

Titre: Optimisation des mémoires dans le flot de conception des systèmes multiprocesseurs sur puces pour des applications de type multimédia
Title:

Auteur: Bruno Girodias
Author:

Date: 2009

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Girodias, B. (2009). Optimisation des mémoires dans le flot de conception des systèmes multiprocesseurs sur puces pour des applications de type multimédia
Citation: [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie.
<https://publications.polymtl.ca/139/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/139/>
PolyPublie URL:

Directeurs de recherche: Gabriela Nicolescu, & El Mostapha Aboulhamid
Advisors:

Programme: Génie informatique
Program:

UNIVERSITÉ DE MONTRÉAL

**OPTIMISATION DES MÉMOIRES DANS LE FLOT DE
CONCEPTION DES SYSTÈMES MULTIPROCESSEURS
SUR PUCES POUR DES APPLICATIONS DE TYPE
MULTIMÉDIA**

BRUNO GIRODIAS

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)

AOÛT 2009

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

OPTIMISATION DES MÉMOIRES DANS LE FLOT DE CONCEPTION
DES SYSTÈMES MULTIPROCESSEURS SUR PUCES POUR
DES APPLICATIONS DE TYPE MULTIMÉDIA

présentée par : GIRODIAS Bruno

en vue de l'obtention du diplôme de : Philosophiæ Doctor

a été dûment acceptée par le jury d'examen constitué de :

Mme. CHERIET Farida, Ph.D., présidente

Mme. NICOLESCU Gabriela, Doct., membre et directrice de recherche

M. ABOULHAMID El Mostapha, Ph.D., membre et codirecteur de recherche

M. LANGLOIS J.M. Pierre, Ph.D., membre

M. ZILIC Zeljko, Ph.D., membre

DÉDICACE

À ma femme, ma famille et mes amis

REMERCIEMENTS

Un grand merci et toute mon estime au professeur et docteur Gabriela Nicolescu. Que cette thèse qu'elle a inspirée et encouragée de sa bienveillance soit un témoignage de ma reconnaissance. Elle m'a généreusement ouverte les portes de son laboratoire et m'a servi de guide dans l'élaboration de ce travail pour lequel elle n'a ménagé ni son temps, ni son aide. Merci également et tout mon respect au professeur et docteur El Mostapha Aboulhamid, codirecteur du projet, pour son temps et son précieux savoir. Je les remercie de leur soutien et de leurs conseils. Qu'ils trouvent ici l'expression de ma profonde gratitude.

Mes remerciements vont aussi au docteur Youcef Bouchebaba pour sa collaboration et son support, ainsi qu'aux docteurs Mathieu Brière et Sébastien Le Beux pour leur encouragement et la pertinence de leur avis et commentaires.

Je veux également exprimer toute ma reconnaissance au docteur Pierre Paulin et l'équipe de STMicroelectronics d'Ottawa, plus particulièrement Bruno Lavigueur, Olivier Benny et docteur Michel Langevin, pour leur collaboration, leur soutien et leurs conseils.

Enfin un remerciement du fond du cœur à ma famille; à ma femme pour ses encouragements et sa patience; à mes parents et à mes frères et sœurs pour leur constant soutien.

RÉSUMÉ

Les systèmes multiprocesseurs sur puce (MPSoC) constituent l'un des principaux moteurs de la révolution industrielle des semi-conducteurs. Les MPSoCs jouissent d'une popularité grandissante dans le domaine des systèmes embarqués. Leur grande capacité de parallélisation à un très haut niveau d'intégration, en font de bons candidats pour les systèmes et les applications telles que les applications multimédia. La consommation d'énergie, la capacité de calcul et l'espace de conception sont les éléments dont dépendent les performances de ce type d'applications. La mémoire est le facteur clé permettant d'améliorer de façon substantielle leurs performances. Avec l'arrivée des applications multimédias embarquées dans l'industrie, le problème des gains de performances est vital. La masse de données traitées par ces applications requiert une grande capacité de calcul et de mémoire. Dernièrement, de nouveaux modèles de programmation ont fait leur apparition. Ces modèles offrent une programmation de plus haut niveau pour répondre aux besoins croissants des MPSoCs, d'où la nécessité de nouvelles approches d'optimisation et de placement pour les systèmes embarqués et leurs modèles de programmation.

La conception niveau système des architectures MPSoCs pour les applications de type multimédia constitue un véritable défi technique. L'objectif général de cette thèse est de relever ce défi en trouvant des solutions. Plus spécifiquement, cette thèse se propose d'introduire le concept d'optimisation mémoire dans le flot de conception niveau système et d'observer leur impact sur différents modèles de programmation utilisés lors de la conception de MPSoCs. Il s'agit, autrement dit, de réaliser l'unification du domaine de la compilation avec celui de la conception niveau système pour une meilleure conception globale.

La contribution de cette thèse est de proposer de nouvelles approches pour les techniques d'optimisation mémoire pour la conception MPSoCs avec différents modèles de programmation. Nos travaux de recherche concernent l'intégration des techniques

d'optimisation mémoire dans le flot de conception de MPSoCs pour différents types de modèle de programmation. Ces travaux ont été exécutés en collaboration avec STMicroelectronics.

