel, logiciel/logiciel et matériel/matériel par passage de messages.
- permet d'obtenir plusieurs résultats de performance.

Le modèle TLM est utilisé pour la modélisation de systèmes à haut niveau d'abstraction. Les détails des communications sont isolés des détails des calculs. Trois niveaux d'abstraction sont considérés dans le modèle TLM : sans notion de temps, avec notion de temps approximatif et précis au niveau du cycle. Des neuf paires possibles entre ces niveaux, six font partie du modèle TLM (elles sont représentées par les lettres 'A' à 'F' dans la figure 3.1).

Ainsi, un développeur dont la tâche est d'optimiser les communications peut, à partir de descriptions de haut niveau d'abstraction, raffiner le modèle du bus tout en conservant les descriptions des autres modules à un haut niveau. Il économise donc du temps de développement et de simulation en se concentrant uniquement sur le module qui le préoccupe.

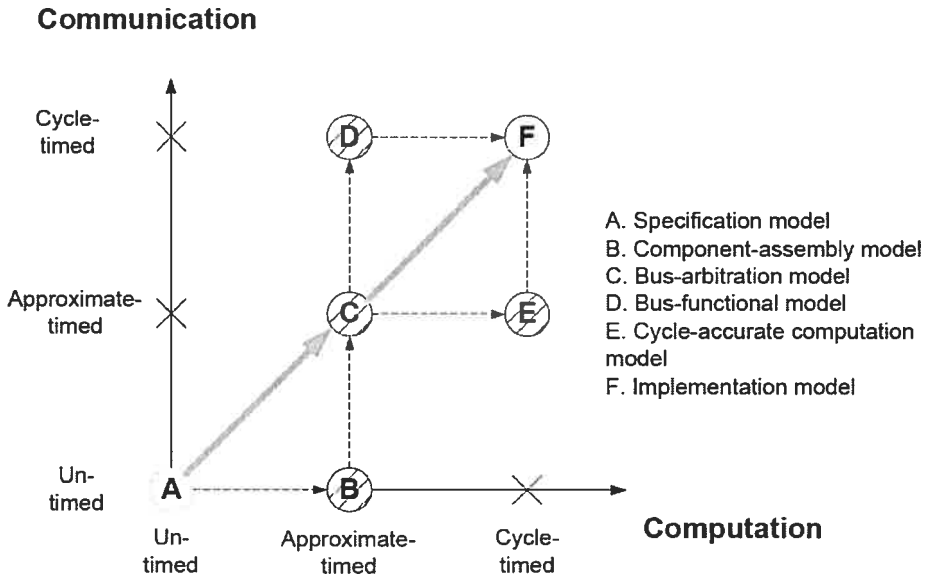


Figure 3.1 : Modèles décrits par le TLM (image tirée de la référence [4]).

CASSE [5] et SPACE [6] utilisent le modèle TLM. CASSE utilise une description fonctionnelle du système à analyser, représenté par un réseau de processus de Kahn [7], comme point de départ. Il effectue ensuite l'allocation des processus sur une architecture prédéfinie, selon les contraintes fournies par le programmeur.

SPACE permet aux développeurs de définir leur système selon plusieurs niveaux d'abstraction. Au plus haut niveau, le modèle est défini comme un ensemble de modules SystemC [8] interconnectés entre eux. Aucun partitionnement logiciel/matériel n'est alors effectué. Au plus bas niveau, un partitionnement est réalisé : les tâches logicielles sont simulées par un ISS[1] du processeur ARM, lequel est encapsulé dans un module SystemC. L'avantage de cet environnement est qu'il inclut un système d'exploitation en temps réel, répondant ainsi aux demandes des développeurs d'applications à contraintes de temps réel dures qui pouvaient

[1] ISS (**I**nstruction **S**et **S**imulator) : modèle de simulation (généralement codé avec un langage de programmation de haut niveau) reproduisant le comportement d'un processeur. Le ISS lit les instructions d'un programme compilé et conserve l'état des registres dans des variables internes. (source : http://www.wikipedia.org traduction libre)

difficilement utiliser SystemC comme environnement de développement. Les communications logiciel/matériel peuvent être fonctionnelles sans notion du temps, fonctionnelles avec notion du temps ou encore précises au niveau cycle.

Récemment, Pimentel et al. ont présenté Sesame [9] qu'ils décrivent comme un environnement de modélisation et de simulation haut niveau effectuant l'évaluation de la performance et l'exploration architecturale de SoC hétérogènes et ciblant les applications multimédias. Sesame prône l'utilisation de la méthodologie de modélisation en Y, présentée dans la figure 3.2. La description des applications, indépendante de celle de l'architecture, est représentée par un réseau de processus de Kahn. Les couches application et architecture sont séparées par la couche allocation, qui convertit les événements provenant du simulateur de la couche application (une écriture mémoire, une structure de contrôle, etc.) en graphes de flots de données contrôlés par des entiers (*integer-controlled data flow graphs*) dont certains nœuds sont synchronisés avec la couche architecture. Le niveau d'abstraction des graphes détermine la précision et la rapidité de la simulation.

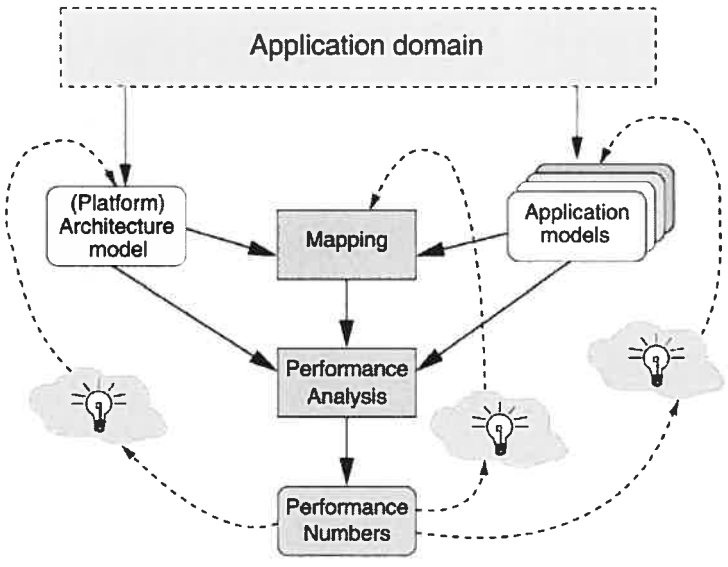


Figure 3.2 : Méthodologie de modélisation en Y (image tirée de la référence [9]).

Des environnements similaires à Sesame ont été développés, notamment Archer [10], MESH [11], MILAN [12], Metropolis [13] et Mescal [14].

D'autres environnements de modélisation et simulation de systèmes sur puce se basent sur UML [15], sur XML [16], utilisent l'environnement de modélisation de processeurs LISA [17] ou permettent la réutilisation de modèles développés en C dans un environnement SystemC [18].

Puisque notre choix s'est finalement arrêté sur l'environnement StepNP, nous en donnons maintenant une description plus détaillée : StepNP permet de créer une plateforme de simulation composée de processeurs, de mémoires cache, de réseaux d'interconnexion, de composantes matérielles spécifiques (la fonction racine carrée, la DCT ou transformée en cosinus discrète bidimensionnelle, etc.) (voir figure 3.3). L'environnement permet également de lancer la simulation de l'exécution d'un programme compilé, possiblement distribuée sur plusieurs processeurs. Enfin, il permet la collecte d'informations et de statistiques pour buts d'analyse et de raffinement du système. Il est basé sur le langage C++ et utilise la librairie SystemC [8] pour son simulateur et son ordonnanceur de processus.

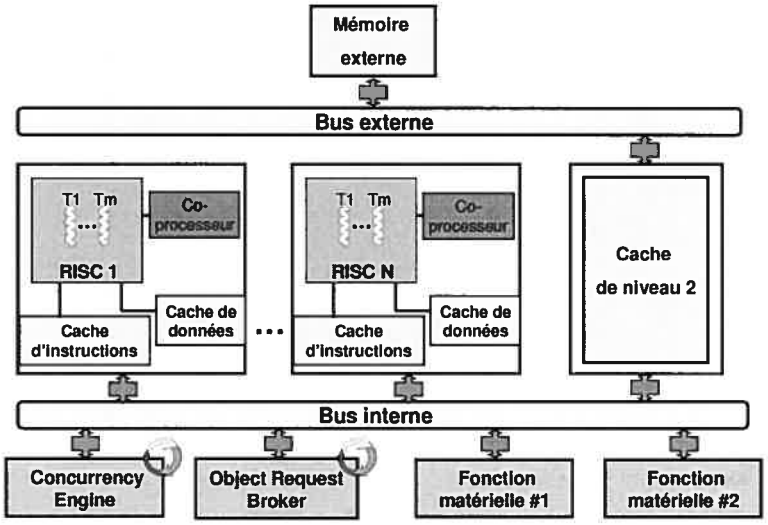


Figure 3.3 : Composantes matérielles d'une plateforme simulée.

Afin de mieux comprendre l'impact de toutes les composantes des plateformes simulées sur la performance des applications étudiées, nous avons effectué plusieurs simulations dont les résultats sont présentés dans les chapitres 4 et 5. Pour cela, nous avons fait varier le nombre de processeurs de 1 à 16, le nombre de contextes d'exécution par processeur de 1 à 16 également et la taille de la mémoire cache de données par processeur de 0,5 à 64 Ko. Nous avons également fait varier la taille de la mémoire cache de niveau 2 de 32 à 1024 Ko. Nous n'avons pas pris en compte les coûts relatifs en termes de dissipation de puissance et de nombre de transistors requis par les solutions étudiées dans l'évaluation de nos résultats.

3.1 Composantes typiques

Plusieurs composantes matérielles sont communes à tous les réseaux sur puce. Il s'agit des processeurs, des mémoires, des réseaux d'interconnexion et des coprocesseurs. Nous examinons plus en détails comment chacune de ces composantes est définie dans l'environnement StepNP.

3.1.1 Processeurs

Plusieurs simulateurs d'instructions (ISS) sont intégrés à l'environnement StepNP. Nous avons choisi d'utiliser le simulateur d'instructions de l'architecture ARM. Il est celui que les concepteurs de l'environnement ont le plus utilisé lors de son développement. De plus, ils ont créé une version multiprocessus (*multithreaded*) du ISS en encapsulant son contenu dans un SC_MODULE de SystemC. À moins d'avis contraires, cette version du processeur est utilisée lors de toutes les simulations dont les résultats sont affichés dans ce travail.

Ce processeur simulé applique les principes du *multithreading* [46], c'est-à-dire qu'il possède plusieurs bancs de registres afin d'effectuer des changements de contexte à

chaque coup d'horloge. Cependant, il ne permet pas à plusieurs contextes d'exécution de lancer des instructions simultanément : il n'applique pas les principes du SMT (*simultaneous multithreading* [46]).

Plusieurs politiques d'ordonnancement peuvent administrer un tel processeur. Nous avons choisi d'utiliser une politique simple : le tourniquet. De cette façon, il est possible de remplir le pipeline du processeur d'instructions provenant de contextes d'exécutions différents. Ces instructions sont donc nécessairement indépendantes. Il est alors possible d'utiliser des processeurs avec des pipelines plus simples, qui ne contiennent pas la logique complexe requise pour effectuer les envois (*forwarding*). Dans l'ordre, chaque contexte d'exécution se voit offrir la possibilité de lancer une instruction.

Il choisit de le faire lorsque :

1) il n'est pas bloqué (c'est-à-dire en attente d'une donnée de la mémoire ou d'une réponse du coprocesseur).
2) il ne possède pas déjà une instruction dans le pipeline du processeur.

Le premier contexte à satisfaire à ces conditions, c, lance une instruction. Lors du cycle suivant, l'ordonnanceur commencera la vérification des conditions à partir du contexte d'exécution $(c + 1)$ % *Nombre_de_contextes*.

3.1.2 Mémoires

Plusieurs modèles de mémoire sont disponibles lors de la création de la plateforme simulée. Nous utilisons trois d'entre eux et avons développé nos propres modèles pour l'implémentation d'un protocole de cohérence de mémoires cache (voir section 3.3) ainsi que pour la prélecture des données par une mémoire bloc-notes (voir section 4.6.5).

Ces mémoires sont configurables. On peut modifier leur taille, leur latence, leur plage d'adresses, la taille d'un mot, etc. On peut également définir le nombre de voies, d'ensembles, la politique d'écriture et la taille des blocs des mémoires cache. Finalement, il est possible de "diviser" une mémoire cache de données de niveau 1 afin que chaque contexte d'exécution possède sa propre mémoire cache logique, indépendante des autres.

3.1.3 Réseaux d'interconnexion

Une plateforme simulée comprend généralement deux réseaux d'interconnexion : un bus interne et un bus externe. Dans les deux cas, nous avons opté pour un réseau de type *crossbar* défini au niveau d'abstraction TLM (*Transaction Level Modeling*) [4]. La raison principale qui a motivé notre choix est la rapidité des communications offerte par ce type de réseaux. En effet, un lien direct est établi entre toutes les paires de composantes accédant au réseau. Puisque nous souhaitons simuler des applications multimédias (qui requièrent une bande passante élevée), cette caractéristique devient très intéressante, comparée à un réseau de type bus, qui utilise un seul lien partagé par toutes les composantes. D'autres types de réseaux sont également en cours de développement. La latence du réseau peut être configurée, ce qui permet d'obtenir une bonne approximation du temps requis pour les communications extra-modulaires (c'est-à-dire entre les processeurs et la mémoire cache de niveau 2, entre la mémoire cache de niveau 2 et la mémoire externe, etc.).

Les réseaux utilisés sont compatibles avec une version réduite du protocole de communications OCP (*Open Core Protocol* [47]) : SOCP (SystemC OCP). Cette version a été développée par les concepteurs de l'environnement StepNP. Elle fait abstraction des notions de signaux et de minutage détaillé, non essentielles pour l'élaboration de réseaux au niveau TLM. Ainsi, tous les modules connectés aux bus interne et externe doivent être soit compatibles avec le protocole SOCP, soit munis d'une interface les rendant compatibles.

3.1.4 Coprocesseurs

Deux types de coprocesseurs coexistent sur la même plateforme : les coprocesseurs associés à chaque processeur ARM, connectés à celui-ci par un pont, et les coprocesseurs accessibles par tous les processeurs, connectés au bus interne.

Dans le premier cas, le processeur communique avec ses coprocesseurs en lisant/écrivant à des adresses désignées. Nous les utilisons entre autres pour accélérer les opérations de division, modulo et valeur absolue.

Dans le deuxième cas, le coprocesseur implémente un ou plusieurs services en matériel, accessibles aux processeurs qui doivent créer un client logiciel leur permettant de lancer des requêtes. Ce type de requêtes par passage de messages est décrit en détails dans la section 3.2.2. Nous utilisons cette méthode pour implémenter des fonctions complexes comme la DCT, la quantisation et la somme des valeurs absolues des différences (SAD).

3.2 Composantes spécifiques

Les concepteurs de réseaux sur puce doivent trouver une solution aux problèmes de la synchronisation des données, de l'ordonnancement des fils d'exécution sur les processeurs disponibles et des communications processeur/processeur et processeur/coprocesseur. StepNP le fait à l'aide des deux modules matériels décrits dans les prochaines sections : le CE (*Concurrency Engine*) et le ORB (*Object Request Broker*).

3.2.1 *Concurrency Engine*

Ce module, connecté au bus interne, possède plusieurs fonctions dont la gestion des sémaphores et des moniteurs et l'ordonnancement des fils d'exécution sur les processeurs. Un processeur souhaitant acquérir un sémaphore effectue une lecture à une adresse désignée et reste bloqué tant que le CE n'a pas répondu.

Pour que le CE puisse fonctionner correctement, le protocole de communications utilisé par le réseau d'interconnexion, SOCP, doit supporter les transactions imbriquées. En effet, le CE doit pouvoir retarder la réponse à une requête d'obtention d'un sémaphore jusqu'à ce que le processeur qui le possède fasse une requête pour le relâcher.

En supposant une fréquence d'horloge de 200 MHz, le lancement d'un fil d'exécution prend moins de 100 ns pour s'accomplir (l'appel à la fonction *fork*, l'exécution de l'ordonnanceur en matériel, le transfert du contexte d'exécution vers un processeur disponible). Ceci permet aux programmeurs d'applications d'utiliser un parallélisme à grains fins (moins de 500 instructions RISC).

3.2.2 *Object Request Broker*

Les communications processeur/processeur et processeur/coprocesseur utilisent le modèle client/serveur. Plusieurs serveurs, implémentés en logiciel ou en matériel, peuvent offrir le même service. La répartition des tâches doit donc être centralisée afin d'assurer l'équilibre des charges entre les serveurs.

Le module matériel responsable de cette fonction est le ORB (voir figure 3.4). Il place les requêtes des clients dans la file d'attente associée au service demandé et envoie celles-ci vers les serveurs lorsqu'ils sont disponibles. Ainsi, il faut moins de 30 instructions (150 ns en supposant une fréquence d'horloge de 200 MHz) pour

lancer une requête et recevoir la réponse. Ceci comprend : le lancement de la requête par le client, l'envoi des arguments en entrée, le choix d'un serveur par le ORB, la réception des arguments en entrée par le serveur et l'envoi du résultat final vers le client.

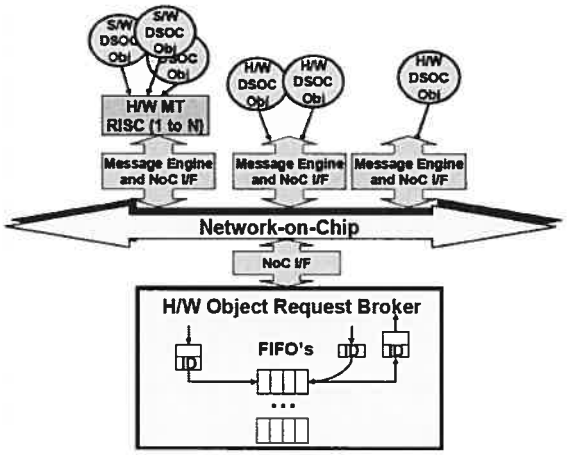


Figure 3.4 : Représentation schématique du *Object Request Broker* (image tirée de [2]).

3.3 Implémentation d'un protocole de cohérence de mémoires cache sur StepNP

Le caractère extensible de l'environnement StepNP nous permet de créer de nouvelles composantes et de les ajouter à la plateforme simulée. Afin de faciliter le débogage des applications distribuées, nous avons développé un modèle de mémoires cache qui implémentent un protocole de cohérence des données.

Certaines applications distribuées partagent beaucoup de données. Ces données doivent souvent être protégées et utilisées à l'intérieur de zones critiques. La responsabilité de créer ces zones critiques, de les protéger avec des *mutex* ou des sémaphores ou de spécifier ces données comme étant non "cachables" incombe aux programmeurs. Il est facile d'oublier de protéger certaines données et d'ainsi créer

des bogues dans l'application.

Il faut donc trouver une façon d'assister le programmeur lors du développement de l'application afin qu'il puisse détecter le plus tôt possible les données sujettes à la concurrence critique. Notre modèle de mémoires cache cohérentes fournit les renseignements nécessaires au programmeur pour éviter toute erreur de ce genre. Une fois que le comportement de l'application est cohérent, le programmeur peut désactiver le protocole de cohérence, qui est uniquement utilisé pour la phase de débogage de l'application.

Nous avons choisi d'utiliser un protocole de cohérence très simple, à trois états, semblable à celui présenté en [48] : chaque bloc de la mémoire cache de niveau 1 peut être dans l'état exclusif, partagé ou invalide. Les transitions entre les différents états sont montrées dans la figure 3.5.

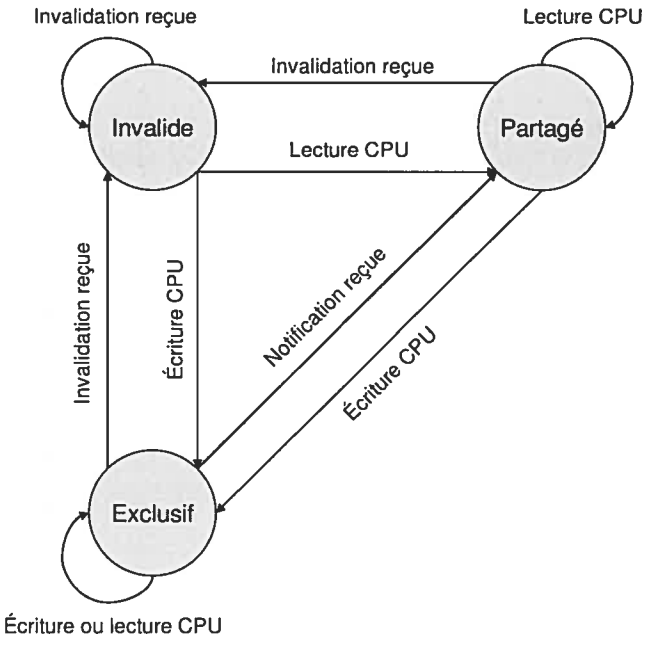


Figure 3.5 : Transitions d'états du protocole de cohérence.

Nous nous sommes ensuite basés sur les travaux de Chaudhuri et Heinrich pour implémenter le protocole dans StepNP. Dans [49], ils présentent SMTp : un processeur SMT (*Simultaneous Multithreading*) qui réserve un de ses contextes d'exécution pour la gestion d'un protocole de cohérence. Augmenté d'un contrôleur mémoire, ce processeur peut servir d'élément de base pour construire un MPSoC qui contient des mémoires cache cohérentes et se base sur une architecture à mémoire partagée distribuée (aussi appelé DSM, pour *distributed shared-memory multiprocessor*). Nous avons donc modifié les modèles de base de StepNP pour créer :

- un modèle de mémoire cache cohérente : elle contient, en plus des données, l'état de tous ses blocs (invalide, partagé ou exclusif). Une écriture à une adresse désignée permet de modifier l'état d'un bloc donné. Par exemple, un message d'invalidation est envoyé à toutes les mémoires cache lorsqu'une d'entre elles demande l'exclusivité sur un bloc. Les mémoires qui contiennent une copie de ce bloc doivent donc modifier son état en conséquence. Un deuxième type de message permet d'informer les autres mémoires cache qu'un bloc donné vient de passer de l'état "invalide" à l'état "partagé". Si l'une d'entre elles détient ce bloc dans l'état "exclusif", elle doit changer son état pour "partagé". Ces messages sont envoyés par le client matériel.
- un modèle de client en matériel : il est responsable de l'envoi des messages d'invalidation et de notification. Lorsqu'un bloc de la mémoire cache associée au client passe à l'état "exclusif" (resp. "partagé"), un message d'invalidation (resp. de notification) est envoyé à tous les processeurs. Chaque mémoire cache possède un module client distinct.
- un modèle de serveur en logiciel : il est responsable de recevoir les messages d'invalidation et de notification des clients et de passer l'information à la mémoire cache associée via une ou plusieurs écritures à une adresse désignée. Le serveur profite du fait que les processeurs utilisés possèdent plusieurs contextes d'exécution. Chaque processeur utilise ainsi un de ses contextes pour exécuter le serveur. Les communications client/serveur se font par

l'entremise du ORB, décrit en 3.2.2.