Cette recherche vise, plus particulièrement, deux types de modèle de programmation: (1) Symmetrical Multi Processing (SMP) et (2) streaming.

Dans un modèle de programmation de type SMP, l'approche de la conception du processeur et de l'optimisation mémoire d'application peut être combinée pour améliorer l'efficacité de la conception. Les techniques d'optimisation mémoire peuvent, ainsi, être adaptées au modèle de programmation de type SMP, pour une amélioration de l'efficacité globale du processeur (c'est-à-dire la localité des données, l'espace mémoire, la taille du code et le temps de traitement). Le haut niveau de parallélisme de l'environnement SMP explique que le processus d'adaptation des techniques soit un objet de controverse. Ces techniques conviennent à un environnement monoprocesseur, mais non pas nécessairement à un environnement multiprocesseur comme le modèle de programmation de type SMP. Notre travail d'adaptation des techniques d'optimisation mémoire nous a permis d'optimiser la taille de la mémoire et du code de l'application, et de réduire le temps d'exécution. Une étude sur l'effet des environnements multiprocesseur avec traitement multifil¹ montre que l'utilisation de ces techniques peut être fortement influencée par la granularité du parallélisme. Un raffinement de la granularité maximise le parallélisme, mais peut rendre l'exécution des techniques d'optimisation mémoire difficile, voir même, impossible. Nos techniques améliorent le taux de succès de l'antémémoire de 20% et réduisent le temps d'exécution de 50%. Pour obtenir ces résultats, nous avons utilisé comme outils d'expérimentation les applications Détection d'images lacunaires et Dématriçage.

Dans un environnement streaming (i.e. par flux), l'utilisation des techniques d'optimisation mémoire se compare à leur utilisation dans un environnement mono-

¹ « Multithreading » en anglais

processeur, mais l'étape du placement lors de la conception peut affecter grandement l'application des techniques d'optimisation mémoire potentielle. Pour maximiser la possibilité d'optimisation mémoire après le processus de placement, on peut utiliser des approches qui combinent l'optimisation mémoire avec l'étape du placement. Nos approches ont apporté des améliorations au niveau de l'optimisation mémoire, mais aussi au niveau de la communication et du temps d'exécution. Pour ces expérimentations, nous nous sommes servis de tests de performance générées aléatoirement et de l'application Dématriçage. Les résultats dépendent en grande partie de l'application utilisée. Nous avons obtenu une diminution de la taille de la mémoire de 36% et une réduction du coût de communication de 8%.

ABSTRACT

Multiprocessor systems-on-chip (MPSoC) are defined as one of the main drivers of the industrial semiconductors revolution. MPSoCs are gaining popularity in the field of embedded systems. Pursuant to their great ability to parallelize at a very high integration level, they are good candidates for systems and applications such as multimedia. Memory is becoming a key player for significant improvements in these applications (i.e. power, performance and area). With the emergence of more embedded multimedia applications in the industry, this issue becomes increasingly vital. The large amount of data manipulated by these applications requires high-capacity calculation and memory. Lately, new programming models have been introduced. These programming models offer a higher programming level to answer the increasing needs of MPSoCs. This leads to the need of new optimization and mapping approaches suitable for embedded systems and their programming models.

The overall objective of this research is to find solutions to the challenges of system level design of applications such as multimedia. This entails the development of new approaches and new optimization techniques. The specific objective of this research is to introduce the concept of memory optimization in the system level conception flow and study its impact on different programming models used for MPSoCs' design. In other words, it is the unification of the compilation and system level design domains.

The contribution of this research is to propose new approaches for memory optimization techniques for MPSoCs' design in different programming models. This thesis relates to the integration of memory optimization to varying programming model types in the MPSoCs conception flow. Our research was done in collaboration with STMicroelectronics.

This research targets two types of programming models: (1) Symmetrical Multi Processing (SMP) and (2) streaming.

In an SMP environment, the approach for processor's design and application's memory optimizations can be combined for more efficient design of the systems. Thus, the memory optimization techniques improving overall performance (i.e. data locality, memory space, code size and processing time) can be adapted for symmetrical multi-processing, improving the overall processor efficiency. The combination of these techniques is mainly challenged by the adaptation of memory optimization techniques to the high parallelism offered by environments like the SMP architecture. These techniques may be adequate for mono-processor environments, but are not necessarily adapted for multiprocessor environments like SMP. The adaptation of memory optimization techniques has allowed us to optimize the size of the memory and the application code, and reduce the execution time. A study of the effect of a multiprocessor and multithreading environment on these techniques shows that the granularity of parallelism can greatly influence their implementation. A refinement of the granularity maximizes the parallelism of the application, but may make the execution of memory optimization techniques difficult and even impossible. Our proposed techniques improve the cache success rate by 20% and reduce the processing time by 50%. The applications Cavity Detection and Demosaicing were used for experiments purposes.

In a streaming environment, the approach for application memory optimizations techniques is somehow similar to that of the mono-processor environment. However, the mapping phase in a multiprocessor design can affect the application of potential memory optimization to a great extent. Thus, approaches for combining memory optimization with mapping of data-driven applications can be used to maximize the possibility of memory optimization after the mapping process. The approaches presented in this thesis have improved the memory optimization, but have also improved the communication and processing time. For these experiments, we used randomly generated benchmarks and the Demosaicing application. Results depend greatly on the application used, but the experiments performed in our thesis work showed a decrease in memory size by 36% and a reduction in communication cost by 8%.