La figure 3.6 montre un exemple d'exécution du protocole dans lequel un processeur effectue la lecture d'un bloc donné. Puisque l'utilisation du protocole a pour principal objectif de faciliter le déboguage des applications distribuées, une série d'informations pertinentes sont affichées dans la console durant la simulation. Par exemple, lors de l'invalidation d'un bloc, les informations suivantes sont disponibles :

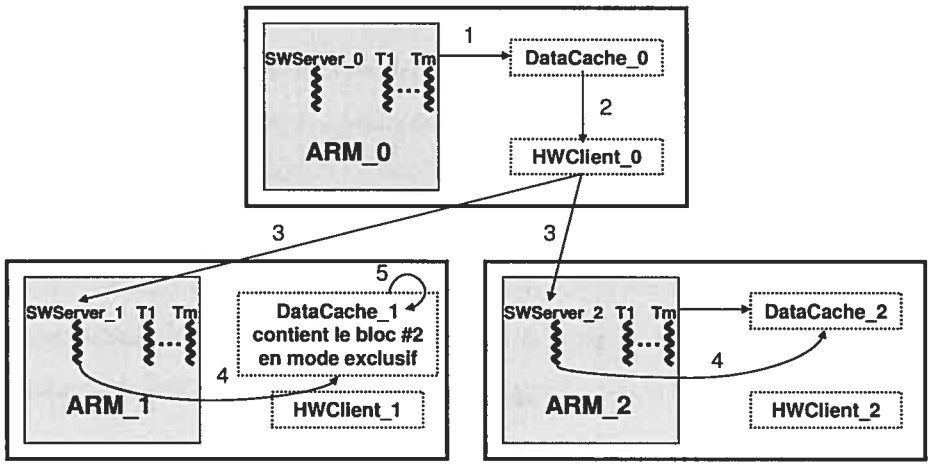
- L'adresse et le contenu du bloc invalidé.
- La ligne de code exécutée par le processeur invalidant.
- La dernière ligne de code, exécutée par le processeur qui détient la copie invalidée du bloc, y faisant référence.
- Le nom de la variable correspondant à l'adresse invalidée.

En plus d'assurer la cohérence[1] des données, le protocole proposé assure leur consistance[2] : tout changement à l'état d'un bloc est immédiatement visible par toutes les mémoires cache. Ce choix d'implémentation provoque un ralentissement de la vitesse de simulation des applications mais permet aux programmeurs de détecter tous les conflits possibles.

[1] Informellement, un système mémoire est cohérent si toute lecture d'un élément de donnée renvoie la valeur la plus récemment écrite de la donnée. (source : [48])
[2] La consistance détermine quand une valeur écrite sera renvoyée par une lecture. (source : [48])

6

```
ARM_1 now shares a block at address 0x8d178

Block data is
int 0: 577960   int 1: 577984   int 2: 578136   int 3: 578160

Invalidating context: Thread 2 of ARM_0 is executing /home/mpeg4_coherent/mpeg4.c:581
Invalidated context last write: Thread 2 of ARM_1 is executing /home/mpeg4_coherent/mpeg4c.c:698
Invalidated context last access: Thread 2 of ARM_1 is executing /home/mpeg4_coherent/mpeg4c.c:698
Address 0x8d178 is used by variable: MB_3
```



1.	ARM_0 fait une lecture du bloc #2.
2.	DataCache_0 appelle HWClient_0.
3.	HWClient_0 envoie un message de notification aux SWServers 1 et 2.
4.	SWServers 1 et 2 avertissent leur DataCache associée.
5.	DataCache_1 modifie l'état du bloc #2 à "partagé".
6.	L'information est imprimée dans la console.

Figure 3.6: Étapes d'exécution du protocole de cohérence des mémoires cache.

Chapitre 4 : Optimisation mémoire d'une application d'encodage vidéo MPEG-4 pour un réseau sur puce

Cette étude de cas a été l'objet d'un stage de 4 mois chez STMicroelectronics Ottawa. Nous avons optimisé la taille de la mémoire requise par une application d'encodage vidéo MPEG-4 [19].

La norme MPEG-4 est principalement utilisée pour le contenu vidéo sur Internet (*streaming video*), pour les visiophones (*videophones*) et pour la télévision numérique. L'encodage d'un fichier vidéo s'effectue ainsi : la première trame est divisée en macroblocks (voir figure 4.1 pour la définition d'un macroblock). Puis, une série de fonctions d'encodage sont appliquées sur tous les macroblocks de la trame. L'encodeur recommence les mêmes opérations pour toutes les trames, pour produire un fichier vidéo compressé qui sera éventuellement lu et affiché à l'écran par un décodeur MPEG-4.

La norme MPEG-4 décrit plusieurs profils d'encodeur : *Simple*, *Simple Scalable*, *Core*, *Main*, etc. L'application que nous utilisons implémente le profil *Simple*, présenté sommairement dans la figure 4.2. Il propose deux modes d'encodage : le mode *intra* et le mode *inter*. L'encodage d'une trame en mode *intra* utilise le principe de la redondance spatiale. Les étapes DCT (*Discrete Cosine Transform*), quantisation et VLC (*Variable Length Coding*) sont alors utilisées. En plus de ce principe, le mode *inter* exploite celui de la redondance temporelle. L'étape d'estimation de mouvement exécute un algorithme d'association de blocs (*block matching*) qui utilise les informations contenues dans la trame précédente. Ainsi, les parties de la trame courante qui sont présentes dans la trame précédente ne sont pas encodées une deuxième fois. Ce mode utilise également les étapes compensation de mouvement, quantisation inverse et IDCT (*Inverse Discrete Cosine Transform*). Ces trois étapes forment le corps d'un décodeur MPEG-4, utilisé par l'encodeur pour produire les trames prédites, requises par l'étape d'estimation de mouvement. En

effet, puisque les décodeurs ne possèdent pas les images sources, l'encodeur ne peut baser sa recherche de blocs temporellement redondants sur la trame précédente non encodée. Il doit donc décoder chaque trame encodée. Ce sont ces trames, nommées trames prédites, qui sont utilisées par l'encodeur lors de l'estimation de mouvement.

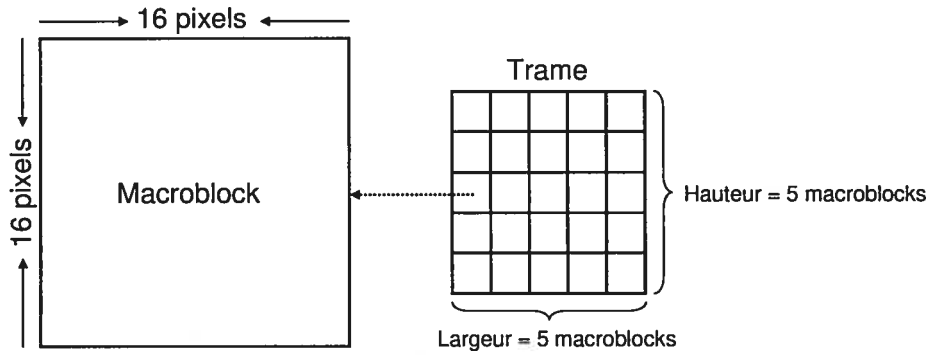


Figure 4.1 : Un macroblock est défini comme un bloc de 16×16 pixels.

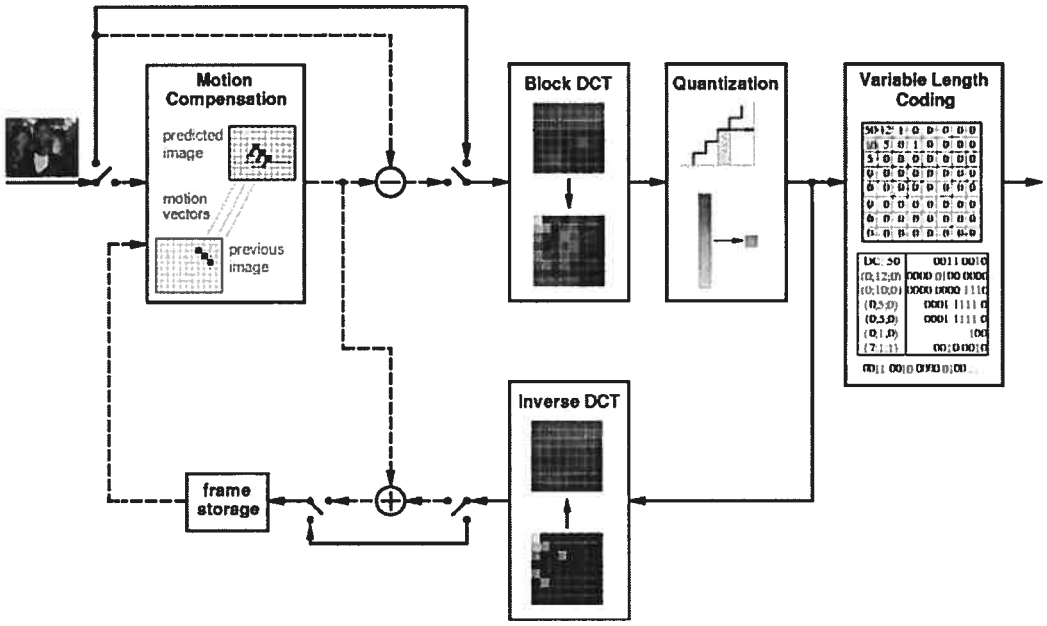


Figure 4.2 : Étapes nécessaires à l'encodage d'un macroblock (image tirée de [50]).

4.1 Présentation de l'application

L'application est écrite en C++. Elle contient plusieurs structures de données dont la structure de données MACROBLOCK. Cette dernière contient toute l'information relative à l'encodage d'un macroblock. Les trois composantes qui occupent la majorité de l'espace mémoire requis par l'encodeur sont donc : le tableau de MACROBLOCK, la trame courante et la trame précédente. La taille des deux dernières ne pouvant être réduite, nous tenterons de réduire celle du tableau de MACROBLOCK.

La version de l'encodeur que nous avons modifiée offrait un haut degré de parallélisme. Par contre, elle nécessitait plusieurs dizaines de Mo de mémoire. Nous l'avons optimisée afin de réduire ce nombre tout en maintenant un haut degré de parallélisme.

La version non optimisée exploite au maximum le parallélisme contenu dans l'algorithme d'encodage. Par exemple, puisque l'étape DCT peut être exécutée sur tous les macroblocks de la trame simultanément, cette version lance autant de fils d'exécution qu'il y a de macroblocks dans la trame. Chaque fil exécute la DCT sur un macroblock de la trame. Cette façon de faire requiert beaucoup de mémoire puisque toute l'information sur tous les macroblocks doit être conservée en mémoire.

Nous avons donc ajouté un pipeline fonctionnel dont chaque étage constitue une étape dans l'encodage d'un macroblock. En limitant le nombre de macroblocks présents dans le pipeline, il est possible de limiter la taille de la mémoire requise par l'encodeur.

Nous avons également fait d'autres optimisations du code de l'application, ciblant à la fois son usage de la mémoire et ses performances. Elles sont décrites dans la section 4.4. La section 4.5 chiffre les gains obtenus par l'ajout du pipeline fonctionnel et par les optimisations du code.

Finalement, nous avons testé plusieurs configurations de plateformes simulées et analysé les résultats obtenus. Nous avons fait varier le nombre de processeurs, de contextes d'exécution, la taille des mémoires cache, etc. Nous avons aussi ajouté une mémoire cache associée au coprocesseur connecté au bus interne ainsi qu'une mémoire bloc-notes qui gère la prélecture des pixels.

4.2 Objectifs spécifiques

- Modifier un encodeur MPEG-4 parallèle, utilisant beaucoup d'espace mémoire (~ 40 Mo), afin qu'il n'utilise plus que quelques centaines de Ko, ceci dans le but de le rendre éventuellement compatible avec les systèmes embarqués actuels. Pour ce faire, implémenter un pipeline fonctionnel qui s'inspire des pipelines matériels utilisés pour ce genre d'applications.
- Conserver un haut degré de parallélisme afin d'exploiter au maximum tous les contextes d'exécution disponibles sur la plateforme simulée.
- Utiliser les environnements de simulation de réseaux sur puce développés chez STMicroelectronics pour faire de l'exploration architecturale et trouver quelle configuration de plateforme simulée convient le mieux à l'application.
- Trouver des moyens de réduire le trafic sur les bus interne et externe (voir figure 3.3) et d'accroître les performances de l'application en étudiant la hiérarchie mémoire (mémoires cache, mémoires bloc-notes, etc.). Nous savons que les accès à la mémoire externe sont très coûteux en termes de temps et de consommation d'énergie, c'est pourquoi les concepteurs de systèmes embarqués doivent accorder une grande importance à la hiérarchie mémoire et aux réseaux d'interconnexion.

4.3 Modèle de programmation avec pipeline fonctionnel

La norme MPEG-4 décrit l'encodage d'une trame comme une série de fonctions à appliquer sur tous les macroblocks. L'ordre dans lequel les macroblocks sont encodés est préétabli. Il doit respecter les dépendances de la première étape d'encodage : l'estimation de mouvement. Cette étape ne peut être exécutée sur le macroblock courant avant que les macroblocks situés à gauche, en haut à gauche, en haut et en haut à droite ne l'aient eux-mêmes exécutée[1] (voir figure 4.3). Ceci limite le degré de parallélisme de l'application. Il est toutefois possible d'exécuter simultanément certaines étapes d'encodage, comme la DCT, sur tous les macroblocks de la trame.

Ainsi, on peut créer un encodeur hautement parallèle exploitant au maximum le parallélisme présent dans chaque étape d'encodage. En supposant la taille des trames égale à 640×480 pixels, cette version de l'encodeur nécessite environ 40 Mo de mémoire pour s'exécuter. Il est donc impensable de l'utiliser sur un réseau sur puce dont la taille de la mémoire est limitée.

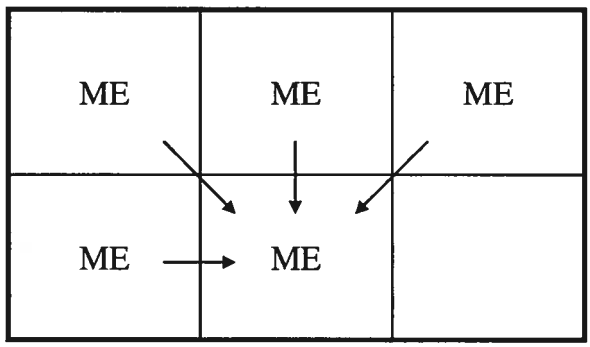


Figure 4.3 : Dépendances de l'estimation de mouvement (les carrés représentent les macroblocks et les flèches représentent les dépendances entre les calculs).

[1] Ceci est vrai pour l'algorithme utilisé, qui a été breveté par STMicroelectronics. Les algorithmes traditionnels d'estimation de mouvement n'ont pas nécessairement les mêmes dépendances.

Afin de diminuer la taille de la mémoire requise par l'encodeur, nous avons créé un pipeline fonctionnel dont chaque étage correspond à une étape d'encodage. Il possède quatre étages : estimation de mouvement (ME), compensation de mouvement et transformée en cosinus discrète bidimensionnelle (DCT), quantisation, codage à longueur variable et quantisation inverse (Q/IQ) et transformée inverse en cosinus discrète bidimensionnelle (IDCT).

Nous définissons un *cycle* comme étant l'exécution simultanée des quatre étages du pipeline. Au début d'un cycle, un appel à la fonction *fork* crée un fil d'exécution pour chaque macroblock présent dans le pipeline. Lorsque tous les fils d'exécution ont terminé leur exécution, les macroblocks sont envoyés vers le prochain étage et un autre cycle peut débuter. Ainsi, l'encodage d'un macroblock nécessite quatre cycles. Les figures 4.4 (a) et (b) montrent l'état d'avancement des macroblocks dans le pipeline aux cycles 6 et 7, respectivement. Des représentations schématiques de l'état du pipeline pour ces mêmes cycles sont données dans les figures 4.4 (c) et (d). Durant le sixième cycle, trois ME, trois DCT, deux Q/IQ et deux IDCT sont exécutés simultanément. Au début du septième cycle, quatre macroblocks entrent dans le pipeline. Ceux à l'intérieur sont envoyés vers le prochain étage. Le nombre d'unités fonctionnelles logiques du pipeline varie en fonction du nombre de macroblocks entrant dans le pipeline.

Puisque tous les macroblocks doivent satisfaire certaines conditions (voir figure 4.3) avant de pouvoir entrer dans le premier étage du pipeline, le degré de parallélisme atteint avec cette technique est moins élevé que celui atteint par la version hautement parallèle décrite dans la section 4.1. Cette dernière, bien que soumise aux mêmes dépendances pour l'étape ME, peut par contre profiter du fait que l'étape DCT peut s'effectuer sur tous les macroblocks en même temps, ce que le pipeline fonctionnel ne permet pas. Cependant, l'utilisation du pipeline nous permet de procéder à une série d'optimisations mémoire qui fera passer la taille de la mémoire requise de 40 Mo à 1 Mo.

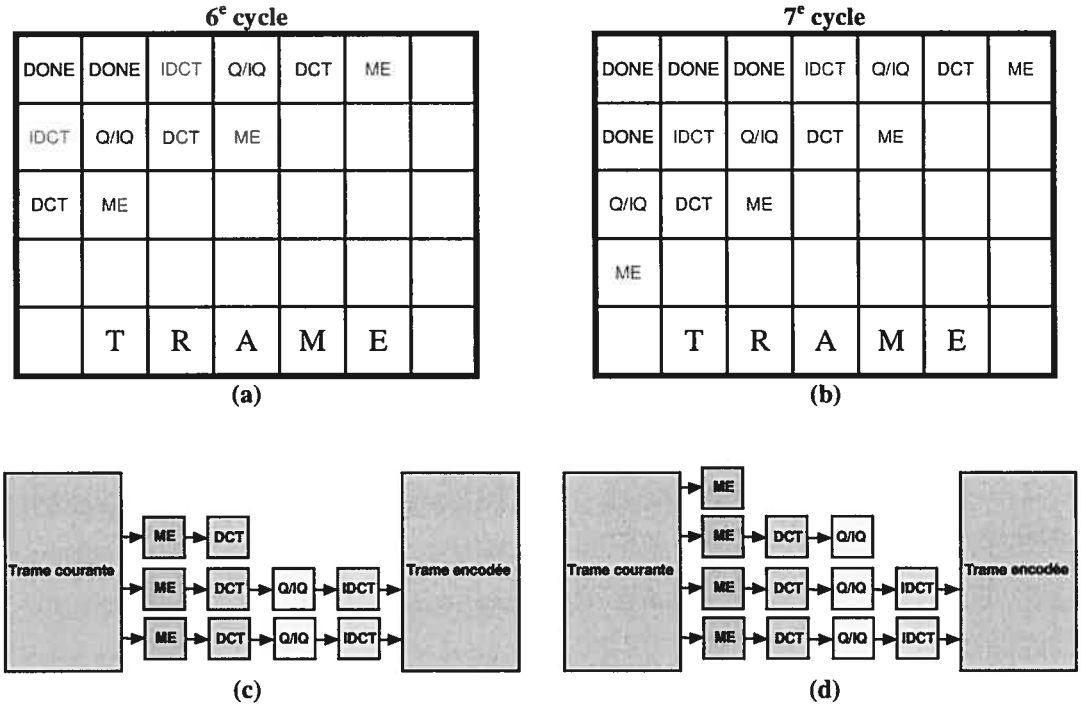


Figure 4.4 : Représentation du processus d'encodage aux cycles six et sept. (a, b) État d'avancement des macroblocks dans le pipeline; (c, d) État du pipeline. Chaque carré représente une unité fonctionnelle logique du pipeline.

La première optimisation découle de la diminution du nombre maximal de macroblocks exécutant simultanément une ou l'autre des étapes d'encodage. Dans la première version, à un certain moment, tous les 1200 macroblocks de la trame exécutent la DCT. Il faut donc créer 1200 instances de la structure de données MACROBLOCK (~ 15 Mo) afin de garder l'information sur tous les macroblocks. Dans la seconde version, le nombre maximal de macroblocks présents dans le pipeline, m, est donné par la valeur maximale du nombre de macroblocks présents dans le pipeline au cycle k, n_k :

$$m = \max\left\{n_k \mid 0 \le k \le 2\cdot(M-1)+(N-1)+(P-1)\right\} \qquad (1)$$

M : hauteur de la trame, en macroblocks.

N : Largeur de la trame, en macroblocks.

P : Nombre d'étages dans le pipeline (quatre dans notre cas).

Les valeurs de n dépendent du nombre de macroblocks qui entrent dans le pipeline, c'est-à-dire qui exécutent la première étape d'encodage, au cycle k, k-1, k-2, ..., k-P+1. Elles sont calculées comme suit :

$$n_k = \sum_{i=k-P+1}^{k} n_{entrant_i} \tag{2}$$

Les valeurs de $n_{entrant}$ dépendent de la taille de la trame et des dépendances mises en évidence dans la figure 4.3. La démarche complète avant d'arriver à l'équation (3) donnant la valeur de $n_{entrant}$ pour chaque cycle k est présentée dans la section 4.3.1.

$$n_{entrant_k} = 1 + \min\left(M-1, \left\lfloor \frac{k}{2} \right\rfloor\right) - \max\left(0, \left\lceil \frac{k-N+1}{2} \right\rceil\right) \tag{3}$$

En supposant la taille des trames égale à 640×480 pixels (M = 30 et N = 40), on obtient m = 80. Il faut donc créer 80 instances de la structure de données MACROBLOCK (~ 1 Mo) pour exécuter la version avec pipeline. La figure 4.5 montre l'évolution de n au cours du temps. 101 cycles sont nécessaires pour l'encodage d'une trame de taille 640×480 avec 80 macroblocks alloués. La moyenne arithmétique de n est 47,52.

Il est possible de réduire davantage le nombre m et par le fait même la taille de la mémoire requise par l'encodeur. En effet, en diminuant la hauteur de la trame (le paramètre M des équations (1) et (3)), on réduit m. Nous pouvons donc décider, par exemple, de diviser les trames dont la taille originale est 640×480 en trois fenêtres horizontales (que l'on nommera *fenêtres d'encodage*) de taille 640×160. Ce faisant, nous trouvons une nouvelle valeur pour m : 40. Naturellement, moins de macroblocks à l'intérieur du pipeline signifie moins de parallélisme à exploiter. Les fenêtres d'encodage sont encodées séquentiellement, à partir de celle du haut (voir figure 4.6).

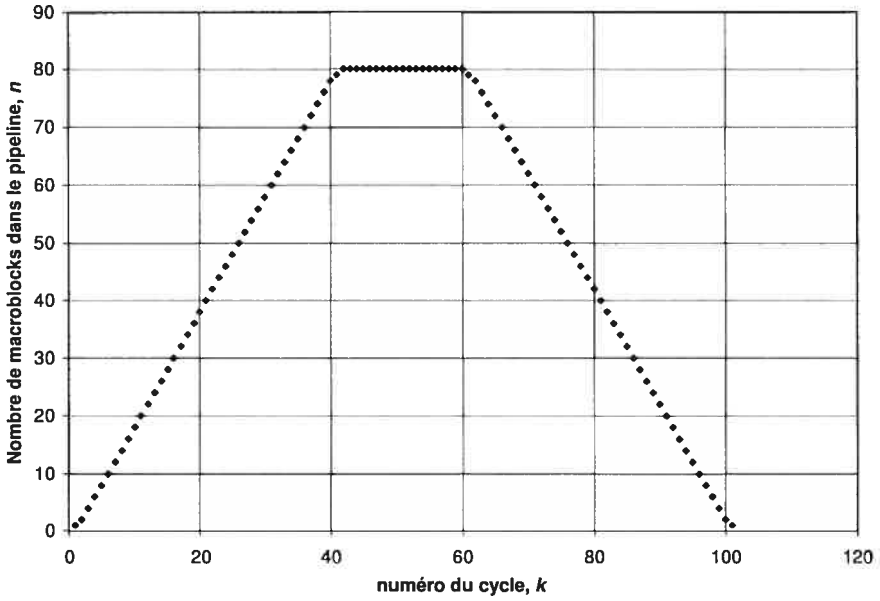


Figure 4.5 : Le nombre de macroblocks à l'intérieur du pipeline, *n*, pour chaque cycle *k*.

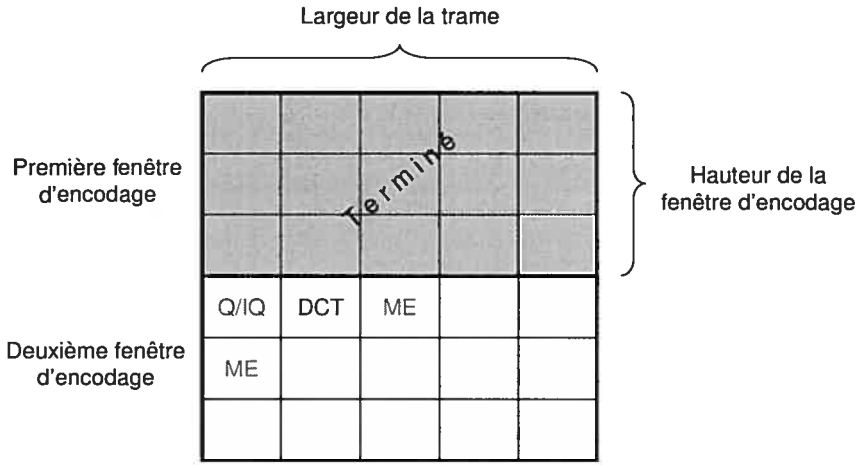


Figure 4.6 : Représentation des fenêtres d'encodage (ici, on débute l'encodage de la deuxième fenêtre).

La prochaine section décrit un modèle mathématique qui nous permettra de modifier la boucle d'encodage d'un encodeur séquentiel pour arriver à la version avec pipeline fonctionnel décrite plus haut. Suivront les sections plus pratiques sur les détails de l'implémentation de la version avec pipeline et ses conséquences.

4.3.1 Formalisme mathématique

La transformation d'un code séquentiel en version parallèle avec pipeline fonctionnel peut être décrite par un modèle mathématique. En appliquant une transformation unimodulaire aux bornes de la boucle d'encodage séquentielle, on parallélise la boucle interne. La boucle séquentielle simplifiée de l'encodeur est exprimée ainsi :

```
for i=0 to M-1
  for j=0 to N-1

    ME(i, j)    = ME(i, j-1),        // dépendance 1
                  ME(i-1, j-1),      // dépendance 2
                  ME(i-1, j),        // dépendance 3
                  ME(i-1, j+1)       // dépendance 4

    DCT(i, j)   = ME(i, j)

    Q(i, j)     = DCT(i, j),
                  Q(i, j-1),          // = dépendance 1
                  Q(i-1, j-1),        // = dépendance 2
                  Q(i-1, j),          // = dépendance 3
                  Q(i-1, j+1)         // = dépendance 4

    IDCT(i, j) = Q(i, j)

  end loop
end loop
```

Ici, le symbole " = " signifie "dépend de". M et N sont respectivement la hauteur et la largeur des trames, en macroblocks. Nous pouvons extraire de ce code quatre vecteurs de dépendance :

$$\vec{d}_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \vec{d}_3 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$
$$\vec{d}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \vec{d}_4 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \tag{4}$$

À partir de ces vecteurs, nous trouvons la transformation unimodulaire suivante, qui permet de paralléliser la boucle j du code séquentiel :

$$\begin{aligned} &\mathrm{T}: \mathrm{Z}^2 \rightarrow \mathrm{Z}^2 \\ &\begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} i' \\ j' \end{pmatrix} \\ &\text{t.q.} \quad \mathrm{T} = \begin{pmatrix} t_1 & t_2 \\ t_3 & t_4 \end{pmatrix} \\ &\text{et} \quad \begin{pmatrix} i' \\ j' \end{pmatrix} = \mathrm{T} \begin{pmatrix} i \\ j \end{pmatrix} \end{aligned} \tag{5}$$

Puisque $i' = t_1 \cdot i + t_2 \cdot j$, on trouve que :

$$\begin{aligned} i' &\geq t_1 \cdot \min(i) + t_2 \cdot \min(j) = t_1 \cdot 0 + t_2 \cdot 0 = 0 \\ i' &\leq t_1 \cdot \max(i) + t_2 \cdot \max(j) = t_1 \cdot (M-1) + t_2 \cdot (N-1) \end{aligned} \tag{6}$$

On trouve un résultat similaire pour j' :

$$\begin{aligned} j' &\geq t_3 \cdot \min(i) + t_4 \cdot \min(j) = t_3 \cdot 0 + t_4 \cdot 0 = 0 \\ j' &\leq t_3 \cdot \max(i) + t_4 \cdot \max(j) = t_3 \cdot (M-1) + t_4 \cdot (N-1) \end{aligned} \tag{7}$$

La matrice T est une matrice unimodulaire, c'est-à-dire carrée, à coefficients entiers et dont la valeur absolue du déterminant est égale à 1. Cette matrice est obtenue en résolvant le système d'équations suivant :

$$\begin{pmatrix} t_1 & t_2 \\ t_3 & t_4 \end{pmatrix} \vec{d}_i \geq 0 \qquad \text{pour } i = 0 \text{ jusqu'à } 4 \tag{8}$$

où "≥ 0" signifie "lexicographiquement positif ou nul". Un vecteur $v = (v_1, v_2, \ldots, v_n)$ est lexicographiquement positif si et seulement si :

$$\begin{aligned} &\exists i \in \mathrm{N} \text{ t.q.} \\ &1 \leq i \leq n, v_i > 0 \text{ et} \\ &v_j = 0 \; \forall j \in \mathrm{N} \text{ t.q.} 1 \leq j < i \end{aligned}$$

En remplaçant d_i dans (8) par les vecteurs trouvés en (4), on obtient :

$$\begin{pmatrix} t_1 & t_2 \\ t_3 & t_4 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} t_2 \\ t_4 \end{pmatrix} \geq 0 \qquad \begin{pmatrix} t_1 & t_2 \\ t_3 & t_4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} t_1 + t_2 \\ t_3 + t_4 \end{pmatrix} \geq 0$$

$$\begin{pmatrix} t_1 & t_2 \\ t_3 & t_4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} t_1 \\ t_3 \end{pmatrix} \geq 0 \qquad \begin{pmatrix} t_1 & t_2 \\ t_3 & t_4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} t_1 - t_2 \\ t_3 - t_4 \end{pmatrix} \geq 0 \tag{9}$$

Ce système d'équations possède une infinité de solutions. Rappelons que l'on cherche à paralléliser la boucle interne j', c'est-à-dire que l'on cherche deux fonctions f_1 et f_2 telles que l'on pourra écrire :

```
for i'=0 to t1*(M-1) + t2*(N-1)
  forall j'=f1(i') to f2(i')
      ...
  end loop
end loop
```

Toutes les itérations de la boucle *forall* s'exécutent simultanément. Ainsi, pour que la transformation T soit valide, on doit ajouter la condition suivante :

$$\begin{aligned} &\text{soient } x_1, x_2 \text{ t.q.} \\ &\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \mathrm{T} \cdot \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} \\ &\text{alors } x_1 > 0 \end{aligned} \tag{10}$$

où le vecteur d est un vecteur dépendance trouvé en (4). Cette condition est plus forte que la condition (8) puisqu'elle force tout calcul dépendant à s'effectuer lors d'itérations distinctes de la boucle externe i.

De (5), on trouve :

$$\begin{aligned} &i' = t_1 \cdot i + t_2 \cdot j \text{ et } j' = t_3 \cdot i + t_4 \cdot j \Rightarrow j = \frac{1}{t_2} \cdot \left(i' - t_1 \cdot \left(\frac{j' - t_4 \cdot j}{t_3} \right) \right) \\ &\text{On sait que } 0 \le j \le N-1 \text{ et donc :} \\ &0 \le \frac{1}{t_2} \cdot \left(i' - t_1 \cdot \left(\frac{j' - t_4 \cdot j}{t_3} \right) \right) \le N-1 \Rightarrow \frac{t_3}{t_1} \cdot (i' - t_2 \cdot (N-1)) + (t_4 \cdot j) \le j' \le \frac{t_3 \cdot i'}{t_1} + (t_4 \cdot j) \end{aligned} \tag{11}$$

Pour obtenir des fonctions f_1 et f_2 qui ne dépendent que de i', on pose : $t_4 = 0$. De (7) et (11), on obtient :

$$\begin{aligned} f_1(i') &= \max\left(0, \frac{t_3}{t_1} \cdot (i' - t_2 \cdot (N-1))\right) \\ f_2(i') &= \min\left(t_3 \cdot (M-1), \frac{t_3 \cdot i'}{t_1}\right) \end{aligned} \tag{12}$$

Pour obtenir un résultat optimal, on cherche à minimiser le nombre d'itérations de la boucle externe i'. Les inéquations (6) montrent qu'il faut alors chercher à minimiser les coefficients t_1 et t_2.

Rappelons également que la matrice T doit satisfaire aux conditions suivantes :

- La valeur absolue de son déterminant, $t_1 \cdot t_4 - t_2 \cdot t_3$, doit être égale à 1.
- Elle doit être une solution du système d'équations (9).
- Elle doit satisfaire la condition énoncée en (10).
- On a posé : $t_4 = 0$.

La matrice T qui donne un résultat optimal est :

$$T = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix} \tag{13}$$

En remplaçant dans (5), on obtient :

$$\begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 2 \cdot i + j \\ i \end{pmatrix} \tag{14}$$

En remplaçant dans (12), on trouve :

$$\begin{aligned} f_1(i') &= \max\left(0, \left\lceil \frac{i' - (N-1)}{2} \right\rceil\right) \\ f_2(i') &= \min\left(M - 1, \left\lfloor \frac{i'}{2} \right\rfloor\right) \end{aligned} \tag{15}$$

Nous avons rajouté les fonctions plafond et plancher car les évaluations de f_1 et f_2 doivent produire des valeurs entières. La version modifiée du code :

```
for i' = 0 to 2*(M-1) + (N-1)
  forall j' = f1(i') to f2(i')
    i = j'
    j = i' - 2*j'
    ME(i, j)
    DCT(i, j)
    Q(i, j)
    IDCT(i, j)
  end loop
end loop
```

En appliquant une transformation unimodulaire aux bornes de la boucle séquentielle, nous avons exploité tout le parallélisme possible tout en respectant les dépendances (voir figure 4.7). La figure 4.8 montre le nouvel ordre d'encodage des macroblocks.

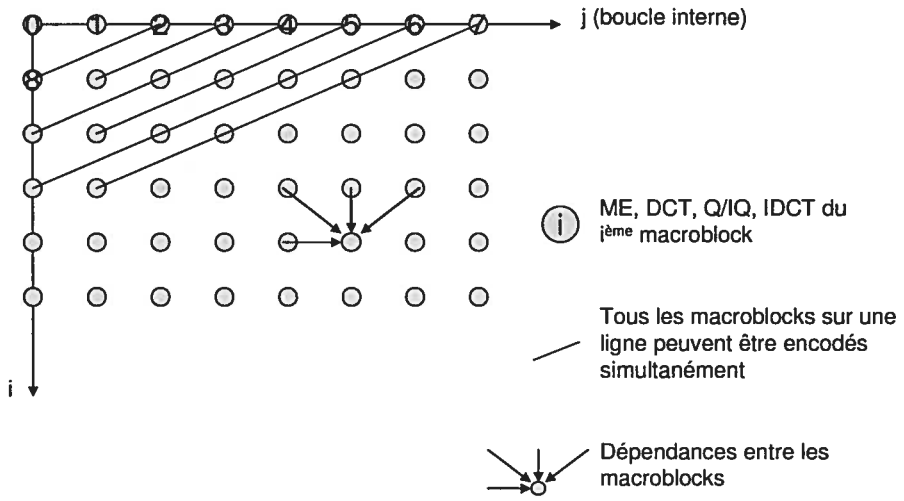


Figure 4.7 : La transformation unimodulaire rend la boucle interne parallèle.

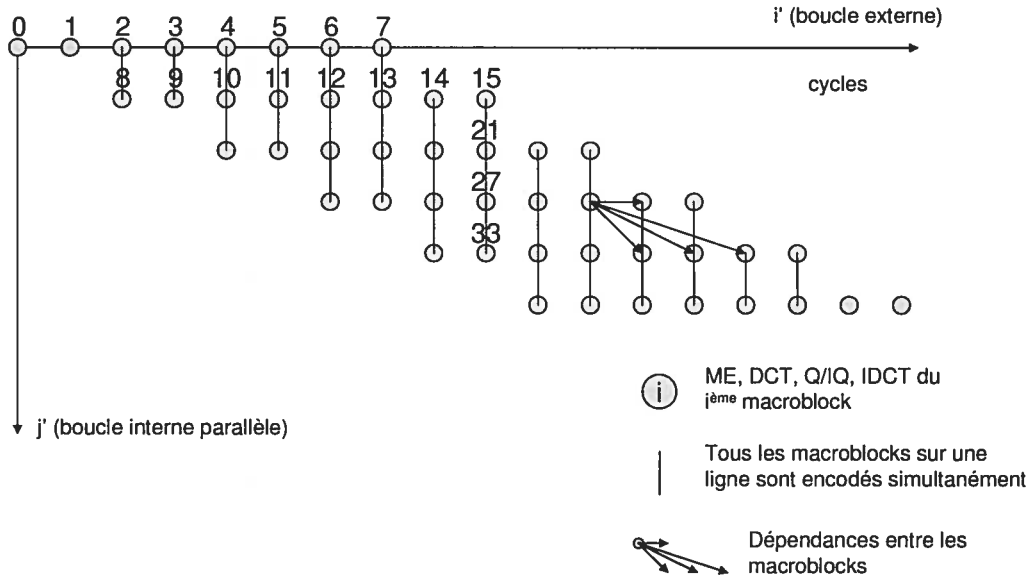


Figure 4.8 : Nouvel ordre d'encodage des macroblocks après avoir parallélisé la boucle interne.

Maintenant que nous possédons la boucle d'encodage parallélisée, nous pouvons facilement séparer les étapes d'encodage et ajouter un pipeline fonctionnel. La boucle devient :

```
for i' = 0 to 2*(M-1) + (N-1) + 3
  forall j' = f1(i') to f2(i')
    i = j'
    j = i' - 2*j'
    ME(i, j)
  end loop
  forall j' = f1(i'-1) to f2(i'-1)
    i = j'
    j = i'-1 - 2*j'
    DCT(i, j)
  end loop
  forall j' = f1(i'-2) to f2(i'-2)
    i = j'
    j = i'-2 - 2*j'
    Q(i, j)
  end loop
  forall j' = f1(i'-3) to f2(i'-3)
    i = j'
    j = i'-3 - 2*j'
    IDCT(i, j)
  end loop
end loop
```

Les figures 4.9 et 4.10 montrent l'ordre dans lequel les quatre étapes d'encodage sont effectuées pour chaque macroblock.

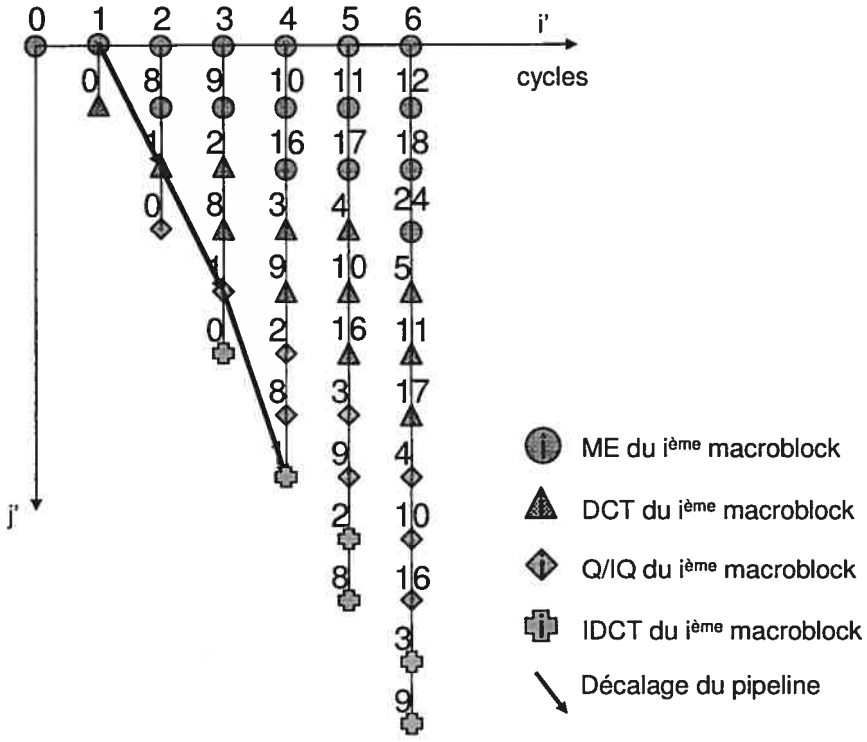


Figure 4.9 : Séparation des quatre étapes d'encodage afin d'augmenter le parallélisme.

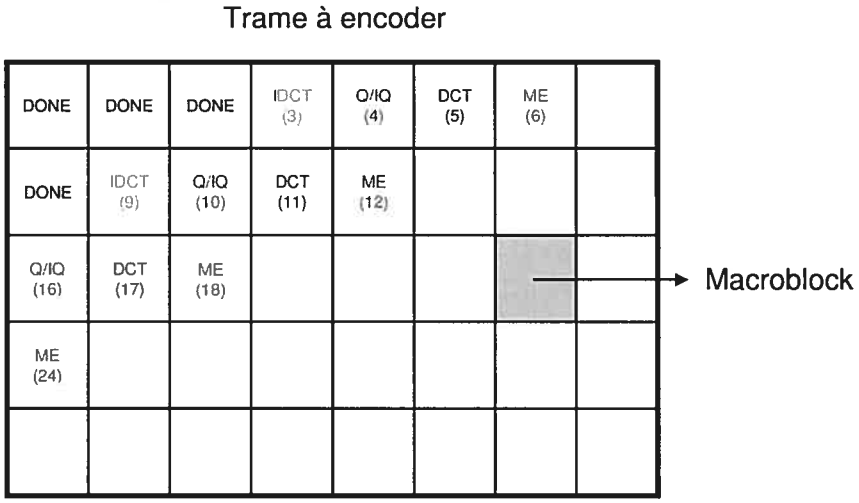


Figure 4.10 : État du pipeline durant le septième cycle d'encodage (correspond au dernier cycle représenté dans la figure 4.9).

4.3.2 Détails d'implémentation

Chaque étage du pipeline fonctionnel est modélisé à l'aide d'une classe C++. Ces classes implémentent une interface qui est utilisée par la fonction *fork* lors de la création de fils d'exécution. Chaque fil créé est envoyé vers un processeur libre qui exécute la méthode *main* de l'interface.

Afin de conserver les informations relatives à la position des macroblocks dans la trame et dans le tableau de MACROBLOCK de taille *m*, nous avons créé la classe *Position_Info*. La boucle d'encodage avec pipeline fonctionnel est la suivante :

```
Get_New_Frame();

While (!Done) {        // Une itération correspond à un cycle, ainsi
                       // que défini dans la section 4.3

  // Décalage du pipeline
  Copy(Position_Info_IDCT, Position_Info_Q);
  Copy(Position_Info_Q, Position_Info_DCT);
  Copy(Position_Info_DCT, Position_Info_ME);

  Position_Info_ME->Create(); // Entrée de nouveaux macroblocks
                              // dans le pipeline

  ME->run();       // Appel de fork pour le premier étage du pipeline
  DCT->run();
  Q->run();
  IDCT->run();
  Join();
  Done = isDone();

}
```

Classe *Position_Info* : Cette classe possède deux fonctions. Premièrement, elle est utilisée comme contenant pour conserver certaines informations sur les macroblocks présents dans le pipeline comme leur position à l'intérieur de la trame, à l'intérieur de la fenêtre d'encodage ainsi qu'à l'intérieur du tableau de MACROBLOCK. Deuxièmement, elle est utilisée pour faire entrer des nouveaux macroblocks dans le pipeline en effectuant la mise à jour des positions décrites plus haut.

Classe *ME* : Effectue le lancement des fils d'exécution qui exécutent l'estimation de mouvement. Puisque certaines informations sur les macroblocks sont utilisées par tous les étages du pipeline, nous profitons du fait que cette classe correspond au premier étage du pipeline pour calculer ces informations et les stocker dans l'instance *Position_Info_ME*. Puis, la fonction d'estimation de mouvement est exécutée.

Classe *DCT* : En plus d'effectuer le lancement des fils d'exécution qui exécutent la compensation de mouvement et la DCT, elle lance les fils d'exécution pour déterminer le type du macroblock (intra ou inter). Ainsi, les fonctions *user_mb_choice* et *dct* sont exécutées en parallèle.

Classe *Q* : Effectue le lancement des fils d'exécution qui exécutent la quantisation, le codage à longueur variable et la quantisation inverse.

Classe *IDCT* : Effectue le lancement des fils d'exécution qui exécutent les fonctions *idct* et *output*. La fonction *output* remplit le tampon de réordonnancement, ajouté afin de construire un fichier de sortie qui soit compatible à la norme MPEG-4 (voir section 4.3.3.1).

Fonction *copy* : Cette fonction copie l'information contenue à l'intérieur d'une instance de la classe *Position_Info* vers une autre instance de la même classe. Elle sert à implémenter le décalage du pipeline.

4.3.3 Conséquences sur l'application

La transformation du code de l'application due à l'introduction du modèle de programmation avec pipeline fonctionnel produit plusieurs effets de bord. Nous expliquons plus en détails trois de ceux-ci : la création d'un tampon de réordonnancement, la résolution des index des macroblocks dont dépend le calcul d'estimation de mouvement du macroblock courant et l'ajout d'un tampon conservant

l'information sur les macroblocks situés sur la dernière ligne de la fenêtre d'encodage.

4.3.3.1 Tampon de réordonnancement

Le fichier vidéo produit par l'encodeur doit satisfaire à la norme MPEG-4 qui spécifie l'ordre dans lequel les informations sur les macroblocks encodés doivent être ajoutées (du macroblock de gauche vers celui de droite, du plus haut vers le plus bas). L'examen des figures 4.4 (a) et (b) permet de voir que l'ordre de sortie des macroblocks ne respecte pas la norme. Il faut donc créer un tampon de réordonnancement dont la tâche est de garder en mémoire l'information à écrire dans le fichier en sortie. Nous n'écrivons dans le fichier que si l'information sur tous les macroblocks de la fenêtre d'encodage est dans le tampon.

À l'intérieur du processus d'encodage d'un macroblock, plusieurs appels à la fonction *append* sont effectués. Dans la version séquentielle de l'encodeur, cette fonction ajoute un ou plusieurs octets (*char*) à la fin du fichier de sortie. Dans la nouvelle version, les octets sont ajoutés à la suite de l'information déjà présente dans le tampon pour la valeur de la position passée en paramètre (voir figure 4.11).

La taille du tampon de réordonnancement dépend de la taille de la fenêtre d'encodage (voir section 4.3). Elle varie entre 30 Ko (pour des fenêtres d'encodage de 40×5 macroblocks) et 175 Ko (pour des fenêtres d'encodage de 40×30 macroblocks).

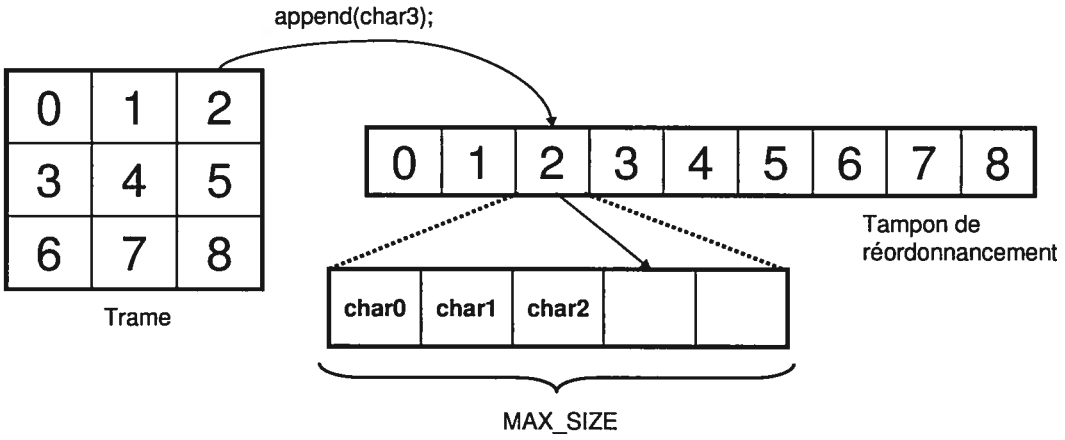


Figure 4.11 : Tampon de réordonnancement.