TABLE DES MATIÈRES

DÉDICACE	III
REMERCIEMENTS	IV
RÉSUMÉ	V
ABSTRACT	VIII
TABLE DES MATIÈRES	X
LISTE DES TABLEAUX	XIV
LISTE DES FIGURES	XV
LISTE DES SIGLES ET ABRÉVIATIONS	XVIII
LISTE DES ANNEXES	XIX
INTRODUCTION	1
CHAPITRE 1. REVUE CRITIQUE DE LA LITTÉRATURE	16
1.1 L'intégration des techniques d'optimisation mémoire dans le flot de conception des MPSoCs	16
1.1.1 Outils de compilation	16
1.1.2 Environnement de conception	18
1.2 L'optimisation mémoire	25
1.2.1 Optimisation mémoire dans une architecture monoprocesseur	25
1.2.2 Optimisation mémoire dans une architecture multiprocesseur	27
1.2.3 Évaluation des performances des techniques d'optimisation mémoire	28
1.3 L'optimisation mémoire dans le flot de conception des MPSoCs	30
CHAPITRE 2. CONCEPTS DE BASE ET VUE GLOBALE DU PROJET	33
2.1 Concepts de base	33
2.1.1 Système multiprocesseur sur puce (MPSoC)	33

2.1.2	Conception niveau système.....	36
2.1.3	Modèle de programmation.....	37
2.1.4	Architecture Mémoire.....	38
2.1.5	Optimisation MPSoC.....	40
2.2	Vue globale du projet.....	44
2.2.1	Flot de conception.....	45
2.2.2	Types d'application.....	47
CHAPITRE 3. ARTICLE 1: MULTIPROCESSOR, MULTITHREADING AND MEMORY OPTIMIZATION FOR ON-CHIP MULTIMEDIA APPLICATIONS.....		52
3.1	Introduction.....	53
3.2	Related work.....	54
3.3	Memory optimization techniques.....	58
3.3.1	Loop fusion.....	58
3.3.2	Buffer allocation.....	62
3.4	Techniques impacts.....	64
3.4.1	Computation time.....	64
3.4.2	Code size increase.....	65
3.5	Improvement in optimization techniques.....	65
3.5.1	Data partitioning.....	66
3.5.2	Modulo operators elimination.....	69
3.5.3	Unimodular transformation.....	71
3.6	Parallelization.....	72
3.6.1	Initial code.....	73
3.6.2	Code with fusion.....	74
3.6.3	Code with fusion and buffer allocation.....	78
3.7	Applications.....	80
3.7.1	General.....	80
3.7.2	Cavity Detection.....	81

3.7.3	Demosaicing.....	82
3.8	Experimental results.....	82
3.8.1	Memory optimization technique improvements	83
3.8.2	Multiprocessor and multithreading effect	86
3.8.3	Processing time	95
3.9	Analysis summary	103
3.10	Conclusions and future work	104
CHAPITRE 4. ARTICLE 2: INTEGRATING MEMORY OPTIMIZATION WITH MAPPING ALGORITHMS FOR MULTI-PROCESSORS SYSTEM-ON-CHIP		109
4.1	Introduction.....	110
4.2	Related Work	112
4.3	Memory Optimization and Mapping.....	114
4.3.1	Memory optimization.....	114
4.4	Design Methodology.....	117
4.4.1	Application Capture and Architecture Specification	119
4.4.2	Application Capture Transformation	119
4.4.3	Application Mapping	120
4.4.4	Memory Optimization.....	121
4.5	Graph Transformations for Memory Optimization.....	121
4.5.1	Problem Formulation	121
4.5.2	Transformation Algorithm Formulation (f)	124
4.6	Evolution Algorithm for Memory Optimization.....	129
4.7	Experimental Results	131
4.7.1	Generated task graph with heuristic approach	133
4.7.2	Demosaicing with heuristic approach	137
4.7.3	Demosaicing with evolutionary approach.....	143
4.8	Discussion	148
4.9	Conclusion and Future Work	149

CHAPITRE 5. DISCUSSION GÉNÉRALE	152
5.1 Architecture avant-gardiste	152
5.2 Modèle de programmation, placement et optimisation mémoire dans le flot de conception des MPSoCs.	153
5.3 Positionnement dans le contexte actuel de l'optimisation mémoire pour les applications de type streaming	156
5.4 Automatisation	157
CONCLUSION ET RECOMMANDATIONS	158
LISTE DE RÉFÉRENCES	163
ANNEXES	170

LISTE DES TABLEAUX

Tableau 0.1 Modèle de programmation	12
Tableau 2.1 SMP vs Streaming	38
Tableau 5.1 Interconnexion Électrique versus Optique	153