4.3.3.2 Résolution des index des macroblocks adjacents au macroblock courant

Comme expliqué à l'aide de la figure 4.3, plusieurs dépendances entre les calculs sur les macroblocks existent. Maintenant que nous avons réduit le nombre maximal de macroblocks présents dans le pipeline, nous devons résoudre un autre problème : où trouver l'information dont le macroblock courant a besoin sur les macroblocks situés à gauche, au-dessus à gauche, au-dessus et au-dessus à droite? Il faut trouver les quatre index à l'intérieur du tableau de MACROBLOCK qui correspondent à ces macroblocks. Une fonction permet de résoudre ce problème. Elle est appelée une fois pour chaque macroblock, avant son entrée dans le pipeline. Les index trouvés sont conservés en mémoire jusqu'à leur sortie du pipeline. La figure 4.12 montre comment l'information sur les macroblocks est stockée en mémoire. Les numéros représentent l'ordre dans lequel les macroblocks entrent dans le pipeline. Ils représentent également l'index dans le tableau de MACROBLOCK où sont conservées les informations sur le macroblock associé. Les mêmes index sont réutilisés plusieurs fois durant l'encodage d'une trame.

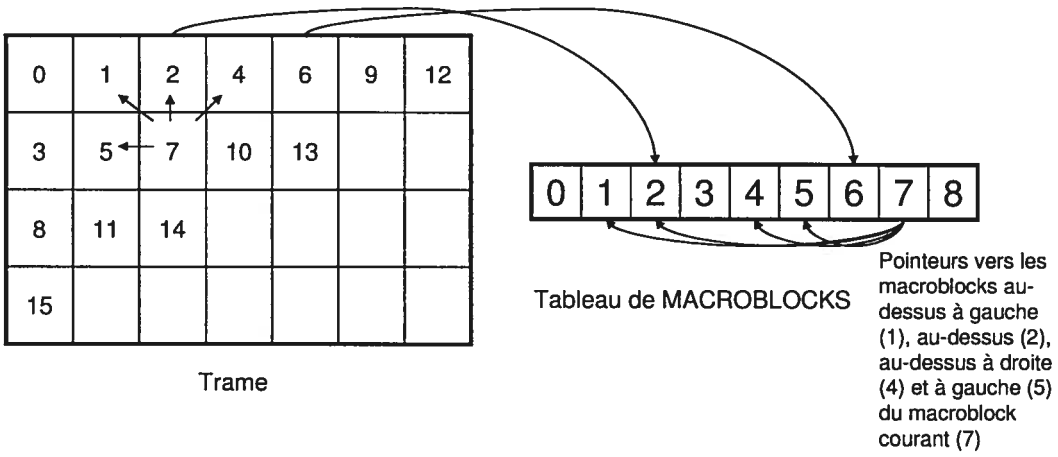


Figure 4.12 : Contenu du tableau de MACROBLOCKS.

4.3.3.3 Sauvegarde de la dernière rangée de macroblocks de la fenêtre d'encodage

Si nous choisissons d'utiliser la technique décrite dans la section 4.3 qui divise la trame à encoder en fenêtres d'encodage horizontales, nous devons s'assurer de conserver en mémoire l'information sur la fenêtre courante requise pour l'encodage de la fenêtre suivante. Afin de satisfaire aux dépendances données à la figure 4.3, il faut, pour encoder correctement les macroblocks situés sur la première ligne de la fenêtre courante, conserver l'information à propos des macroblocks situés sur la dernière ligne de la fenêtre précédente (voir figure 4.13). Nous avons donc créé une nouvelle structure de données qui est un sous-ensemble de la structure de données MACROBLOCK. Si la largeur des trames à encoder est de 40 macroblocks, 40 instances de cette nouvelle structure de données sont créées (~ 35 Ko). La création de 40 MACROBLOCK nécessiterait quant à elle approximativement 200 Ko.

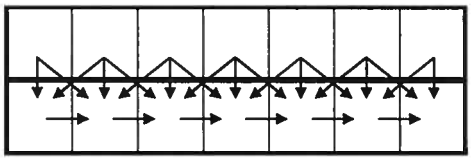


Figure 4.13 : Sauvegarde de l'information sur la dernière ligne de chaque fenêtre (les flèches représentent les dépendances entre les calculs d'estimation de mouvement sur les macroblocks).

4.3.4 Bénéfices

En utilisant le modèle de programmation avec pipeline fonctionnel et en limitant le nombre de macroblocks présents dans le pipeline à une certaine valeur flexible *m*, nous pouvons contrôler la taille de la mémoire requise par l'encodeur vidéo.

Pour m = 80, la figure 4.5 montre que l'application conserve un haut degré de parallélisme. En supposant que la taille des trames est de 640×480, chaque trame est divisée en 1200 macroblocks. 6 fils d'exécution sont lancés pour encoder un macroblock (4 étages du pipeline + MB_CHOICE + OUTPUT). Par conséquent, 7200 fils d'exécution sont lancés pour encoder une trame complète, répartis sur 101 cycles. Il y a donc en moyenne 71,29 fils d'exécution lancés à chaque cycle.

4.4 Optimisation du code de l'application

Le fait d'utiliser un pipeline fonctionnel contribue à réduire la mémoire requise par l'encodeur, mais il y a encore plus à faire si nous voulons atteindre les objectifs de départ. Dans cette section, nous présentons d'autres optimisations logicielles qui permettront d'atteindre notre objectif.

4.4.1 Une nouvelle structure de données pour représenter les macroblocks de la trame précédente

Dans la section 4.3, nous avons expliqué comment les dépendances entre les calculs sur les macroblocks de la trame courante influencent l'ordre dans lequel les macroblocks entrent dans le pipeline. Cette section traite un autre type de dépendances : celles entre les macroblocks de la trame courante et les macroblocks de la trame précédente. Afin de conserver l'information sur les 1200 macroblocks d'une trame de taille 640×480, nous n'avons d'autres choix que de créer un tableau de MACROBLOCK de taille 1200. Puisque ce tableau, qui contient l'information sur la trame précédente, est utilisé lors de l'encodage de la trame courante, nous ne pouvons réduire sa taille.

Cependant, nous avons remarqué que seulement une partie de l'information à propos des macroblocks de la trame précédente contenue dans la structure de données MACROBLOCK est utilisée lors de l'encodage de la trame courante. Ainsi, nous avons créé une nouvelle structure de données qui est un sous-ensemble de MACROBLOCK. À titre de comparaison, un tableau de 1200 MACROBLOCK occupe ~ 6090 Ko de mémoire (en tenant compte des optimisations présentées dans les sections 4.4.3 et 4.4.4) alors que le même tableau occupe ~ 70 Ko de mémoire lorsque la nouvelle structure de données est utilisée. Pour plus de détails sur la déclaration des structures de données mentionnées, voir l'annexe A.

4.4.2 Fusion des trames précédente et prédite

Tout au long du processus d'encodage de la trame courante, les macroblocks sont encodés puis décodés. La trame ainsi créée est appelée trame reconstruite ou trame prédite. Cette trame est celle que le décodeur vidéo affichera à l'écran. Puisque le décodeur ne dispose pas des images sources, il ne peut effectuer correctement l'étape de compensation de mouvement si l'encodeur n'utilise pas la trame prédite de la

trame précédente pour effectuer l'étape inverse, soit l'estimation de mouvement. Ainsi, à la fin de l'encodage de chaque trame, nous écrasons l'ancienne trame précédente avec l'information contenue dans la trame prédite.

Il est toutefois possible de fusionner les deux, c'est-à-dire d'écraser les macroblocks de la trame précédente dès que possible, sans attendre à la toute fin du processus d'encodage de la trame courante. La trame prédite est alors remplacée par un tampon de petite taille. Lorsque tous les macroblocks susceptibles d'accéder à cette partie de la trame précédente sont encodés, nous pouvons la mettre à jour avec les données présentes dans le tampon (voir figure 4.14). À l'intérieur de l'étage IDCT, un appel à *insert* écrase un macroblock dans la trame précédente avec le macroblock correspondant dans le tampon prédit. En supposant des trames de taille 640×480, nous devons créer un tampon de taille maximale égale à 45 Ko. Par comparaison, la taille de la trame prédite est 450 Ko.

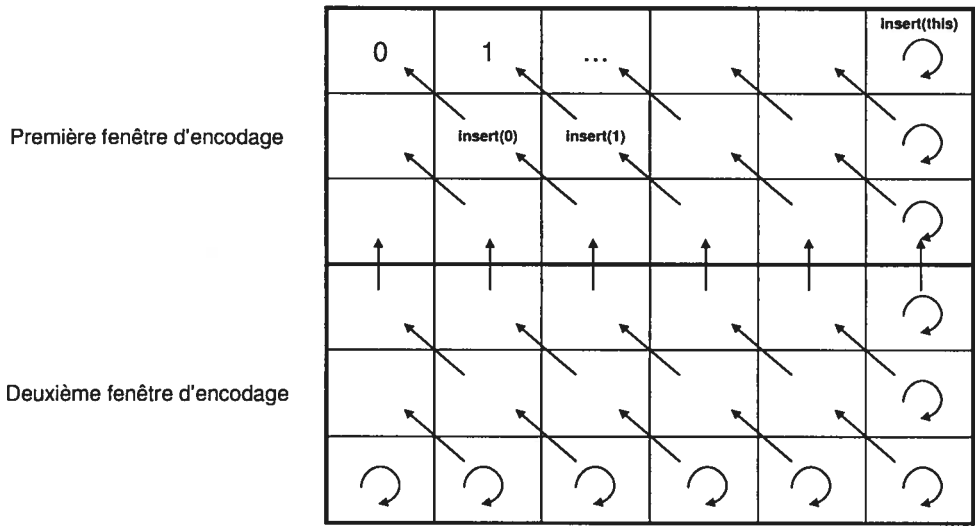


Figure 4.14 : Mise à jour de la trame précédente (les flèches représentent quels macroblocks sont insérés durant l'exécution de l'étage IDCT d'un macroblock donné).

4.4.3 Réduction de la taille de la structure de données MACROBLOCK

En divisant le processus d'encodage d'un macroblock en quatre étapes distinctes, nous avons découvert que plusieurs membres de la structure de données MACROBLOCK dont l'utilisation ne se chevauche pas pouvaient partager le même espace mémoire. Ce constat nous a permis de réduire la taille d'une instance de MACROBLOCK de 14,5 Ko à 8 Ko.

4.4.4 Remplacement de variables de type *int* par des variables de type *short*

Partout où il était possible de le faire, nous avons remplacé les entiers de 32 octets par des entiers de 16 octets. Cette optimisation a contribué à réduire la taille de la structure de données MACROBLOCK de 8 Ko à 5 Ko.

4.5 Résultats de l'optimisation logicielle

L'ajout du modèle de programmation avec pipeline fonctionnel a permis de réduire considérablement l'espace mémoire utilisé par l'encodeur vidéo MPEG-4. Les résultats des optimisations mémoire sont présentés dans la figure 4.15 (la taille de la fenêtre d'encodage considérée est 40×10) et le tableau I.

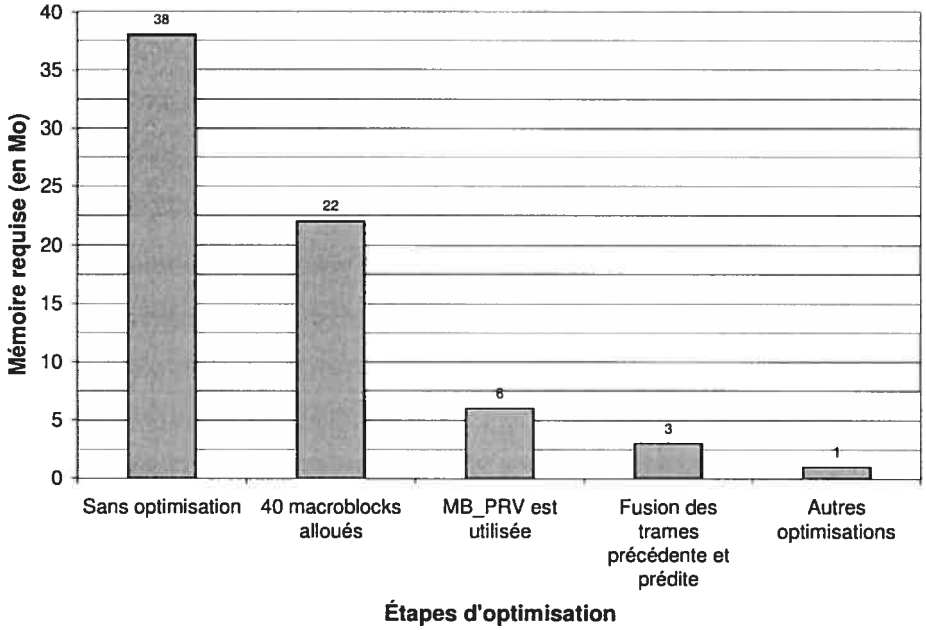


Figure 4.15 : La taille de la mémoire requise pour différentes versions de l'encodeur (les résultats sont cumulatifs).

Tableau I : Espace mémoire requis par l'application après optimisations. L'espace de travail contient les MACROBLOCK alloués, les vecteurs de mouvement, etc.

Taille de la fenêtre d'encodage (en macroblocks)	Espace de travail (en Ko)	Trame courante (en Ko)	Trame précédente (en Ko)	Totaux (en Ko)
40×6	276	90	500	881
40×10	369	150	500	1024
40×15	481	225	500	1229
40×30	543	300	500	1362

4.6 Exploration architecturale et résultats

Dans cette section, nous détaillons les expériences réalisées sur la plateforme de simulation dont la configuration est présentée dans la figure 4.16. Nous avons étudié l'impact de ses composantes sur la performance de l'application d'encodage vidéo.

En particulier, nous avons fait varier le nombre de processeurs et de contextes d'exécution disponibles sur chaque processeur, nous avons modifié et testé plusieurs configurations de mémoires cache de niveau 1 et 2, nous avons ajouté une mémoire cache partagée par les coprocesseurs ainsi qu'une mémoire bloc-notes responsable de la prélecture des pixels. À moins d'avis contraires, tous les paramètres de la plateforme sont fixés aux valeurs présentées dans le tableau II.

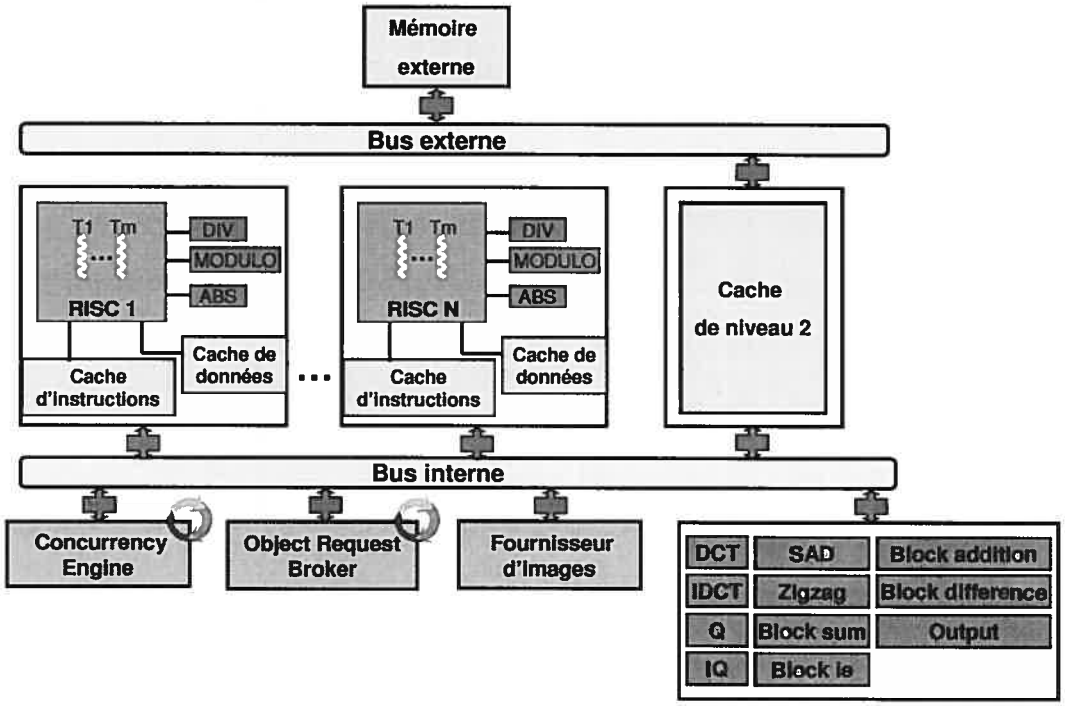


Figure 4.16 : Plateforme simulée utilisée pour tester l'application d'encodage vidéo MPEG-4.

Tableau II : Configuration de base de la plateforme simulée.

Paramètres des processeurs	
Configuration	4 processeurs à 4 contextes d'exécution
Fréquence	200 MHz
Taille de la pile (par contexte d'exécution)	4 Ko
Paramètres des bus	
Transaction sur le bus interne	30 ns
Transaction sur le bus externe	60 ns
Gigue	0 ns
Paramètres des mémoires cache de données de niveau 1	
Taille (par processeur)	4 Ko
Taille d'un bloc	4 octets
Associativité	4 voies
Politique d'écriture	Écriture simultanée
Paramètres des mémoires cache de niveau 2	
Taille	128 Ko
Taille d'un bloc	4 octets
Associativité	Correspondance directe
Politique d'écriture	Réécriture
Paramètres de l'application	
Fonction	Encodage d'une trame de taille 640×480 en mode inter
Taille des fenêtres d'encodage	40×10 macroblocks

4.6.1 Nombre de processeurs / contextes d'exécution

Nous avons d'abord étudié l'impact qu'ont le nombre de processeurs simulés et le nombre de contextes d'exécution sur le temps d'exécution de l'encodeur, le taux d'échecs des mémoires cache et le trafic généré sur le bus interne. Les résultats sont donnés dans les figures 4.17, 4.18 et 4.19. Nous remarquons que le taux d'échecs des mémoires cache ne varie pas beaucoup lorsque les configurations de la plateforme changent et ce, même si la taille des mémoires reste fixée à 16 Ko par processeur peu importe leur nombre de contextes d'exécution. Il est également intéressant de noter que le temps d'exécution décroît peu lorsque nous doublons le nombre de processeurs simulés : il diminue de 32% lorsqu'on passe d'un à deux, de 15% lorsqu'on passe de deux à quatre, de 10% lorsqu'on passe de quatre à huit et de 13% lorsqu'on passe de huit à seize. Puisque la configuration avec quatre processeurs semble constituer un bon compromis, nous l'avons utilisée pour tous les autres tests des sections suivantes.

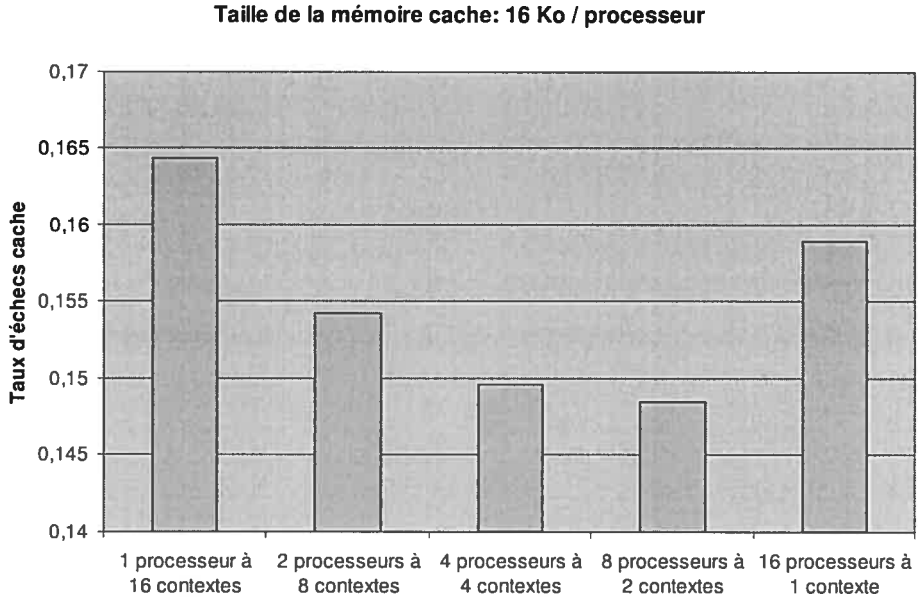


Figure 4.17 : Variations du taux d'échecs des mémoires cache pour cinq configurations possibles d'une plateforme à seize contextes d'exécution.

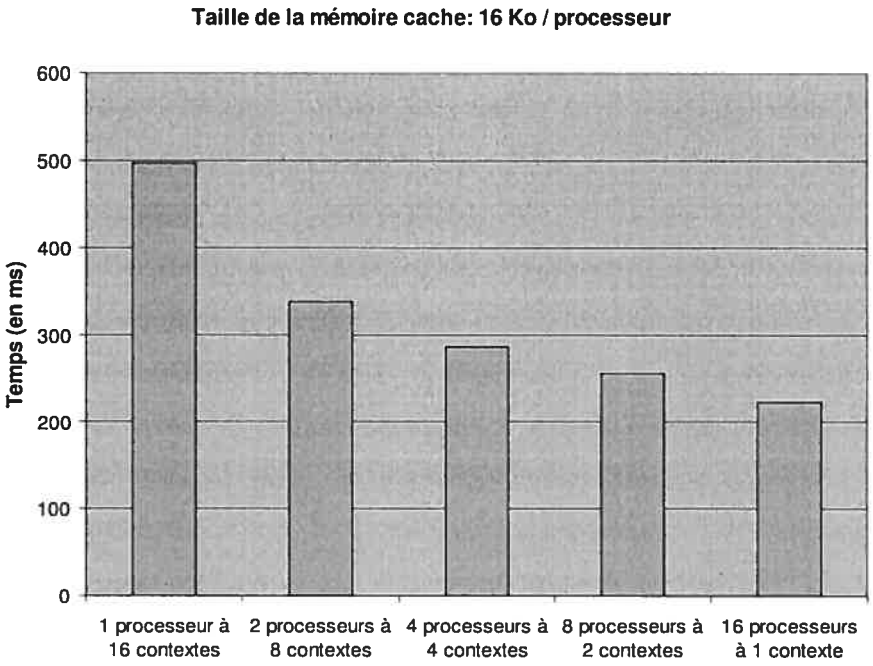


Figure 4.18 : Variations du temps d'exécution de l'application pour cinq configurations possibles d'une plateforme à seize contextes d'exécution.

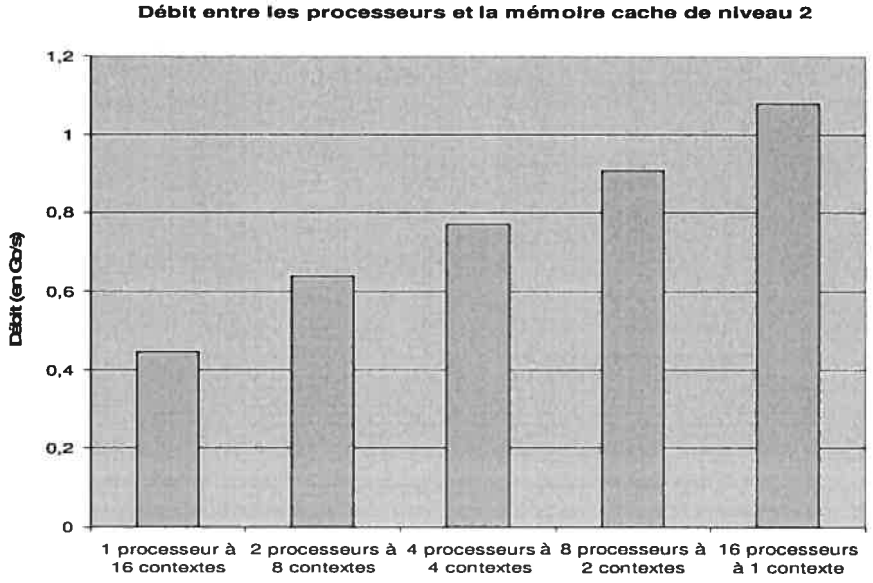


Figure 4.19 : Variations du débit des communications processeurs/mémoire cache de niveau 2 pour cinq configurations possibles d'une plateforme à seize contextes d'exécution.