LISTE DES FIGURES

Figure 0.1 Architecture générale d'un système multiprocesseur sur puce.....	2
Figure 0.2 Écart Mémoire	4
Figure 0.3 Objectifs: unification de 2 mondes	11
Figure 1.1 L'environnement de STMicroelectronics [1]	19
Figure 1.2 Daedalus[24, 25].....	22
Figure 1.3 Compaan[26, 27]	24
Figure 2.1 Traitement classique (gauche) et traitement multifil (droite)[68].	35
Figure 2.2 Exemple d'un code et de son domaine d'itération.....	43
Figure 2.3 Flot de conception général.....	45
Figure 2.4 Détection d'images lacunaires	50
Figure 2.5 Dématriçage[76]	51
Figure 3.1 An example of loop fusion	59
Figure 3.2: An example of 3 nested loops	60
Figure 3.3: Partitioning after loop fusion.....	61
Figure 3.4: An example of buffer allocation.....	62
Figure 3.5: Buffer allocation for array B	63
Figure 3.6: Classic partitioning.....	66
Figure 3.7: Paper's partitioning	67
Figure 3.8: Buffer allocation for array B with new partitioning.....	68
Figure 3.9: Sub-division of processor P1's block	69
Figure 3.10: Elimination of modulo operators.....	70
Figure 3.11: Execution order (a) without fusion (b) after fusion and (c) after unimodular transformation	71
Figure 3.12: Part of the initial code partitioned on 2 CPUs and 2 threads.....	74
Figure 3.13: Unimodular transformation	76
Figure 3.14: Examples of partitioning	77
Figure 3.15: Buffer allocation in fine grain approach.....	79

Figure 3.16: Cavity Detection Application	81
Figure 3.17: Demosaicing Application	82
Figure 3.18: DCache hit ratio #CPU=4.....	84
Figure 3.19: Processing time #CPU=4.....	85
Figure 3.20: Cavity Detection DCache hit ratio.....	87
Figure 3.21: DCache hit ratio #CPUs=1	89
Figure 3.22: DCache hit ratio #CPUs=4	90
Figure 3.23: DCache hit ratio #CPUs=16	90
Figure 3.24: DCache hit ratio #CPUs varies	91
Figure 3.25: Demosaicing DCache hit ratio	92
Figure 3.26: DCache hit ratio #CPUs=1	93
Figure 3.27: DCache hit ratio #CPUs=4	94
Figure 3.28: DCache hit ratio #CPUs=16	94
Figure 3.29: DCache hit ratio #CPUs varies	95
Figure 3.30: Processing time (ns)	96
Figure 3.31: Processing time (ns) #CPUs=1	97
Figure 3.32: Processing time (ns) #CPUs=4.....	98
Figure 3.33: Processing time (ns) #CPUs=16	98
Figure 3.34: Processing time (ns) #CPUs varies.....	99
Figure 3.35: Processing time (ns)	100
Figure 3.36: Processing time (ns) #CPUs=1	101
Figure 3.37: Processing time (ns) #CPUs=4.....	102
Figure 3.38: Processing time (ns) #CPUs=16	102
Figure 3.39: Processing time (ns) #CPUs varies.....	103
Figure 4.1 An example of 2 nested loops	115
Figure 4.2 An example of buffer allocation	116
Figure 4.3 Design Flow.....	118
Figure 4.4 Input Graph Annotations	122

Figure 4.5 Output graph	123
Figure 4.6 Genetic algorithm approach.....	130
Figure 4.7 Application Task Graph.....	133
Figure 4.8 Memory Gain.....	134
Figure 4.9 Communication	135
Figure 4.10 Load Variance.....	136
Figure 4.11 Physical Link	137
Figure 4.12 Demosaicing Application Task Graph.....	137
Figure 4.13 Memory Gain (Demosaicing)	138
Figure 4.14 Communication (Demosaicing).....	139
Figure 4.15 Load Variance (Demosaicing).....	140
Figure 4.16 Physical Link (Demosaicing)	141
Figure 4.17 Brief comparison of the approach's effect on the different mapping algorithms	142
Figure 4.18 Pareto Curves - Three metrics (i.e. Memory, Communication and Load)	144
Figure 4.19 Pareto Curves - Two metrics (i.e. Memory and Communication)	145
Figure 4.20 Pareto Curves - Two metrics (i.e. Memory and Load)	146
Figure 4.21 Pareto Curves - Two metrics (i.e. Communication and Load)	147

LISTE DES SIGLES ET ABRÉVIATIONS

CP	Constraint Programming
CPU	Central Processing Unit
DSP	Digital Signal Processor
FPE	Fusion Preventing Edge
ILP	Integer Linear Programming
ITRS	International Technology Roadmap for Semiconductors
KPN	Khan Process network
MPSoC	Multi-Processor System on Chip
MPAssign	Multi-Processor Assignment
NoC	Network on Chip
PE	Processing Element
ONoC	Optical Network on Chip
SMP	Symetrical Multi-Processor
SoC	System on Chip
UC	Unité de Calul