4.6.2 Mémoires cache de données de niveau 1

Nous avons fait varier la taille de la mémoire cache de niveau 1 et étudié les taux d'échecs et le temps d'exécution de l'application. Les résultats sont présentés dans les figures 4.20 et 4.21. Puisque les taux d'échecs et le temps d'exécution varient très peu, nous avons choisi de fixer la taille des mémoires cache de niveau 1 à 4 Ko lors des tests des prochaines sections. Il est important de rappeler que nous avons fixé la taille des blocs à 4 octets. Une taille plus grande réduit grandement le taux d'échecs mais provoque des incohérences entre les différents niveaux de la hiérarchie mémoire.

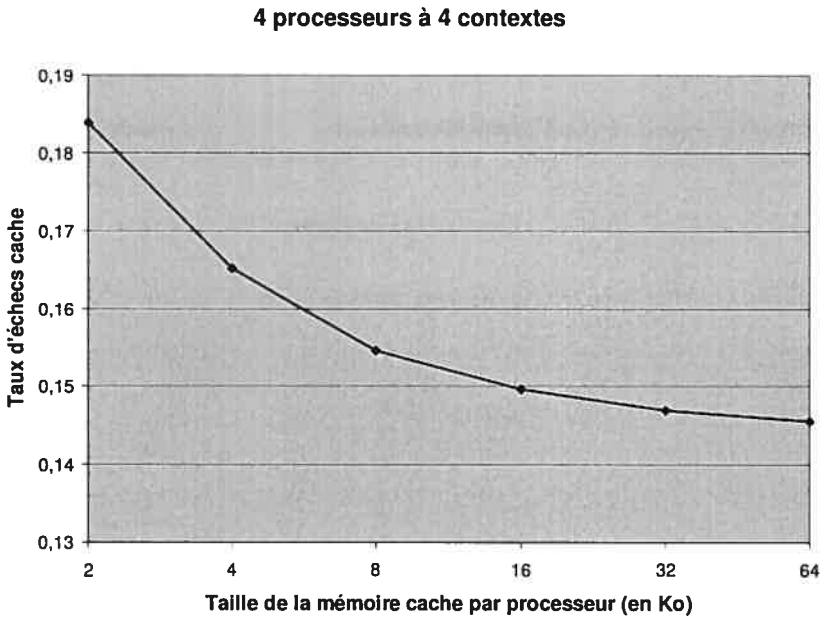


Figure 4.20 : Taux d'échecs mesurés pour différentes tailles des mémoires cache.

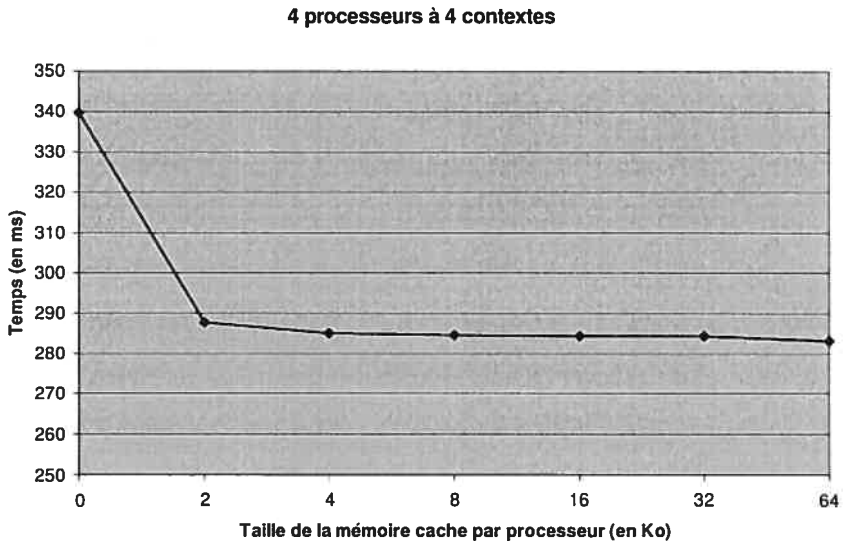


Figure 4.21 : Temps d'exécution de l'application mesurés pour différentes tailles des mémoires cache.

4.6.3 Mémoire cache de niveau 2

Nous avons étudié l'impact de la taille de la mémoire cache de niveau 2 sur la vitesse d'exécution de l'application. La figure 4.22 présente les taux d'échecs observés pour des tailles variant entre 32 et 1024 Ko et la figure 4.23 montre les temps d'exécution obtenus pour les mêmes tests. Il est à noter que les accès à la mémoire externe sont très lents (estimés à 120 ns dans nos simulations) et constituent le goulot d'étranglement du système. Par exemple, 1 085 371 accès à la mémoire externe pour un taux d'échecs de 0,11 et un temps d'exécution de 376 ms sont nécessaires lorsque la taille de la mémoire cache est 64 Ko alors que 319 547 accès pour un taux d'échecs de 0,03 et un temps d'exécution de 219 ms sont nécessaires lorsque la taille de la mémoire cache est 512 Ko. Ainsi, une diminution de 8% du taux d'échecs diminue le temps d'exécution de 42%. Nous avons fixé à 128 Ko la taille de la mémoire cache de niveau 2 pour les tests des autres sections.

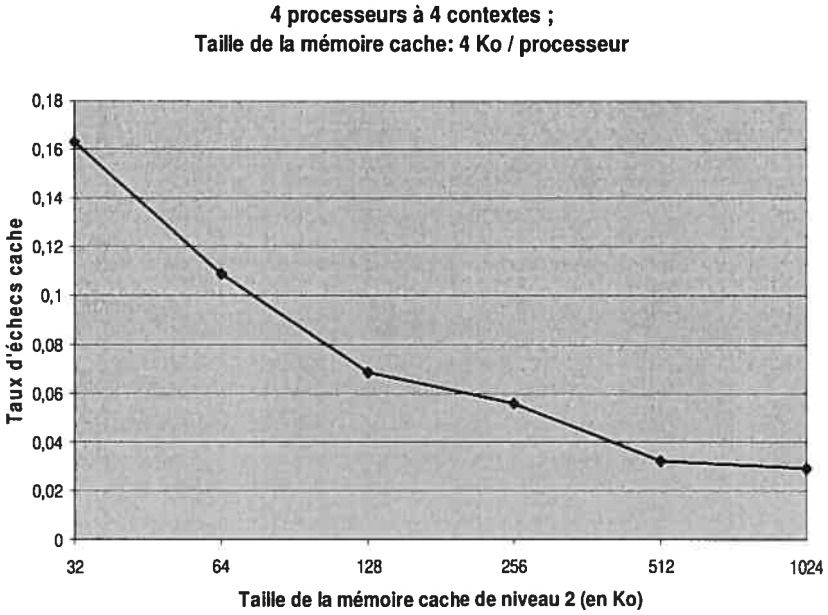


Figure 4.22 : Taux d'échecs mesurés pour différentes tailles de la mémoire cache de niveau 2.

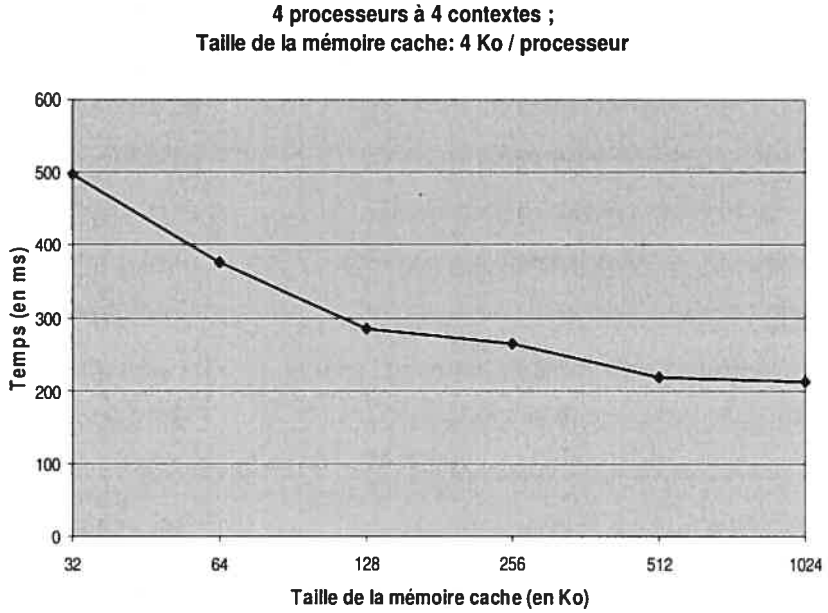


Figure 4.23 : Temps d'exécution de l'application mesurés pour différentes tailles de la mémoire cache de niveau 2.

4.6.4 Mémoire cache de données du coprocesseur

Nous avons ajouté une mémoire cache associée au coprocesseur lié au bus interne (voir figure 4.16) en espérant réduire le trafic sur le bus interne et accélérer l'exécution des fonctions matérielles. Malheureusement, des problèmes de cohérence avec les autres mémoires cache ont surgi, nous empêchant de tester le plein potentiel de cette solution. Nous avons dû limiter la taille des blocs de la mémoire cache à 4 octets et n'avons pu utiliser une mémoire de taille supérieure à 8 Ko. Nous avons également dû limiter son utilisation : certaines zones mémoire ont ainsi été configurées pour court-circuiter la mémoire cache. Ces zones ont été déterminées avec l'aide du protocole de cohérence de mémoires cache présenté dans la section 3.3. Les résultats sont tout de même présentés dans les figures 4.24 et 4.25 qui montrent respectivement le taux d'échecs et le débit des communications entre le coprocesseur et la mémoire cache de niveau 2 pour différentes tailles de la mémoire cache du coprocesseur.

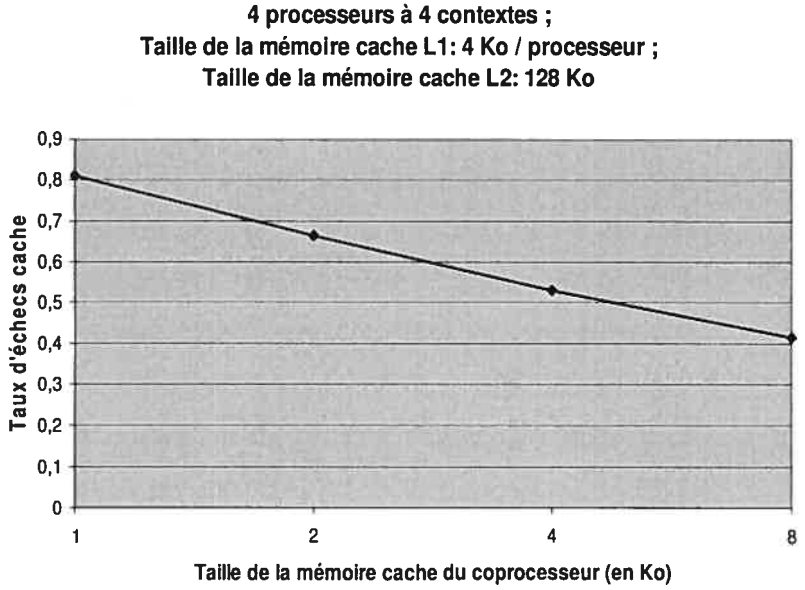


Figure 4.24 : Taux d'échecs mesurés pour différentes tailles de la mémoire cache du coprocesseur.

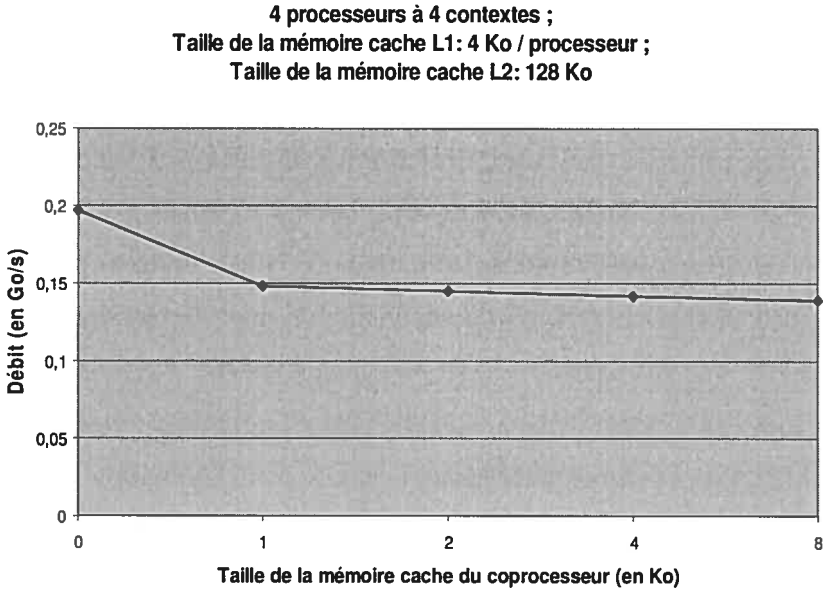


Figure 4.25 : Variations du débit des communications coprocesseur/mémoire cache de niveau 2 pour différentes tailles de la mémoire cache du coprocesseur.

4.6.5 Prélecture des données par une mémoire bloc-notes

Afin de réduire le nombre d'accès à la mémoire externe et d'accélérer l'exécution de l'application, nous avons créé un modèle de mémoire bloc-notes qui utilise le concept de la prélecture pour conserver en tout temps l'information nécessaire à l'encodage des macroblocks présents dans le pipeline fonctionnel. Pour l'instant, nous avons limité la plage d'adresses de la mémoire bloc-notes afin qu'elle contienne les parties de la trame courante associées aux macroblocks dans le pipeline. Les autres données, incluant les structures de données MACROBLOCK et la trame précédente, sont conservées dans la mémoire cache de niveau 2.

La prélecture des données se fait ainsi : lors de l'exécution d'un cycle du pipeline fonctionnel, un processeur déclenche la prélecture des données associées aux macroblocks qui entreront dans le pipeline lors du prochain cycle. Ce déclenchement s'effectue à l'aide d'écritures à une adresse désignée connue de la mémoire bloc-

notes. La prélecture s'effectue en parallèle avec l'encodage des macroblocks. Lorsque le cycle est terminé, le prochain peut commencer immédiatement (à condition que la prélecture soit terminée). Ainsi, la latence de la mémoire externe est cachée par l'ajout de cette mémoire bloc-notes dont la taille est liée au nombre maximum de macroblocks présents dans le pipeline logiciel (voir section 4.3). Pour des fenêtres d'encodage de taille 40×10 macroblocks, nous avons trouvé un maximum égal à 40, pour une taille de la mémoire bloc-notes égale à 15 Ko :

15 Ko = 40 macroblocks × (16 × 16 pixels × 1 plan + 8 × 8 pixels × 2 plans).

Le tableau III compare les versions avec et sans prélecture. Il est intéressant de noter que la version avec prélecture est 1,13 fois plus rapide que celle sans prélecture et ce, même si la prélecture ne s'effectue que pour les pixels de la trame courante. Il serait intéressant d'appliquer cette technique pour faire la prélecture de la trame précédente.

Tableau III : Impact de la prélecture sur le temps d'exécution et les accès à la mémoire externe.

	Sans prélecture	Avec prélecture
Temps d'exécution (en ms)	285	252
Temps d'exécution normalisé	1	0,88
Nombre d'accès à la mémoire externe	676 475	514 045
Nombre d'accès à la mémoire externe normalisé	1	0,76

Chapitre 5 : Optimisation d'une application de détection de contours pour un réseau sur puce

Les applications de traitement d'images utilisent généralement des boucles imbriquées afin d'exécuter une ou plusieurs fonctions sur tous les pixels. Il est possible et souvent souhaitable d'optimiser ces boucles dans le but d'améliorer le taux de succès de la mémoire cache du système ciblé par les développeurs de l'application. Ces optimisations, appelées "transformations de boucles", ont été abondamment étudiées pour des systèmes monoprocesseurs [28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40].

Dans ce chapitre, nous démontrerons qu'il est possible d'étendre les concepts de base des transformations de boucles afin de les appliquer dans un contexte multiprocesseur [43]. Pour cela, nous utiliserons une application de traitement d'images connue : la détection de contours.

Les prochaines sections présenteront l'application puis les concepts de base des transformations de boucles. Ensuite, nous utiliserons ces concepts dans le contexte d'un réseau sur puce. Enfin, nous comparerons les versions monoprocesseur et multiprocesseur en présentant une série de résultats obtenus avec l'environnement de simulation StepNP.

5.1 Présentation de l'application

L'application utilise quatre fonctions appliquées sur chaque pixel d'une image source pour produire une image qui met en évidence les contours des objets présents dans l'image source.

La première étape consiste à appliquer un filtre gaussien sur l'image source afin de réduire le bruit. Puis, les valeurs absolues des différences entre chaque pixel et ses

huit voisins sont calculées. La valeur de chaque pixel est remplacée par la valeur maximale de ces différences. Les couleurs sont ensuite inversées. La valeur finale des pixels est calculée comme suit : si au moins un de ses huit voisins a une valeur plus grande que lui, le pixel prend la valeur 0 (noir). Sinon, il prend la valeur 255 (blanc). Un exemple est donné à la figure 5.1.

La boucle de traitement de l'application séquentielle non modifiée est donnée par :

```
for i=0 to M
  for j=0 to N
    Temp1[i][j] = GaussBlurVert(IN)     // Filtre gaussien vertical

for i=0 to M
  for j=0 to N
    Temp2[i][j] = GaussBlurHor(Temp1)   // Filtre gaussien horizontal

for i=0 to M
  for j=0 to N
    Temp3[i][j] = ComputeEdges(Temp2)   // Détection de contours

for i=0 to M
  for j=0 to N
    Temp4[i][j] = Reverse(Temp3)        // Inversion des couleurs

for i=0 to M
  for j=0 to N
    OUT[i][j] = DetectRoots(Temp4)      // Image finale, en noir et
                                        // blanc
```

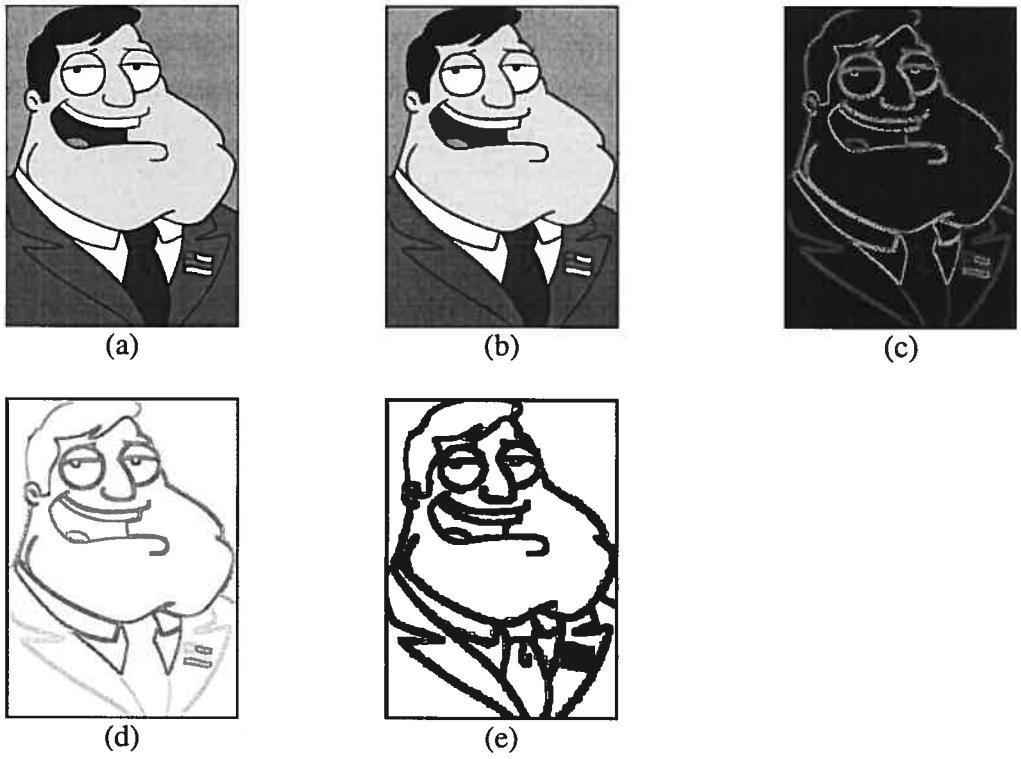


Figure 5.1 : Étapes de la détection de contours. (a) Image source (tirée du site http://www.fox.com/americandad); (b) Après l'application du filtre gaussien; (c) La valeur de chaque pixel est remplacée par la valeur maximale des différences avec ses voisins; (d) Après l'inversion de couleurs; (e) Après la conversion de l'image en noir et blanc.

5.2 Objectifs spécifiques

- Distribuer l'application et l'exécuter sur un réseau sur puce formé de plusieurs processeurs (préalable pour le prochain objectif).
- Montrer que les concepts de la fusion, de l'allocation de tampons et du pavage s'appliquent à l'application distribuée (ainsi qu'à d'autres applications satisfaisant aux conditions présentées dans la section 5.4.1). Pour ce faire, créer plusieurs versions de l'application, chacune utilisant un concept différent.
- Accélérer davantage l'exécution de l'application en utilisant des processeurs à contextes d'exécution multiples (*multithreading*) [46].

- Calculer les taux de succès des mémoires cache et les comparer avec ceux obtenus pour les versions de l'application séquentielle non modifiée et distribuée non modifiée (telle que présentée en [41]) afin de montrer la performance de la méthodologie proposée.

5.3 Concepts de base

Pour répondre à nos objectifs, nous avons dû modifier le code de l'application de détection de contours. Nous avons utilisé des techniques de transformation de boucles connues comme la fusion, le décalage, l'allocation de tampons et le pavage. Nous donnons une brève description de ces techniques dans les sections suivantes.

5.3.1 Fusion de boucles

La fusion consiste à créer un nid de boucles unique à partir de plusieurs nids de boucles séquentiels, ceci dans le but d'augmenter la localité des données. Un exemple simple est donné à la figure 5.2.

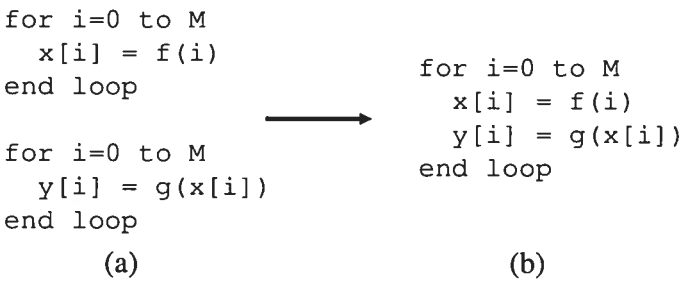


Figure 5.2 : Exemple d'une fusion de deux boucles. (a) Les deux boucles possèdent le même domaine d'itération et peuvent être fusionnées; (b) Résultat de la fusion.

De façon générale, la fusion suppose qu'il n'y ait aucun vecteur dépendance lexicographiquement négatif (voir section 4.3.1) entre les résultats produits par l'exécution d'une boucle précédente et les données requises pour exécuter la boucle courante. Si de tels vecteurs existent, il est encore possible d'appliquer la fusion, à condition cependant d'utiliser également la technique du décalage de boucles.

5.3.2 Décalage de boucles

En modifiant les bornes de la boucle fusionnée ainsi que les fonctions d'accès aux tableaux de données, nous pouvons éliminer les dépendances entre les calculs à l'intérieur de la boucle. Cette technique s'appelle le décalage de boucles (*loop shifting*). Un exemple est donné à la figure 5.3. Dans (a), nous ne pouvons appliquer la fusion car il existe un vecteur dépendance lexicographiquement négatif entre les deux boucles (le calcul de $y[i]$ nécessite la valeur $x[i+1]$). En combinant les techniques de la fusion et du décalage de boucles, nous obtenons une boucle unique qui satisfait les dépendances, montrée en (b). Nous pouvons facilement éliminer la condition sur i pour obtenir une boucle plus performante. La boucle obtenue est présentée en (c).

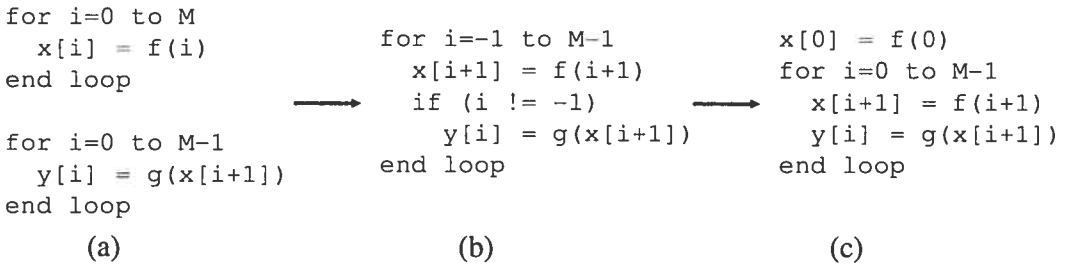


Figure 5.3 : Exemple du décalage de boucles.

5.3.3 Pavage

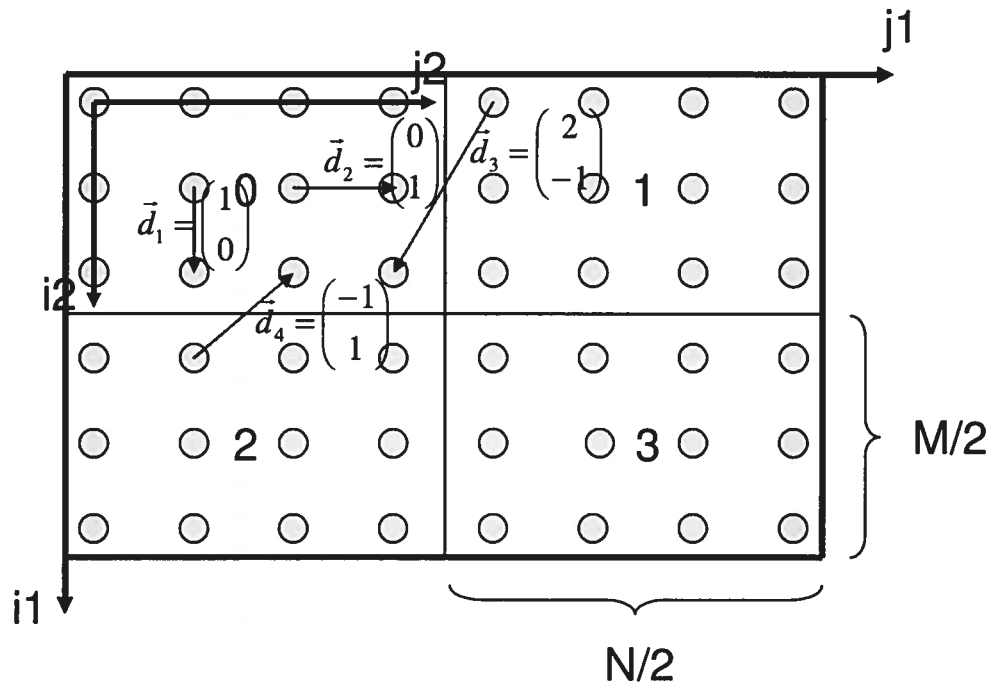
La technique du pavage est utilisée pour augmenter la localité temporelle des données. Elle consiste à modifier l'ordre des calculs à l'intérieur d'un nid de boucles en découpant l'espace d'itération en blocs de même profondeur mais de dimensions plus petites. Ce faisant, nous augmentons la localité temporelle des données et diminuons le nombre d'échecs de la mémoire cache. Une condition nécessaire à l'application du pavage est la suivante : il faut que tous les éléments de tous les vecteurs dépendance soient positifs ou nuls. Un exemple est donné à la figure 5.4. Une boucle qui pourrait bénéficier d'une transformation avec la méthode du pavage est présentée en (a). Après l'application du pavage, la localité temporelle des données est augmentée (voir (b)). En (c), une représentation graphique des tuiles est donnée. Les chiffres représentent l'ordre d'exécution des tuiles. La présence d'un vecteur dépendance dont un des éléments est négatif (les vecteurs d_3 et d_4, par exemple) empêcherait l'encodage de la tuile 0.

```
for i=1 to M
  for j=1 to N
    x[i][j] = f1(i, j)
    y[i][j] = f2(x[i-1][j],
                 x[i][j-1])
  end loop
end loop
```

(a)

```
for i1=0 to 1
  for j1=0 to 1
    for i2=1 to M/2
      for j2=1 to N/2
        i = i2+(i1*M/2)
        j = j2+(j1*N/2)
        x[i][j] = f1(i, j)
        y[i][j] = f2(x[i-1][j],
                     x[i][j-1])
      end loop
    end loop
  end loop
end loop
```

(b)



(c)

Figure 5.4 : Exemple de la méthode du pavage. (a) Les vecteurs dépendance sont d_1 = (1, 0) et d_2 = (0, 1); (b) L'espace d'itération est divisé en quatre tuiles; (c) Une dépendance négative fausserait les résultats.

5.3.4 Allocation de tampons

Lorsque plusieurs boucles sont fusionnées en une seule, certains tableaux de données peuvent possiblement être remplacés par des tampons de tailles inférieures. En plus de réduire l'espace mémoire requis par l'application, cette technique augmente la localité spatiale des données. Un exemple est donné à la figure 5.5. Dans cet exemple, les tableaux intermédiaires x et y de tailles M+1 et M sont remplacés par les tampons B_1 et B_2 de tailles 2 et 3.

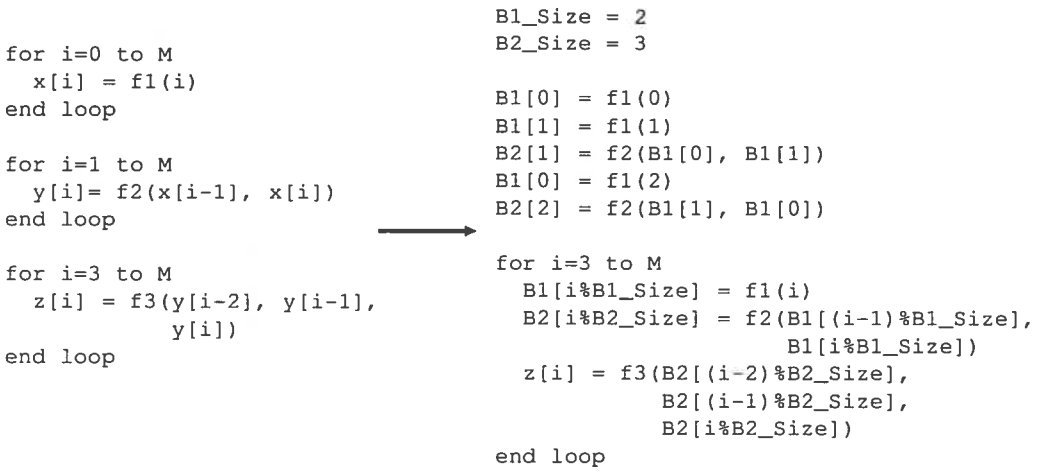


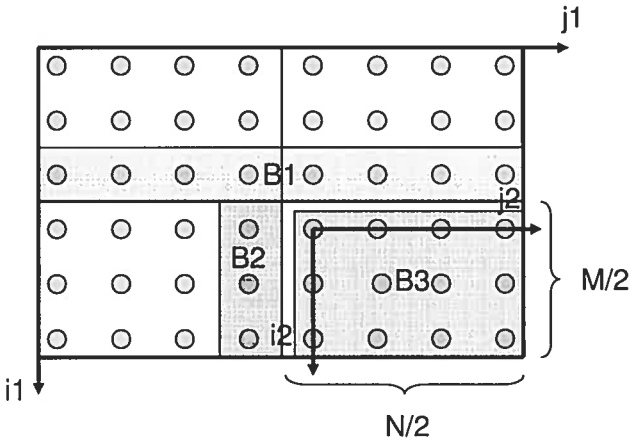
Figure 5.5 : Fusion et allocation de tampons.

Il est également possible d'utiliser la technique d'allocation de tampons avec la technique du pavage. Les calculs effectués sur une tuile peuvent dépendre des résultats des calculs effectués sur d'autres tuiles. Ces dernières doivent donc avoir été traitées antérieurement. Si le nid de boucles à optimiser est de profondeur n, il faudra possiblement n + 1 tampons, soit un pour chaque boucle (pour conserver les données sur les frontières des tuiles) et un pour la tuile courante, afin de satisfaire les dépendances. La figure 5.6 reprend l'exemple de la figure 5.4 et applique la technique de l'allocation de tampons. Le tableau intermédiaire x est remplacé par

trois tampons : B_1 conserve les données situées sur la frontière horizontale, B_2 conserve les données sur la frontière verticale et B_3 conserve les données intermédiaires à l'intérieur de la tuile courante.

```
for i1=0 to 1
  for j1=0 to 1
    for i2=1 to M/2
      for j2=1 to N/2

        i = i2+(i1*M/2)
        j = j2+(j1*N/2)

        B3[i2][j2] = f1(i, j)

        if (i2 == 1)
          arg1 = B1[j]
        else
          arg1 = B3[i2-1][j2]
        if (j2 == 1)
          arg2 = B2[i2]
        else
          arg2 = B3[i2][j2-1]

        y[i][j] = f2(arg1, arg2)

        if (i2 == M/2)
          B1[j] = B3[i2][j2]
        if (j2 == N/2)
          B2[i2] = B3[i2][j2]

      end loop
    end loop
  end loop
end loop
```

(a)



(b)

Figure 5.6 : Pavage et allocation de tampons. (a) Exemple de code; (b) Représentation graphique des tampons B_1, B_2 et B_3 utilisés dans le code de (a).

5.4 Distribution de l'application

Nous souhaitons étendre les concepts de base de la transformation de boucles à un contexte multiprocesseur. Il faut donc chercher à distribuer l'application de détection de contours tout en respectant les conditions nécessaires à l'utilisation des concepts de base. Puisque les seules dépendances qui existent sont des dépendances inter boucles, nous pouvons paralléliser chacune des boucles et ainsi utiliser plusieurs processeurs. Toutefois, cette solution ne met pas à profit les techniques de transformation de boucles. La solution choisie divise l'image à encoder en tuiles. Nous associons une tuile à chaque processeur, responsable de son traitement (voir figure 5.7). Ainsi, nous pouvons appliquer la fusion, le décalage de boucles, le pavage ainsi que l'allocation de tampons. Les sections suivantes énoncent les conditions nécessaires pour appliquer cette solution, décrivent le problème des frontières, détaillent le traitement des tuiles et enfin proposent des stratégies mettant à profit les processeurs *multithreaded*.

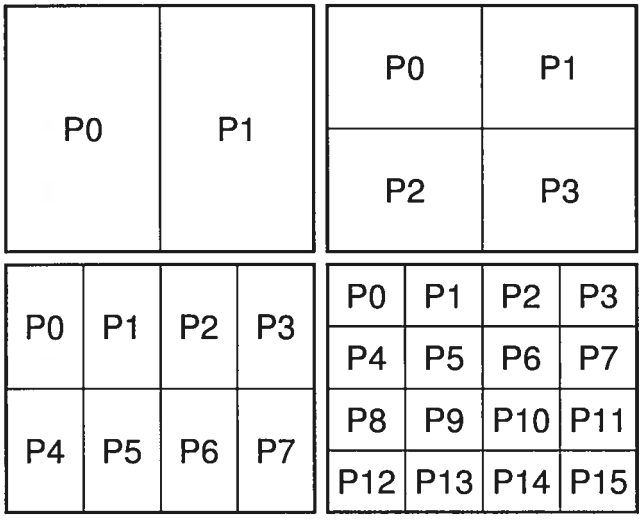


Figure 5.7 : Distribution de l'application sur 2, 4, 8 ou 16 processeurs.

5.4.1 Conditions

Pour déterminer si l'application que nous souhaitons modifier répond aux conditions nécessaires à l'utilisation des techniques de transformation de boucles, nous devons examiner plus en détails les vecteurs dépendance contenus dans la boucle d'encodage. Voici la boucle d'encodage de l'application avant transformation :

```
for i=2 to M-2
  for j=1 to N-1
    Temp1[i][j] = GaussBlurVert(IN[i-1][j], IN[i][j], IN[i+1][j])

for i=2 to M-2
  for j=2 to N-2
    Temp2[i][j] = GaussBlurHor(Temp1[i][j-1], Temp1[i][j],
                               Temp1[i][j+1])

for i=3 to M-3
  for j=3 to N-3
    Temp3[i][j] = ComputeEdges(Temp2[i+1][j+1], Temp2[i+1][j],
                               Temp2[i+1][j-1], Temp2[i][j+1],
                               Temp2[i][j],     Temp2[i][j-1],
                               Temp2[i-1][j+1], Temp2[i-1][j],
                               Temp2[i-1][j-1])

for i=3 to M-3
  for j=3 to N-3
    Temp4[i][j] = Reverse(Temp3[i][j])

for i=4 to M-4
  for j=4 to N-4
    OUT[i][j] = DetectRoots(Temp4[i+1][j+1], Temp4[i+1][j],
                            Temp4[i+1][j-1], Temp4[i][j+1],
                            Temp4[i][j],     Temp4[i][j-1],
                            Temp4[i-1][j+1], Temp4[i-1][j],
                            Temp4[i-1][j-1])
```

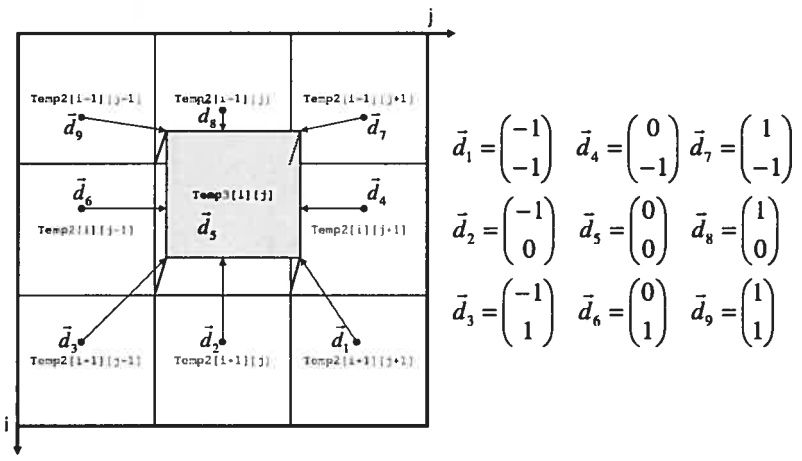


Figure 5.8 : Dépendances entre le calcul de `Temp2` et celui de `Temp3`.

La figure 5.8 montre les dépendances qui existent entre le calcul de `Temp2` et celui de `Temp3`. Nous pouvons constater que les vecteurs d_1 à d_4 sont lexicographiquement négatifs. Nous ne pouvons donc appliquer la fusion sans considérer également le décalage de boucles. Après la fusion et le décalage de boucles, le code de l'application devient :

```
for i=0 to M
  for j=0 to N
    Temp1[i+2][j+3] = GaussBlurVert(IN[i+1][j+3], IN[i+2][j+3],
                                    IN[i+3][j+3])

    Temp2[i+2][j+2] = GaussBlurHor(Temp1[i+2][j+1], Temp1[i+2][j+2],
                                   Temp1[i+2][j+3])

    Temp3[i+1][j+1] = ComputeEdges(Temp2[i+2][j+2], Temp2[i+2][j+1],
                                   Temp2[i+2][j],   Temp2[i+1][j+2],
                                   Temp2[i+1][j+1], Temp2[i+1][j],
                                   Temp2[i][j+2],   Temp2[i][j+1],
                                   Temp2[i][j])

    Temp4[i+1][j+1] = Reverse(Temp3[i+1][j+1])

    OUT[i][j] = DetectRoots(Temp4[i+1][j+1], Temp4[i+1][j],
                            Temp4[i+1][j-1], Temp4[i][j+1],
                            Temp4[i][j],     Temp4[i][j-1],
                            Temp4[i-1][j+1], Temp4[i-1][j],
                            Temp4[i-1][j-1])
  end loop
end loop
```

Pour simplifier, les gardes devant les calculs ont été enlevées. Les vecteurs dépendance sont maintenant tous lexicographiquement positifs ou nuls. En fait, il y a mieux : tous les éléments de tous les vecteurs dépendance sont positifs ou nuls. Nous pouvons donc appliquer le pavage et l'allocation de tampons, comme montré à la figure 5.6. Les vecteurs dépendance associés au calcul de `Temp3` sont donnés à la figure 5.9. Toutes ces transformations nous permettent presque de distribuer l'application comme le montre la figure 5.7. Il reste cependant un problème à résoudre : celui des frontières entre les tuiles.

$$\vec{d}_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \vec{d}_4 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \vec{d}_7 = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$
$$\vec{d}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \vec{d}_5 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \vec{d}_8 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$
$$\vec{d}_3 = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \quad \vec{d}_6 = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \vec{d}_9 = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

Figure 5.9 : Vecteurs dépendance extraits du code fusionné et décalé.

5.4.2 Problème des frontières

Les dépendances entre les calculs à effectuer empêchent le traitement simultané de certaines tuiles. En effet, les deux dépendances mises en évidence dans la figure 5.10, par exemple, empêchent le processeur P1 d'amorcer le traitement de la tuile associée jusqu'à ce que le processeur P0 ait terminé. Pour contourner ce problème, nous traitons d'abord les parties de l'image situées sur les frontières entre les tuiles, puis ensuite seulement nous lançons le traitement parallèle des tuiles. Appliquée à l'exemple de la figure 5.10, cette solution traite d'abord la frontière verticale entre les zones associées à P0 et P1, d'une largeur égale à deux pixels, avant d'amorcer le traitement simultané des deux tuiles.

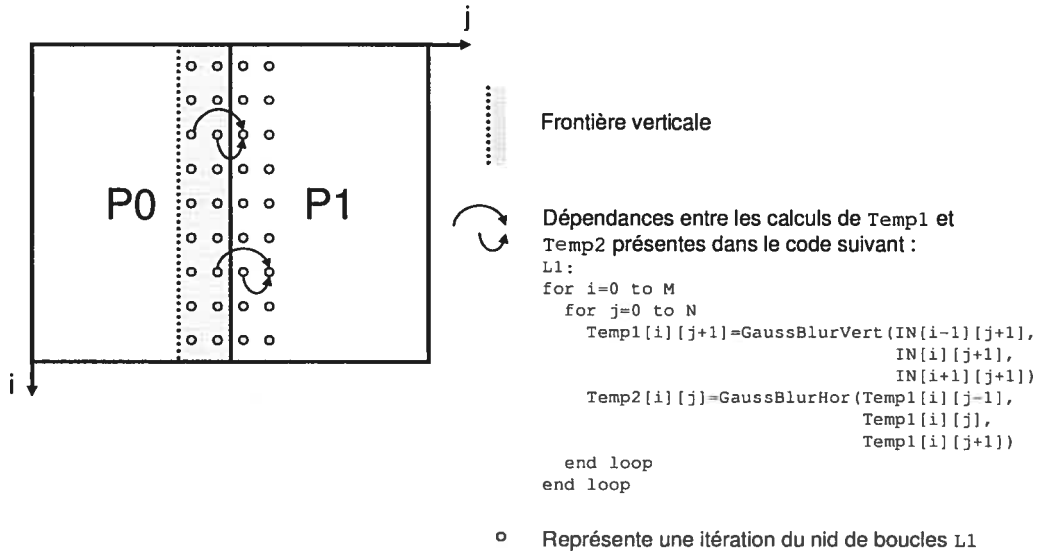


Figure 5.10 : Problème des frontières.

Afin de déterminer la largeur des frontières à traiter, nous devons étudier les dépendances. Pour l'application de détection de contours, nous avons déjà trouvé dans la section précédente les dépendances entre les calculs de `Temp2` et de `Temp3` (voir figure 5.9). Puisque les dépendances les plus grandes sont égales à 2 sur les axes *i* et *j*, nous pourrions penser que des frontières verticale et horizontale de largeur 2 suffisent pour le calcul de `Temp3`. Cependant, il est essentiel de tenir compte de toutes les dépendances présentes dans l'application. Si, par exemple, une dépendance égale à 1 existait entre les calculs de `Temp3` et de `Temp4`, il faudrait élargir de 1 les frontières pour le calcul de `Temp3`.

Ainsi, un algorithme simple pour trouver la largeur des frontières est le suivant :

```
Initialiser X à 0

Pour i = n jusqu'à 1 :
  Trouver les vecteurs dépendance entre les calculs C_{i-1} et C_i, que
  l'on nommera V_0 … V_m

  Pour j = 1 jusqu'à dimension des vecteurs V_x :
    maximum = 0

    Pour k = 0 jusqu'à m :
      Si la composante j du vecteur V_k > maximum
        maximum = V_{k_j}

    X_{i-1_j} = maximum + X_{i_j}

    Largeur de la Frontière_i selon l'axe j = X_{i_j}

Pour j = 1 jusqu'à dimension des vecteurs V_x :
  Largeur de la Frontière_0 selon l'axe j = X_{0_j}
```

Cet algorithme suppose que toutes les dépendances se retrouvent entre deux nids de boucles adjacents, que tous les éléments des vecteurs dépendance sont positifs ou nuls et que tous les nids de boucles à fusionner ont la même profondeur. Appliqué à l'application qui nous intéresse, l'algorithme retourne les valeurs présentées dans le tableau IV pour les largeurs des frontières.

Tableau IV : Largeur des frontières, en pixels.

Calcul / Frontière	Temp1	Temp2	Temp3	Temp4	Out
Horizontale	4	4	2	2	0
Verticale	6	4	2	2	0

Ainsi, si nous distribuons l'application sur quatre processeurs, nous devons traiter une frontière horizontale et une frontière verticale avant de pouvoir lancer l'exécution simultanée des quatre tuiles à traiter. Puisque le traitement des frontières se fait de

façon indépendante, il est possible d'associer un processeur à chacune des frontières et de procéder à leur traitement en parallèle. Une représentation graphique des frontières est donnée à la figure 5.11.

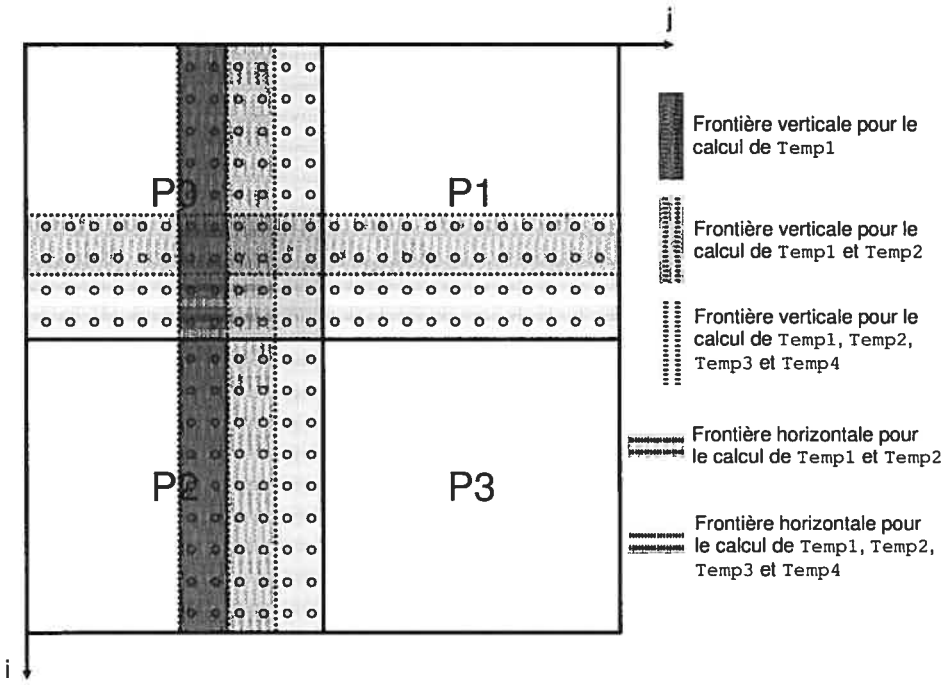


Figure 5.11 : Frontières à traiter pour l'application de détection de contours lorsque configurée pour quatre processeurs.