LISTE DES ANNEXES

ANNEXE 1 : PUBLICATIONS	170
ANNEXE 2 : COMPLÉXITÉ DE L'ALGORITHME	173

INTRODUCTION

Contexte

L'ITRS (International Technology Roadmap for Semiconductor) définit les systèmes multiprocesseurs sur puces (MPSoC²) comme étant un des principaux promoteurs de la révolution industrielle des semi-conducteurs parce qu'ils permettent l'intégration de fonctionnalités complexes sur une seule puce [2]. Grâce à leurs performances, les MPSoCs acquièrent une popularité grandissante dans le domaine des systèmes embarqués. Nous ne soupçonnons pas toujours l'usage de plus en plus répandu que nous en faisons au quotidien. Dans le secteur du multimédia, ils sont omniprésents dans les télévisions, les consoles de jeux et les appareils photographiques. Dans le secteur de la télécommunication, on les retrouve dans les cellulaires, les bornes d'accès sans-fil et les antennes de type intelligent. Dans le domaine médical, on les utilise dans les instruments d'imagerie, de numérisation et d'analyse. Leur grande capacité de parallélisme à très haut niveau d'intégration, en font de bons candidats pour les systèmes à plateformes complexes comme les applications multimédia, les applications sans fil et les applications médicales[3]. La somme importante des données manipulées par ce type d'applications exige une grande puissance de calcul, une mémoire de grande capacité et un nombre significatif d'accès à la mémoire externe pour chaque élément de calcul dans l'architecture MPSoC[4].

² Venant de l'anglais Multi-Processors System-on-Chip

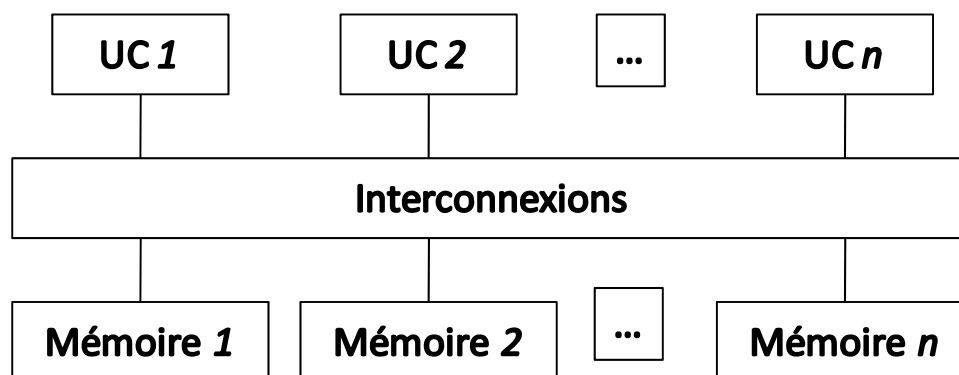


Figure 0.1.1 Architecture générale d'un système multiprocesseur sur puce

La Figure 0.1.1 illustre un MPSoC générique typique et ses divers composants: processeurs embarqués (ARM, MIPS, PowerPC...), mémoires partagées sur puce et leurs interconnexions par différentes sortes de bus (réseau sur puce (NoC³), STBUS, XBar, AMBA...).

Pour gérer la complexité des systèmes multiprocesseurs sur puce, la tendance actuelle est d'élever le niveau d'abstraction. Le concepteur se voit contraint de faire des suppositions exactes pour respecter les concepts et aspects de bas niveau lors de la conception de MPSoCs. L'augmentation du niveau conceptuel d'abstraction définit la conception au niveau système. Les principaux objectifs de la conception au niveau système sont de respecter les normes requises de performance et de consommation de puissance du système, le coût de production et les délais de mise en marché [5].

Problématique lors la conception système des MPSoCs

Les objectifs, mentionnés à la fin du paragraphe précédent, sont difficiles à respecter lors de la conception de MPSoCs. Le concepteur doit surmonter plusieurs difficultés aux différentes phases de la conception : conception de l'architecture, conception du logiciel embarqué, l'accommodation d'architecture et du logiciel embarqué, développement de

³ Venant de l'anglais Network-on-Chip

concepts et de techniques propres à la conception des systèmes MPSoCs. Nous discuterons plus en détails de ces difficultés de conception dans ce chapitre.

La conception d'architecture

Selon la loi bien connue de Moore, la puissance de calcul double tous les 18 mois [6]. Ce constat ne peut que motiver les concepteurs d'architecture MPSoCs qui cherchent sans cesse à augmenter la vitesse de traitement. Ce gain de vitesse ne s'obtient pas sans coût supplémentaire. La loi de Moore énonce encore que la consommation d'énergie des unités de calcul double elle-aussi tous les dix-huit mois [6]. Malheureusement, il n'y a pas de compromis possible : la conception d'architecture est énergivore. On doit s'en accommoder.

La mémoire est un facteur déterminant pour la performance globale des architectures MPSoCs. Elle joue un grand rôle dans la taille de la surface de conception (i.e. taille du circuit) et la consommation d'énergie, deux éléments essentiels lors de la conception de MPSoCs. La vitesse de la mémoire est un élément important. L'écart des performances des unités de calcul et de la mémoire ne cesse de s'élargir [6]. Le gain en vitesse des unités de calcul surpasse largement celui de la mémoire. Si la vitesse des unités de calcul double tous les deux ans, celle de la mémoire double tous les six ans. L'écart de vitesse croît de façon exponentielle [6], tel que montré à la Figure 0.1.2.