5.4.3 Encodage des tuiles

Lorsque le traitement des frontières est terminé, le traitement des tuiles peut commencer. Nous avons considéré quatre versions : avec fusion, avec fusion et allocation de tampons, avec fusion, allocation de tampons et pavage et avec utilisation de processeurs *multithreaded* (voir section suivante).

Les séquences de code décrivant la fonction générale *Encode_Image*, la fonction *Encode_Frontieres_Verticales* spécifique à la première version étudiée ainsi que les

fonctions *Encode_Tuiles* spécifiques à la première et à la deuxième version sont présentées dans l'annexe B.

5.4.4 *Multithreading*

Il est possible de distribuer l'application en utilisant des processeurs à contextes d'exécution multiples. Nous proposons deux stratégies : réutiliser en totalité les concepts proposés dans les sections précédentes ou paralléliser la boucle d'encodage interne, tel que détaillé dans la section 4.3.1.

La première stratégie est très simple : au lieu d'utiliser, par exemple, 16 processeurs pour traiter une trame, nous choisissons d'utiliser 4 processeurs qui peuvent gérer 4 contextes d'exécution chacun. Ainsi, chaque processeur est responsable du traitement de 4 tuiles (voir figure 5.12). Cette stratégie ne requiert aucune modification dans le code de l'application.

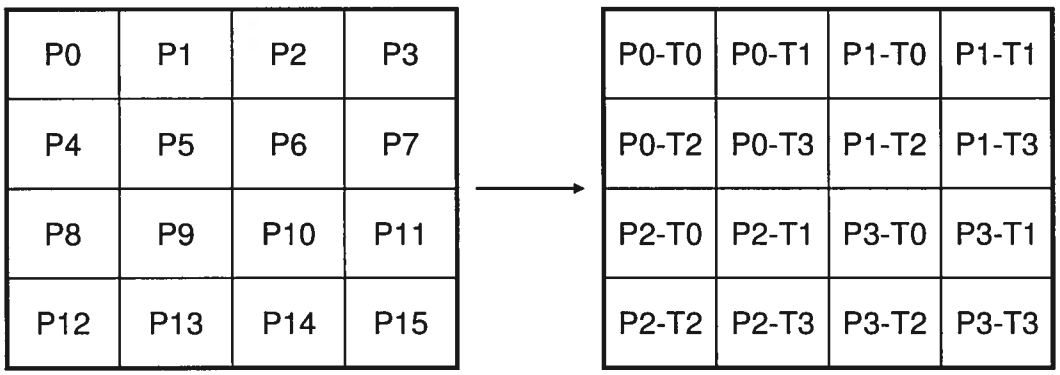


Figure 5.12 : Distribution de l'application de détection de contours sur 16 processeurs, puis sur 4 processeurs à contextes d'exécution multiples (P : processeur; T : *thread*).

La deuxième stratégie est similaire à celle utilisée pour optimiser l'application d'encodage vidéo dans le chapitre 4. Il s'agit d'appliquer une transformation unimodulaire aux bornes de la boucle de traitement fusionnée.

Dans le cas de l'application de détection de contours, la matrice de transformation T est donnée par :

$$T = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

La boucle de traitement devient :

```
for i' = 0 to (M-1) + (N-1)
  forall j' = f1(i') to f2(i')
    i = j'
    j = i' - j'

    idx = (i+2)*N+j+3
    Temp1[idx] = GaussBlurVert(IN[idx-N], IN[idx], IN[idx+N])

    idx = idx-1
    Temp2[idx] = GaussBlurHoriz(Temp1[idx-1], Temp1[idx],
                                Temp1[idx+1])

    idx = idx-1-N
    Temp3[idx] = ComputeEdges(Temp2[idx+N+1], Temp2[idx+N],
                              Temp2[idx+N-1], Temp2[idx+1],
                              Temp2[idx],     Temp2[idx-N+1],
                              Temp2[idx-N],   Temp2[idx-N-1],
                              Temp2[idx-1])

    Temp4[idx] = Reverse(Temp3[idx])

    idx = idx-1-N
    OUT[idx] = DetectRoots(Temp4[idx+N+1], Temp4[idx+N],
                           Temp4[idx+N-1], Temp4[idx+1],
                           Temp4[idx],     Temp4[idx-N+1],
                           Temp4[idx-N],   Temp4[idx-N-1],
                           Temp4[idx-1])
  end loop
end loop
```

Les gardes devant les cinq étapes de traitement ont été enlevées pour alléger l'écriture. Les fonctions f_1 et f_2 sont données par :

$$f_1(i') = \max(0, i'-N+1)$$
$$f_2(i') = \min(M-1, i')$$

Cette stratégie peut être utilisée pour un nombre quelconque de processeurs. Elle ne divise pas l'image à traiter en tuiles et ne nécessite donc aucun traitement des frontières. Par contre, elle profite moins de la localité des données. En effet, chaque fil d'exécution est responsable du traitement d'un seul pixel. La stratégie de l'ordonnanceur (premier arrivé, premier servi) fait en sorte que des pixels adjacents ne sont en général pas traités par un même processeur. Des résultats comparant les deux stratégies proposées sont donnés dans la section suivante.

5.5 Résultats

Toutes les méthodologies proposées dans les sections précédentes ont été implémentées en C++ et testées avec l'environnement de simulation StepNP. Les prochaines sous-sections présentent les résultats de ces tests pour des environnements monoprocesseur et multiprocesseur. La figure 5.13 donne une vue schématique de la plateforme simulée utilisée pour obtenir ces résultats. La configuration de la mémoire cache utilisée lors de toutes les simulations est la suivante :

Associativité : 4 voies
Taille d'un bloc : 4 octets

Taille de l'image source utilisée : 640×400 pixels

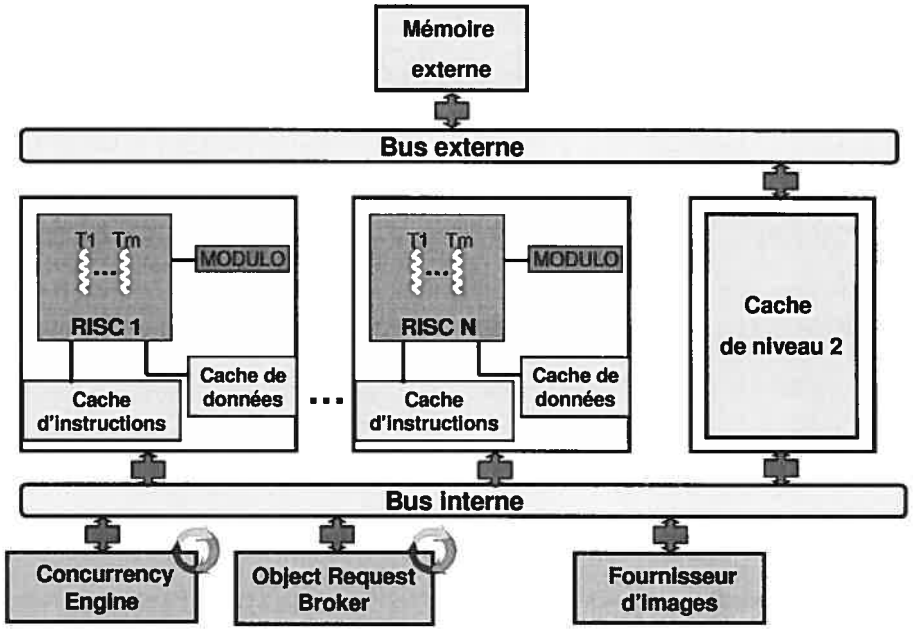


Figure 5.13 : Plateforme simulée utilisée pour tester l'application de détection de contours.

5.5.1 Monoprocesseur

Les transformations de boucles présentées en 5.3 visent toutes à réduire le taux d'échecs de la mémoire cache. Ces transformations ont été introduites dans la version initiale de l'application de détection de contours puis simulées pour obtenir les taux d'échecs de la mémoire cache (voir figure 5.14) ainsi que le nombre d'instructions exécutées (voir figure 5.15).

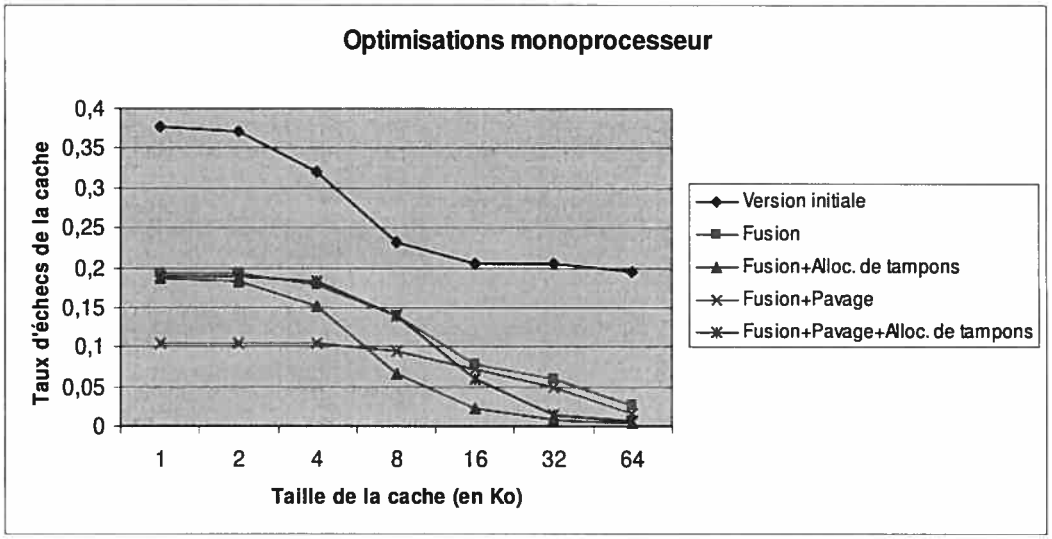


Figure 5.14 : Taux d'échecs de la mémoire cache obtenus pour différentes versions de l'application.

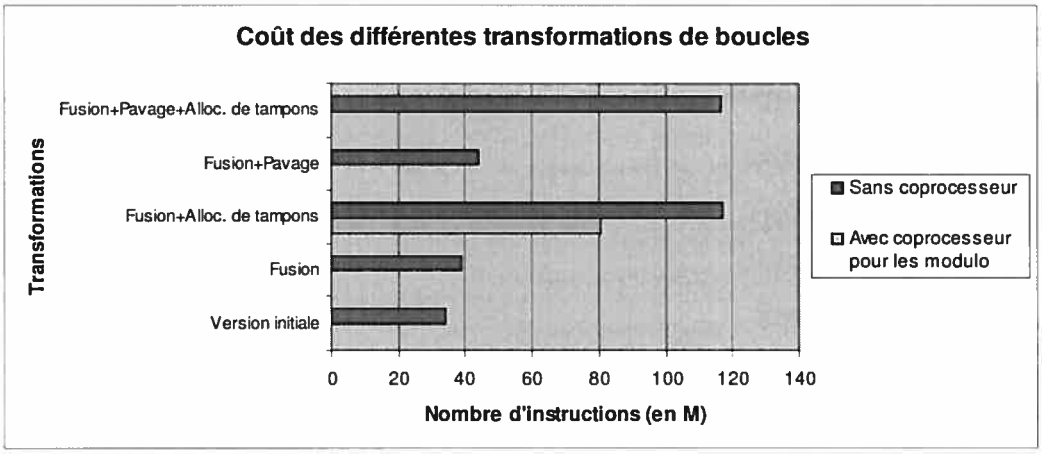


Figure 5.15 : Nombre d'instructions exécutées pour effectuer la détection de contours sur une image de 640×400 pixels.

5.5.2 Multiprocesseur

Les optimisations présentées en 5.4 visant à distribuer efficacement l'application de détection de contours ont été implémentées et simulées avec l'environnement StepNP. Elles sont comparées avec la version initiale qui ne profite pas de la localité des données (voir figure 5.16). Les figures 5.17 et 5.18 montrent les résultats des simulations respectivement pour le taux d'échecs des mémoires cache et pour le temps d'exécution. La plateforme simulée comprend quatre processeurs à contexte d'exécution unique. Le nombre élevé d'opérations modulo empêche les versions avec allocation de tampons de surclasser la version initiale au niveau du temps d'exécution. Des solutions existent pour éliminer ces opérations et tirer ainsi pleinement avantage du taux d'échecs des mémoires cache réduit causé par l'allocation de tampons. Bouchebaba et al. [43] proposent une telle technique et parviennent ainsi à réduire de moitié le temps d'exécution de la version avec fusion, pavage et allocation de tampons tout en conservant un taux d'échecs des mémoires cache très bas.

Les paramètres des bus interne et externe sont configurés ainsi :

Transaction sur le bus interne : 30 ns
Transaction sur le bus externe : 60 ns
Gigue : 0 ns

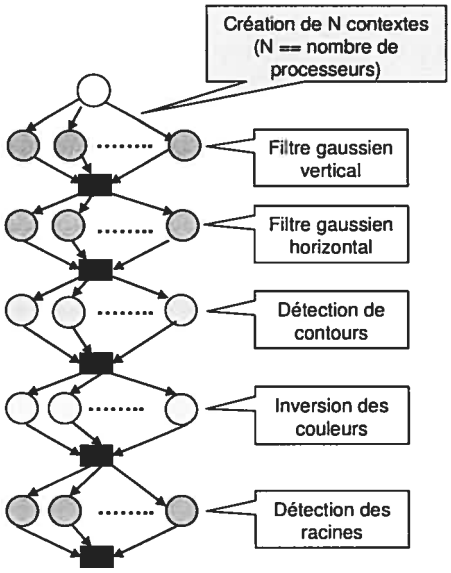


Figure 5.16 : Version initiale multiprocesseur de l'application de détection de contours.

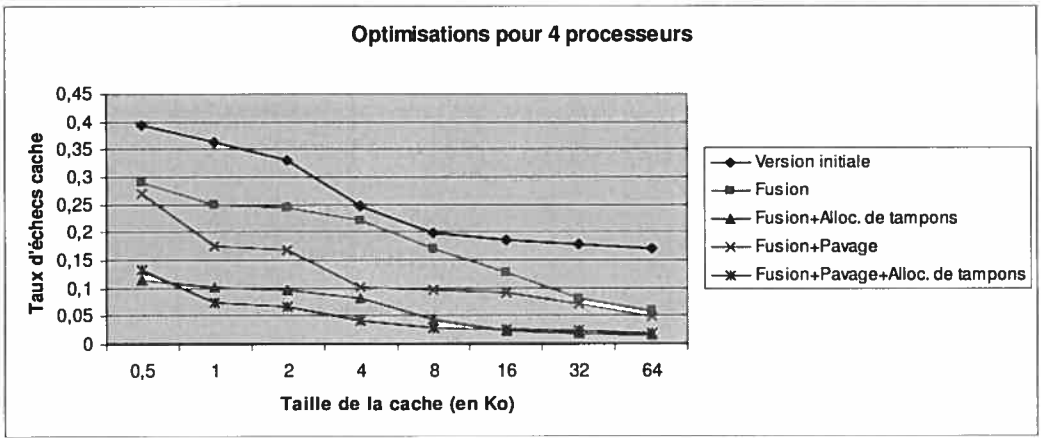


Figure 5.17 : Taux d'échecs des mémoires cache d'un système multiprocesseur exécutant différentes versions de l'application de détection de contours.

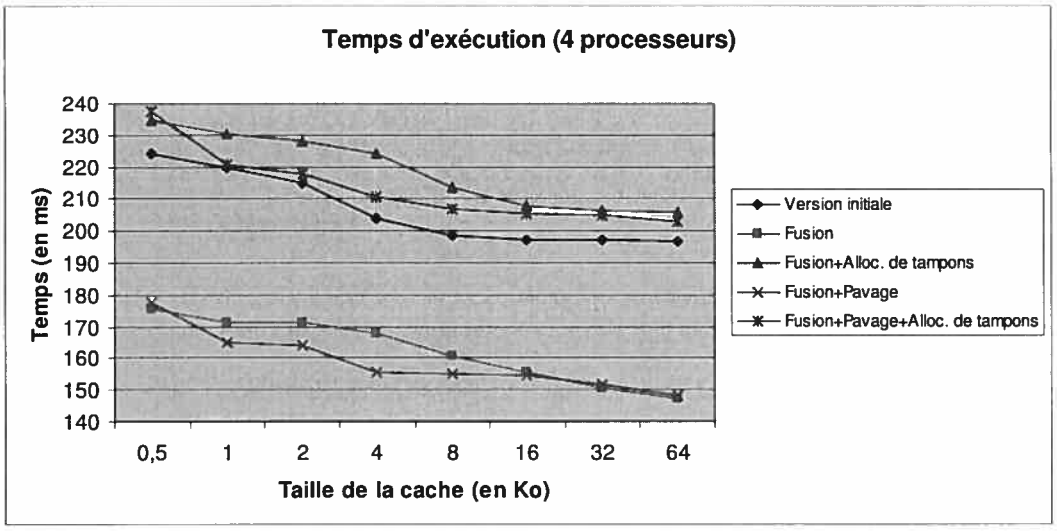


Figure 5.18 : Temps d'exécution de différentes versions de l'algorithme de détection de contours pour une image de 640×400 pixels.

Il est également intéressant d'étudier l'impact du nombre de processeurs utilisés sur le taux d'échecs des mémoires cache et le temps d'exécution de l'algorithme. Les figures 5.19, 5.20 et 5.21 montrent que la méthodologie proposée est extensible : le taux d'échecs des mémoires cache reste stable et le temps d'exécution décroît approximativement linéairement lorsqu'on passe de quatre à huit processeurs, puis de huit à seize.

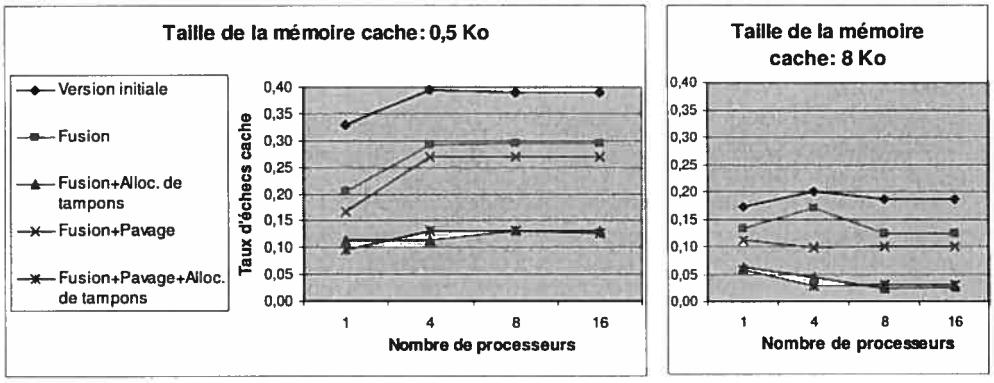


Figure 5.19 : Variations du taux d'échecs des mémoires cache selon le nombre de processeurs simulés.

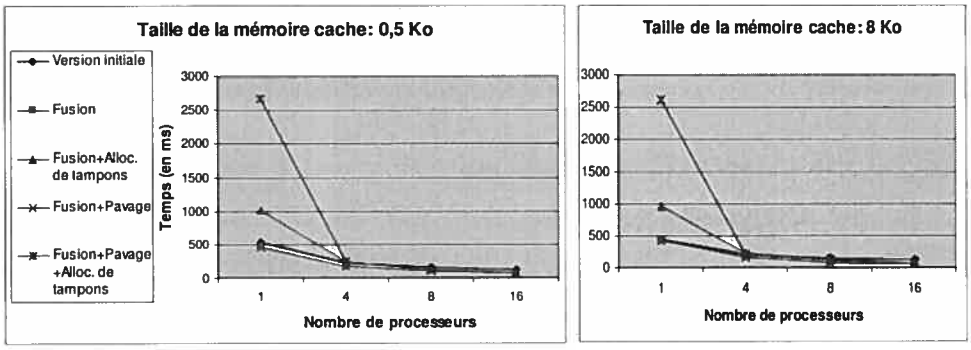


Figure 5.20 : Variations du temps d'exécution de l'application selon le nombre de processeurs simulés.

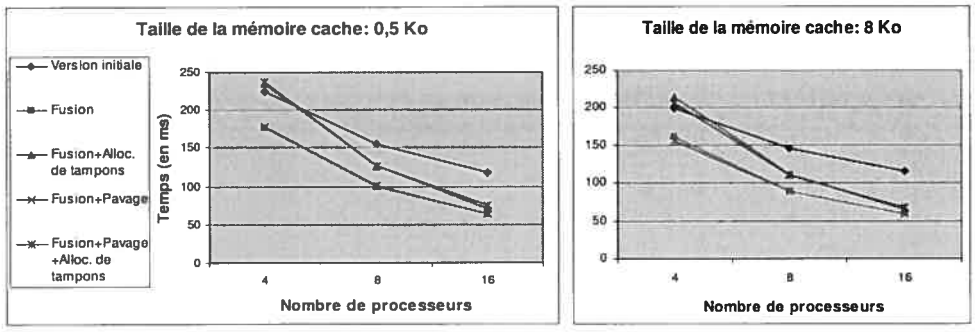


Figure 5.21 : Version "agrandie" de la figure 5.20 (utilisée pour montrer l'efficacité de la méthodologie proposée pour des systèmes multiprocesseurs).

En faisant varier le nombre de contextes d'exécution des processeurs, nous pouvons évaluer l'influence du *multithreading*, présenté dans la section 5.4.4. En plus des versions déjà étudiées, nous avons testé la version initiale avec transformation unimodulaire, présentée en 5.4.4. Nous avons mesuré le taux d'échecs des mémoires cache et le temps d'exécution sur quatre plateformes à huit contextes d'exécution, répartis comme ceci :

- 1 processeur à 8 contextes.
- 2 processeurs à 4 contextes.
- 4 processeurs à 2 contextes.
- 8 processeurs à 1 contexte.

Les résultats sont donnés dans les figures 5.22 et 5.23. Il est intéressant de noter que les temps d'exécution calculés pour les plateformes à 4 processeurs (32 Ko de mémoire cache de niveau 1 au total) sont en moyenne 12,7% plus lents que ceux calculés pour les plateformes à 8 processeurs (64 Ko de mémoire cache de niveau 1 au total). Dans un tel contexte, l'utilisation de processeurs à contextes d'exécution multiples doit être sérieusement envisagée.