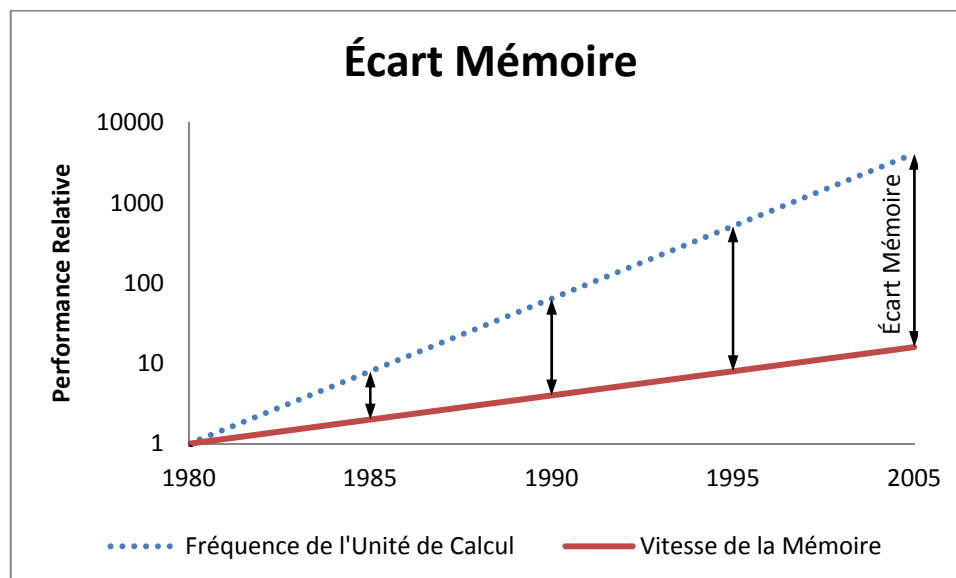


Figure 0.1.2 Écart Mémoire

La conception de logiciel embarqué

La conception de logiciel embarqué implique un concept nouveau et important dans la conception des MPSoCs. Les concepteurs de puces étaient habitués à travailler sur les solutions pour résoudre les problèmes de conception d'architecture. Ils se souciaient peu de la partie logicielle. Dans un MPSoC, le matériel et le logiciel peuvent s'avérer être une solution pour résoudre les problèmes. Pour les concepteurs de logiciels, la partie matérielle leur est souvent complètement abstraite. Le logiciel qui sera embarqué dans le MPSoC doit être extrêmement fiable et se plier à de sérieuses contraintes matérielles telles que les contraintes de synchronisation et de consommation d'énergie. L'intrication des supports logiciel et matériel dans la conception des MPSoCs est source de difficultés, mais un processus informatique fort intéressant d'un point de vue scientifique. Il faut développer de nouvelles méthodologies pour faciliter, harmoniser et rentabiliser les tâches conjointes des concepteurs de matériels et de logiciels [5]. Le coût du développement du logiciel embarqué va s'ajouter au coût de développement d'une plate-forme système sur puce. Les systèmes actuels sur puce supportent de plus en plus

de nouvelles fonctionnalités. Celles-ci répondent à une demande du milieu informatique, des réseaux de communications et d'un marché qui se diversifie. La loi de Moore peut s'appliquer à la croissance de complexité du matériel dans les SoCs: le nombre de transistors augmente de 56% par année; la croissance annuelle de complexité du logiciel dans les SoCs atteint 140% et compte aujourd'hui pour 75% du coût de développement de la plupart des MPSoCs [7].

La complexité du logiciel embarqué nécessite une réutilisation des plateformes spécifiques à un domaine d'application, l'exploitation de modèles de programmation adaptés aux différents domaines d'application ainsi que la définition et le développement d'outils de conception automatique pour chaque type de plateforme et de modèle de programmation.

L'accommodation de l'architecture et du logiciel embarqué

L'étape de la programmation du logiciel embarqué peut avoir un impact majeur sur les performances des systèmes basés sur des architectures MPSoCs. Pour remédier à ce problème, les compilateurs se composent d'outils efficaces pour analyser, optimiser et placer un logiciel embarqué. Malheureusement, les compilateurs existants requièrent l'ajout de nouvelles fonctionnalités pour opérer sur des architectures MPSoCs. Tout un travail de recherche sera nécessaire avant de pouvoir tirer profit, dans les limites des ressources disponibles, du degré de parallélisme offert par ce type d'architecture [5].

La définition de concepts et techniques propres à la conception des systèmes MPSoCs

Le recours aux systèmes MPSoCs suppose l'élaboration de nouveaux concepts et le développement de nouvelles techniques. Qu'il s'agisse de modèle d'abstraction, de techniques d'optimisation mémoire, de stratégie de placement des applications sur les

architectures ou de l'automatisation du processus de conception, les innovations nécessaires constituent autant de défis pour la recherche informatique.

Abstraction

L'abstraction réduit de beaucoup les coûts de production. Les modèles de programmation donnent une vue abstraite du matériel au programmeur. Les modèles de programmation de plateforme de haut niveau aident à développer le logiciel tout en exploitant l'abstraction de systèmes hétérogènes constitués d'unités de calcul, de mémoires locales ou partagées, de canaux de communication et d'entrées/sorties. Les modèles de programmation doivent cacher les détails de l'implémentation tout en exposant le comportement de l'architecture MPSoC.

L'hétérogénéité des différentes unités de calcul et des divers outils de programmation de ces unités de calcul ne doit pas être apparente. Les modèles doivent masquer le bas niveau d'abstraction des mécanismes de communication entre les unités de calcul, les composantes de stockage et des entrées/sorties. Cependant les modèles de programmation doivent afficher certaines caractéristiques de l'architecture, comme le niveau et le type de parallélisme et le degré d'abstraction des fonctionnalités de tous ses composants (les accélérateurs matériels, les canaux de communication, les composantes de stockage et les entrées/sorties). Notre travail de thèse porte sur deux modèles de programmation d'usage courant bien établis: (1) le modèle de programmation de type SMP⁴ et (2) le modèle de programmation de type streaming. Nous donnerons dans le chapitre 2 une description de ces deux modèles.

⁴ Venant de l'anglais Symetric Multi-Processing

Techniques d'optimisation mémoire

Les techniques d'optimisation permettent d'adapter le logiciel embarqué aux architectures. En ce qui nous concerne, nous utiliserons les techniques d'optimisation visant à diminuer la taille mémoire et le nombre d'accès à la mémoire. Elles améliorent les performances du système, diminuent la consommation de puissance et réduisent la taille du circuit.

La technologie actuelle nous permet de construire des systèmes embarqués avec des antémémoires à large capacité. Pour des applications avec des accès mémoire réguliers, la grande disponibilité des antémémoires s'avère efficace. Elle est, par contre, inefficace ou mal adaptée pour des applications ayant des accès mémoire irréguliers ou utilisant des motifs d'accès répétitifs ne convenant pas aux algorithmes de gestion des antémémoires. Pour les systèmes embarqués, disposer d'antémémoires à grande capacité ou d'une hiérarchie étendue de mémoire, autorise des temps d'accès variables aux données. Il en résulte une prédiction plus complexe du temps d'exécution du système ce qui est un élément important pour les applications dans un environnement temps réel. Dans le cas d'applications multimédia et sans-fil, l'accès mémoire se fait de façon régulière, mais ne correspond pas nécessairement aux politiques de remplacement utilisées dans les antémémoires. Les applications multimédia consomment beaucoup de mémoire, car elles manipulent des flux de son, de vidéo ou d'images fixes très gourmands en mémoire. L'ITRS [2] prévoit que 94% de la surface des systèmes sur puce sera dédiée à la mémoire vers 2014. La consommation d'un accès à la mémoire dépend du niveau de la hiérarchie mémoire. Plus les données utiles se situent à un bas niveau dans la hiérarchie mémoire, plus l'accès à ces données sera lent et coûteux en consommation d'énergie [8]. Il est donc bénéfique d'optimiser la localité temporelle des accès pour concentrer les données à un plus haut niveau dans la hiérarchie et d'optimiser la quantité de mémoire pour diminuer le nombre des niveaux hiérarchiques de mémoire.

Faire des transformations au niveau du code source d'application multimédia permet de rentabiliser au maximum les ressources de la hiérarchie mémoire. Ces transformations sont des modifications de partie de code source permettant de rendre le code source plus efficace lors de son exécution. Dépendant du modèle de programmation, ces transformations peuvent affecter l'organisation du système et la répartition des tâches entre les différentes unités de calcul.

Placement des applications

Le placement joue aussi un grand rôle dans les performances du système et la consommation de puissance. Le placement consiste à placer une application sur une architecture comportant différentes unités de calcul. Différentes métriques doivent être minimisées ou maximisées selon une ou plusieurs fonctions objectives. Les métriques les plus populaires sont le coût du temps d'exécution, le coût de la charge de chaque unité de calcul, le coût des communications, les coûts de consommation d'énergie et le coût de la taille de la surface de conception.

Le placement est, par ailleurs, une étape importante dans la conception au niveau système. Il sert d'intermédiaire entre les abstractions à haut niveau et l'implémentation à bas niveau. Plus le niveau d'abstraction augmente, plus l'étape de placement se complexifie. L'étape de placement est facilitatrice d'abstraction au niveau système.

Outils de conception

La création d'outils de conception est indispensable pour la mise en place de techniques et d'approches propres au domaine MPSoCs. Les outils de conception favorisent l'élaboration de nouveaux concepts. Des outils implantant les concepts/aspects discutés dans les sections précédentes (l'abstraction, techniques d'optimisation mémoire et placement des applications) sont nécessaires.

Problématique dans les applications multimédia

Les applications de type multimédia jouissent d'une popularité grandissante; elles envahissent les technologies de notre quotidien et accaparent une grande part du marché. De futures technologies comme le 4G sont déjà en cours de préparation, mais leur mise au point doit surmonter de nombreux obstacles. Le 4G introduit la convergence de plusieurs technologies. Il intègre différents modes de communication sans fil: réseaux intérieurs, (WiFi et Bluetooth), systèmes cellulaires, transmissions par radio, communications par satellite [9]. Les applications utilisant la technologie 4G seront riches en multimédia.

L'intégration de toutes ces technologies demande une grande puissance de calcul. La puissance de calcul requise par ces nouvelles plateformes outrepassa la loi de Moore [10]. Les impératifs physiques vont constituer un obstacle majeur à l'implantation de ces puissantes unités de calcul si l'on ne parvient pas à réduire leur consommation d'énergie [10].