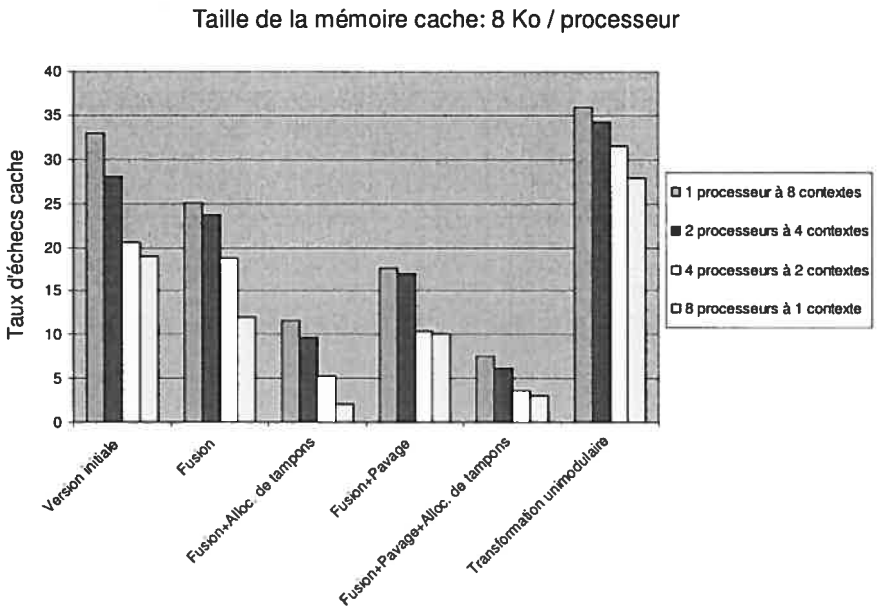


Figure 5.22 : Variations du taux d'échecs des mémoires cache pour quatre configurations possibles d'une plateforme à huit contextes d'exécution.

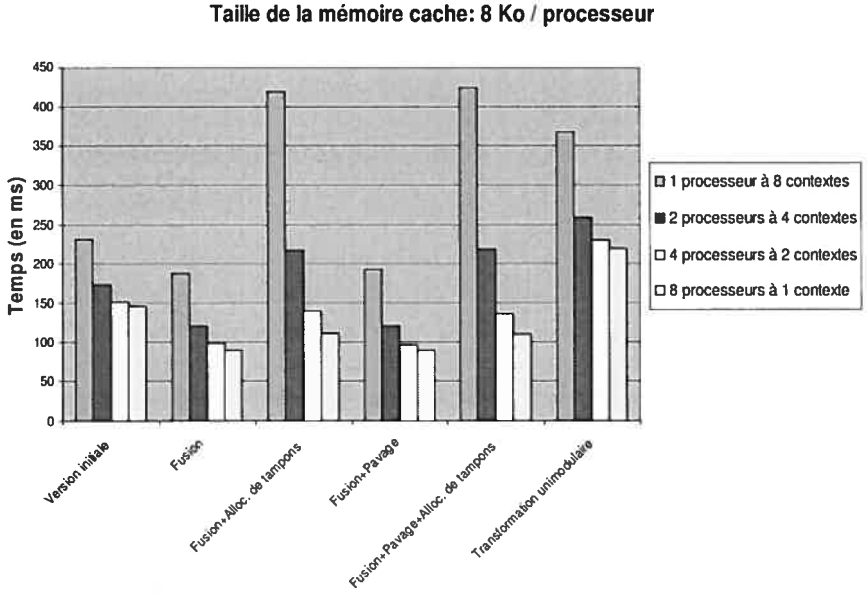


Figure 5.23 : Variations du temps d'exécution de l'application pour quatre configurations possibles d'une plateforme à huit contextes d'exécution.

Puisque l'application étudiée est simple et demande peu de puissance de calcul, le coût en termes de nombre d'instructions supplémentaires et de temps d'exécution pour implémenter les méthodes proposées peut paraître assez important. Par contre, le coût associé à la transformation d'une application plus complexe serait proportionnellement plus bas, ce qui rendrait l'utilisation des méthodes proposées plus intéressantes.

Chapitre 6 : Travaux futurs

Plusieurs idées brièvement discutées dans ce document pourraient devenir d'intéressantes voies à suivre pour améliorer les méthodologies proposées. Par exemple, l'ajout d'une mémoire cache associée au coprocesseur et d'une mémoire bloc-notes responsable de la prélecture des données, deux concepts présentés dans les sections 4.6.4 et 4.6.5 respectivement, ne sont pas totalement exploités. Il serait intéressant d'étudier plus en détails les problèmes liés à la cohérence des mémoires cache lors de l'exécution de l'encodeur MPEG-4 afin de créer un modèle de mémoire cache plus adapté au coprocesseur. Il faudrait aussi résoudre les problèmes liés à l'augmentation de la taille des blocs des mémoires cache. Des blocs de taille plus grande réduiraient les taux d'échecs des mémoires cache. Il serait également intéressant de développer le concept de la prélecture afin de l'appliquer à la trame précédente en plus de la trame courante.

Afin de valider les résultats obtenus pour l'application de détection de contours, il faudrait appliquer la méthodologie proposée sur d'autres applications multimédias et effectuer une série de tests semblables à ceux présentés dans la section 5.5. Plusieurs facteurs peuvent potentiellement influencer les résultats obtenus. Par exemple, la configuration de la mémoire cache, la latence des bus, la taille de l'image et la simplicité de l'algorithme étudié font en sorte qu'il serait précipité de tirer des conclusions sur les performances de la méthodologie proposée pour des systèmes multiprocesseurs.

Chapitre 7 : Conclusion

Ce travail aura permis de mieux comprendre comment les choix des développeurs d'applications destinées aux réseaux sur puce peuvent influencer la performance du système final, tant au niveau de la rapidité d'exécution que de la puissance dissipée. Un développeur ne peut faire abstraction de l'architecture du système lorsqu'il construit son application. De même, il doit considérer l'application à exécuter lorsqu'il conçoit la plateforme cible. Les optimisations de l'application et de l'architecture doivent donc aller de pair. Ce constat se dégage principalement du chapitre 4, dans lequel nous avons présenté les résultats de l'optimisation d'un algorithme d'encodage MPEG-4 distribué et simulé sur la plateforme de simulation StepNP. Nous avons modifié une version hautement parallèle de l'encodeur en introduisant un pipeline fonctionnel. Ceci, en plus d'autres optimisations, nous a permis de réduire la taille de la mémoire requise par l'encodeur. Elle est passée d'environ 40 Mo à 1 Mo.

Nous avons également présenté une méthodologie qui permet d'utiliser les concepts de base des transformations de boucles comme la fusion et le pavage dans un contexte multiprocesseur. Nous avons testé cette méthodologie sur une application de détection de contours. Nous avons implémenté et comparé plusieurs versions de cette application : des versions monoprocesseurs et multiprocesseurs avec et sans transformations de boucles. Nous avons également fait varier le nombre de contextes d'exécution des processeurs. Dans un contexte multiprocesseur (quatre processeurs simulés), lorsque toutes les optimisations sont utilisées (fusion, pavage et allocation de tampons), le taux d'échecs des mémoires cache est réduit en moyenne de 20% par rapport à la version originale. Les résultats démontrent également que les taux d'échecs des mémoires cache n'augmentent en moyenne que de 5,5% lorsqu'on passe des versions monoprocesseurs à multiprocesseurs (quatre processeurs simulés, taille des mémoires cache : 0,5 Ko). Les mêmes tests, effectués avec une mémoire cache de 8 Ko, donnent des résultats encore plus intéressants : les taux d'échecs des mémoires cache diminuent alors de 0,5% en moyenne pour les versions

multiprocesseurs. Finalement, la méthodologie proposée est extensible : les taux d'échecs des mémoires cache n'augmentent pas lorsque le nombre de processeurs utilisés augmente.

Bibliographie

[1] P. Paulin, C. Pilkington and E. Bensoudane, "StepNP: A System-Level Exploration Platform for Network Processors," *IEEE Des. Test,* 2002, pp. 17-26.

[2] P. Paulin et al., "A Multi-Processor SoC Platform and Tools for Communications Applications," in *Embedded Systems Handbook,* CRC Press, 2004.

[3] P. Paulin et al., "Parallel Programming Models for a Multi-Processor SoC Platform Applied to Networking and Multimedia," to appear in *IEEE Transactions on Very Large Scale Integration Systems,* 2006.

[4] L. Cai and D. Gajski, "Transaction Level Modeling: An Overview," *CODES+ISSS,* 2003, pp. 19-24.

[5] V. Reyes et al., "CASSE: A System-Level Modeling and Design-Space Exploration Tool for Multiprocessor Systems-on-Chip," *DSD,* 2004, pp. 476-483.

[6] J. Chevalier et al., "SPACE: A Hardware/Software SystemC modeling platform including an RTOS," in *Languages for System Specification and Verification,* Kluwer Academic Publishers, 2004.

[7] G. Kahn, "The semantics of a simple language for parallel programming," *Information processing,* 1974, pp. 471-475.

[8] Site web de SystemC: http://www.systemc.org

[9] A.D. Pimentel, C. Erbas and S. Polstra, "A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels," *IEEE Transactions on Computers,* vol. 55, no. 2, 2006, pp. 99-112.

[10] V.D. Zivkovic et al., "Fast and Accurate Multiprocessor Architecture Exploration with Symbolic Programs," *DATE,* 2003.

[11] A.S. Cassidy, J.M. Paul and D.E. Thomas, "Layered, Multi-Threaded, High-Level Performance Design," *DATE,* 2003.

[12] S. Mohanty and V.K. Prasanna, "Rapid System-Level Performance Evaluation and Optimization for Application Mapping onto SOC Architectures," *ASIC/SOC,* 2002.

[13] F. Balarin et al., "Metropolis: An Integrated Electronic System Design Environment," *Computer,* vol. 36, no. 4, 2003, pp. 45-52.

[14] A. Mihal et al., "Developing Architectural Platforms: A Disciplined Approach," *IEEE Design & Test of Computers* , vol. 19, no. 6, 2002, pp. 6-16.

[15] K.D. Nguyen et al., "Model-Driven SoC Design via Executable UML to SystemC," *RTSS,* 2004, pp. 459-468.

[16] A. Tsikhanovich, E.M. Aboulhamid and G. Bois, "A Methodology for Hw/Sw Specification and Simulation at Multiple Levels of Abstraction," *IWSOC,* 2005, pp. 24-29.

[17] A. Wieferink et al., "A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platforms," *DATE,* 2004.

[18] O. Blaurock, "A SystemC-Based Modular Design and Verification Framework for C-Model Reuse in a HW/SW-Codesign Flow," *ICDCSW,*

2004, pp. 838-843.
[19] The MPEG Home Page: http://www.chiariglione.org/mpeg/
[20] K. Denolf et al., "Memory Centric Design of an MPEG-4 Video Encoder," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 15, no. 5, 2005, pp. 609-619.
[21] F. Catthoor et al., *Custom Memory Management Methodology, Exploration of memory organization for embedded multimedia system design,* Kluwer Academic Publ., 1998.
[22] L. Nachtergaele et al., "System-Level Power Optimization of Video Codecs on Embedded Cores: A Systematic Approach," *Journal of VLSI Signal Processing Systems,* vol. 18, no. 2, 1998, pp. 89-109.
[23] S.-H. Wang et al., "A Platform Based MPEG-4 Advanced Video Coding (AVC) Decoder with Block Level Pipelining," *ICICS-PCM,* 2003, pp. 51-55.
[24] H.-J. Stolberg et al., "The M-PIRE MPEG-4 Codec DSP and its Macroblock Engine," *ISCAS,* 2000, pp. II 192–II 195.
[25] E. Salminen, A. Kulmala and T.D. Hämäläinen, "HIBI-Based Multiprocessor SoC on FPGA," *ISCAS,* 2005, pp. 3351-3354.
[26] E. Salminen et al., "HIBI v.2 Communication Network for System-on-Chip," *SAMOS,* 2004, pp. 412-422.
[27] O. Lehtoranta et al., "A Parallel MPEG-4 Encoder for FPGA Based Multiprocessor SoC," *FPL,* 2005, pp. 380-385.
[28] A. Darte, "On the Complexity of Loop Fusion," *PACT,* 1999, pp. 149.
[29] A. Fraboulet, K. Kodary and A. Mignotte, "Loop fusion for memory space optimization," *ISSS,* 2001, pp. 95-100.
[30] K. Kennedy, "Fast Greedy Weighted Fusion," *International Journal of Parallel Programming,* vol. 29, no. 5, 2001, pp. 463-491.
[31] Y. Bouchebaba and F. Coelho, "Tiling and Memory Reuse for Sequences of Nested Loops," *Euro-Par,* 2002, pp. 255-264.
[32] P. Marchal, J.I. Gómez and F. Catthoor, "Optimizing the memory bandwidth with loop fusion," *CODES+ISSS,* 2004, pp. 188-193.
[33] S. Carr and K. Kennedy, "Scalar replacement in the presence of conditional control flow," *Software—Practice & Experience,* vol. 24, no. 1, 1994, pp. 51-77.
[34] M. Budiu and S.C. Goldstein, "Inter-Iteration Scalar Replacement in the Presence of Conditional Control-Flow," *ODES,* 2005, pp. 20-29.
[35] E. De Greef, "Storage Size Reduction for Multimedia Application," Phd thesis, IMEC, 1998.
[36] M. Kandemir, A.N. Choudhary and J. Ramanujam, "I/O-Conscious Tiling for Disk-Resident Data Sets," *Euro-Par,* 1999, pp. 430-439.
[37] N. Ahmed, N. Mateev and K. Pingali, "Tiling imperfectly-nested loop nests," *Supercomputing,* 2000.
[38] M. Kandemir, "A Compiler-Based Approach for Improving Intra-Iteration Data Reuse," *DATE,* 2002, pp. 984-990.
[39] M. Kandemir et al., "Optimizing inter-nest data locality," *CASES,* 2002, pp. 127-135.
[40] I. Kadayif and M. Kandemir, "Data space-oriented tiling for enhancing locality," *ACM Transactions on Embedded Computing Systems,* vol. 4, no. 2,

2005, pp. 388-414.

[41] F. Li and M. Kandemir, "Locality-conscious workload assignment for array-based computations in MPSOC architectures," *DAC,* 2005, pp. 95-100.

[42] G. Chen and M. Kandemir, "Code Restructuring for Improving Cache Performance of MPSoCs," *ICCAD,* 2005, pp. 271-274.

[43] Y. Bouchebaba et al., "MPSoC Memory Optimization Using Loop Transformations," submitted to *DAC,* 2006.

[44] M. Kandemir et al., "Dynamic management of scratch-pad memory space," *DAC,* 2001, pp. 690-695.

[45] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," *CASES,* 2003, pp. 276-286.

[46] J. Stokes, "Introduction to Multithreading, Superthreading and Hyperthreading," 2002, http://arstechnica.com/articles/paedia/cpu/hyperthreading.ars/1

[47] OCP International Partnership: http://www.ocpip.org/

[48] J.L. Hennessy and D.A. Patterson, *Computer Architecture: a Quantitative Approach,* Morgan Kaufmann Publishers Inc, 2002.

[49] M. Chaudhuri and M. Heinrich, "SMTp: An Architecture for Next-generation Scalable Multi-threading," *ISCA,* 2004, pp. 124-135.

[50] Fraunhofer Institut Integrierte Schaltungen: http://www.iis.fraunhofer.de/

Annexe A : Déclarations des structures de données

```
typedef struct {
  int mtype;
  int mquant;
  int inter;
  int nbit;
  int mvbit;
  int cbp;
  int dir;
  int *sum;
  int *ie;
  int *sad;
  MV **amv;  // Motion Vector
  MV **pmv;
  MV **dmv;
  int *info;

  // Blocs de 8×8 pixels
  SHORT_BLOCK **pred;
  INT_BLOCK **diff;
  SHORT_BLOCK **coef;
  SHORT_BLOCK **qres;
  SHORT_BLOCK **zres;
  SHORT_BLOCK **qinv;
  INT_BLOCK **dinv;

  RL **rl;   // Run/Level

} MACROBLOCK;
```

```
typedef struct {
  int mtype;
  int mquant;
  MV **amv;

} MB_PRV; // voir section 4.4.1

typedef struct {
  MV **amv;
  SHORT_BLOCK **qinv;

} MB_LAST; // voir section
           // 4.3.3.3
```

Annexe B : Fonctions d'encodage distribué de l'application de détection de contours

```
Fonction Encode_Image() {                          // Fonction générale
  // Traitement parallèle sur 4 processeurs

  // Traitement de la frontière horizontale
  Fork(Encode_Frontieres_Horizontales(), 1)
  // Traitement de la frontière verticale
  Fork(Encode_Frontieres_Verticales(), 1)
  Join()
  // Traitement des 4 tuiles
  Fork(Encode_Tuiles(), 4)
  Join()
}

Fonction Encode_Frontieres_Verticales(k) { // Spécifique à la
                                           // version fusionnée

  // k: représente le numéro de la frontière verticale
  // Pour 4 processeurs, il n'y a qu'une seule frontière verticale
  // et donc k = 0

  borne_j = N/2 + k * N/4

  for i=0 to M - 1
    for j=borne_j-6 to borne_j-1
     idx = (i+2)*N+j+3
     Temp1[idx] = GaussBlurVert(IN[idx-N], IN[idx], IN[idx+N])

  for i=0 to M - 1
    for j=borne_j-4 to borne_j-1
      idx = (i+2)*N+j+2
      Temp2[idx] = GaussBlurHoriz(Temp1[idx-1], Temp1[idx],
                                  Temp1[idx+1])

  for i=0 to M - 1
    for j=borne_j-2 to borne_j-1
      idx = (i+1)*N+j+1
      Temp3[idx] = ComputeEdges(Temp2[idx+N+1], Temp2[idx+N],
                                Temp2[idx+N-1], Temp2[idx+1],
                                Temp2[idx], Temp2[idx-N+1],
                                Temp2[idx-N], Temp2[idx-N-1],
                                Temp2[idx-1])
      Temp4[idx] = Reverse(Temp3[idx])
}
```

```
Fonction Encode_Tuiles(k) {      // Spécifique à la version fusionnée
  // k: représente le numéro de la tuile (0, 1, 2 ou 3)

  // M: hauteur de la trame
  // N: largeur de la trame

  borne_inf_i = (k % 2) * M/2
  borne_sup_i = borne_inf_i + M/2
  borne_inf_j = (k / 2) * N/2
  borne_sup_j = borne_inf_j + N/2

  for i=borne_inf_i to borne_sup_i - 1
    for j=borne_inf_j to borne_sup_j - 1

      idx = (i+2)*N+j+3
      Temp1[idx] = GaussBlurVert(IN[idx-N], IN[idx], IN[idx+N])
      idx = idx-1
      Temp2[idx] = GaussBlurHoriz(Temp1[idx-1], Temp1[idx],
                                  Temp1[idx+1])
      idx = idx-1-N
      Temp3[idx] = ComputeEdges(Temp2[idx+N+1], Temp2[idx+N],
                                Temp2[idx+N-1], Temp2[idx+1],
                                Temp2[idx], Temp2[idx-N+1],
                                Temp2[idx-N], Temp2[idx-N-1],
                                Temp2[idx-1])
      Temp4[idx] = Reverse(Temp3[idx])
      idx = idx-1-N
      OUT[idx] = DetectRoots(Temp4[idx+N+1], Temp4[idx+N],
                             Temp4[idx+N-1], Temp4[idx+1],
                             Temp4[idx], Temp4[idx-N+1],
                             Temp4[idx-N], Temp4[idx-N-1],
                             Temp4[idx-1])
}

Fonction Encode_Tuiles(k) {      // Spécifique à la version fusionnée
                                 // avec allocation de tampons
  // k: représente le numéro de la tuile (0, 1, 2 ou 3)

  // M: hauteur de la trame
  // N: largeur de la trame

  borne_inf_i = (k % 2) * (M/2)
  borne_sup_i = borne_inf_i + (M/2)
  borne_inf_j = (k / 2) * (N/2)
  borne_sup_j = borne_inf_j + (N/2)
  offset = (k % 2) * M*6 / 2
  offset2 = (k % 2) * M*4 / 2
  offset3 = (k % 2) * M*2 / 2
  offset4 = 0
  offset5 = 0

  for i=borne_inf_i to borne_sup_i - 1
    for j=borne_inf_j to borne_sup_j - 1

      idx = (i+2)*N+j+3
      B1[idx%SIZEOF_B1] = GaussBlurVert(IN[idx-N], IN[idx],
                                        IN[idx+N])
```

```
idx = idx-1
arg2 = B1[(idx+1)%SIZEOF_B1]

// Si j == (borne_inf_j ou borne_inf_j+1), on doit
// utiliser l'information calculée lors du traitement
// des frontières verticales
if (j == borne_inf_j)
  arg0 = B1_vert[offset+4]
  arg1 = B1_vert[offset+5]
else if (j == borne_inf_j + 1)
  arg0 = B1_vert[offset+5]
  arg1 = B1[idx%SIZEOF_B1]
  offset += 6
else
  arg0 = B1[(idx-1)%SIZEOF_B1]
  arg1 = B1[idx%SIZEOF_B1]

B2[idx%SIZEOF_B2] = GaussBlurHoriz(arg0, arg1, arg2)

idx = idx-1-N
arg0 = B2[(idx+N+1)%SIZEOF_B2]
arg1 = B2[(idx+N)%SIZEOF_B2]
arg2 = B2[(idx+N-1)%SIZEOF_B2]
arg3 = B2[(idx+1)%SIZEOF_B2]
arg4 = B2[idx%SIZEOF_B2]
arg5 = B2[(idx-N+1)%SIZEOF_B2]
arg6 = B2[(idx-N)%SIZEOF_B2]
arg7 = B2[(idx-N-1)%SIZEOF_B2]
arg8 = B2[(idx-1)%SIZEOF_B2]

if (j == borne_inf_j)
  arg1 = B2_vert[offset2+3]
  arg2 = B2_vert[offset2+2]
  arg4 = B2_vert[6]
  arg6 = B2_vert[6]
  arg7 = B2_vert[5]
  arg8 = B2_vert[5]
else if (j == borne_inf_j + 1)
  arg2 = B2_vert[offset2+3]
  arg7 = B2_vert[6]
  arg8 = B2_vert[6]
  offset2 += 4

// Si i == (borne_inf_i ou borne_inf_i+1), on doit
// utiliser l'information calculée lors du traitement
// des frontières horizontales
if (i == borne_inf_i)
  arg3 = B2_horiz[offset4+3*N+j]
  arg4 = B2_horiz[16]
  arg5 = B2_horiz[offset4+2*N+j]
  arg6 = B2_horiz[15]
  arg7 = B2_horiz[15]
  arg8 = B2_horiz[16]
else if (i == borne_inf_i + 1)
  arg5 = B2_horiz[offset4+3*N+j]
  arg6 = B2_horiz[16]
  arg7 = B2_horiz[16]
```

```
        Scalaire = ComputeEdges(arg0, arg1, arg2, arg3, arg4, arg5,
                                arg6, arg7, arg8)
        B3[idx%SIZEOF_B3] = Reverse(Scalaire)

        idx = idx-1-N
        arg0 = B3[(idx+N+1)%SIZEOF_B3]
        arg1 = B3[(idx+N)%SIZEOF_B3]
        arg2 = B3[(idx+N-1)%SIZEOF_B3]
        arg3 = B3[(idx+1)%SIZEOF_B3]
        arg4 = B3[idx%SIZEOF_B3]
        arg5 = B3[(idx-N+1)%SIZEOF_B3]
        arg6 = B3[(idx-N)%SIZEOF_B3]
        arg7 = B3[(idx-N-1)%SIZEOF_B3]
        arg8 = B3[(idx-1)%SIZEOF_B3]

        if (j == borne_inf_j)
          arg1 = B3_vert[offset3+1]
          arg2 = B3_vert[offset3]
          arg4 = B3_vert[31]
          arg6 = B3_vert[31]
          arg7 = B3_vert[2]
          arg8 = B3_vert[2]
        else if (j == borne_inf_j + 1)
          arg2 = B3_vert[offset3+1]
          arg7 = B3_vert[31]
          arg8 = B3_vert[31]
          offset3 += 2

        if (i == borne_inf_i)
          arg3 = B3_horiz[offset5+N+j]
          arg4 = B3_horiz[34]
          arg5 = B3_horiz[offset5+j]
          arg6 = B3_horiz[34]
          arg7 = B3_horiz[34]
          arg8 = B3_horiz[34]
        else if (i == borne_inf_i + 1)
          arg5 = B3_horiz[offset5+N+j]
          arg6 = B3_horiz[34]
          arg7 = B3_horiz[34]

        OUT[idx] = DetectRoots(arg0, arg1, arg2, arg3, arg4, arg5,
                               arg6, arg7, arg8)
}
```