La plupart de ces applications vont opérer sur des dispositifs portables (cellulaires, ordinateurs portables, assistants électroniques, etc.). Il sera difficile de concevoir des systèmes sur puces performants, consommant peu d'énergie et de coût de conception réduit [11].

L'interaction de la communication et de la mémoire contribue en grande partie aux coûts liés à la taille de surface de ces applications et à leur consommation d'énergie [12, 13]. En effet, le système mémoire est une structure propice à la gestion des problèmes de temps réel et de consommation d'énergie [4, 14].

Les applications multimédia comptent souvent plusieurs boucles imbriquées. Les techniques actuelles de compilation pour les architectures parallèles (MPSoCs ou autres)

examinent séparément chaque boucle imbriquée et, ce faisant, ne parviennent pas à saisir l'interaction entre les différentes boucles.

On s'attend à ce que l'architecture des MPSoCs devienne l'architecture prédominante des applications de type multimédia sans fil ou autre. Les MPSoCs offrent un bon équilibre entre puissance, consommation d'énergie et flexibilité. Les chercheurs vont s'ingénier à développer des outils et des méthodologies de conception qui soient adaptés à ce type d'architecture.

Développer des outils et des méthodologies pour toutes les applications multimédia existantes est une tâche difficile voire même impossible. Un grand nombre de ces applications ont des caractéristiques communes. Ces caractéristiques sont les suivantes : (1) elles sont orientées données (i.e. beaucoup de données), (2) elles recourent fréquemment à des structures de boucles imbriquées et (3) elles utilisent des tableaux temporaires pour le traitement intermédiaire des données.

Objectifs

L'objectif général de cette thèse est de trouver des solutions aux problèmes de conception au niveau système des architectures MPSoCs pour des applications de type multimédia.

De façon plus spécifique, nous voulons introduire le concept d'optimisation mémoire dans le flot de conception au niveau système et observer son impact sur différents modèles de programmation utilisés lors de la conception de MPSoCs. Comme illustré par la Figure 0.1.3, il s'agit d'unifier les domaines de la compilation et de la conception au niveau système pour une meilleure conception globale.

Cette thèse porte avant tout sur les optimisations mémoire dans les architectures MPSoCs. Ces optimisations visent, en réduisant la taille de la mémoire et en limitant le nombre d'accès mémoire, à diminuer la communication et la consommation d'énergie.

Nous présentons des techniques d'optimisation spécialement adaptées aux architectures MPSoCs et nous proposons une approche qui combine ces techniques à l'étape de placement dans la conception des MPSoCs.

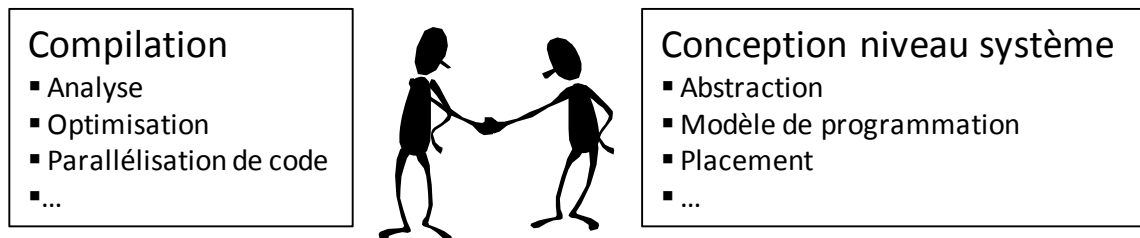


Figure 0.1.3 Objectifs: unification de 2 mondes

Contributions

Les trois principales contributions de cette thèse portent sur : (1) l'intégration des techniques d'optimisation mémoire dans le flot de conception des MPSoCs, (2) l'optimisation mémoire dans le flot de conception des MPSoCs à modèle de programmation de type SMP et l'évaluation du traitement multifil parfois présent dans les architectures utilisant ce modèle de programmation et (3) l'optimisation mémoire dans le flot de conception des MPSoCs à modèle de programmation streaming.

L'intégration des techniques d'optimisation mémoire dans le flot de conception des MPSoCs

La première contribution consiste en l'intégration des techniques d'optimisation mémoire dans le flot de conception des MPSoCs. Nous proposons une nouvelle approche et des techniques novatrices d'optimisation pour la conception de MPSoCs. Cette approche nécessite l'extension de techniques existantes applicables aux architectures classiques monoprocesseurs afin de les adapter aux architectures MPSoCs.

Nous utiliserons donc, aux différentes étapes du flot de conception, plusieurs des outils actuellement disponibles. Ces outils se prêtent bien à nos approches et à nos techniques d'optimisation.

Cette contribution comporte deux parties. La première partie s'intéresse aux techniques d'optimisation mémoire pour les MPSoCs à modèle de programmation de type SMP. La deuxième partie est consacrée aux techniques d'optimisation mémoire pour les MPSoCs à modèle de programmation de type streaming. Étant données les caractéristiques des deux modèles de programmation, l'intégration s'exécute différemment.

Dans un modèle de programmation de type SMP, le parallélisme se fait au niveau des données (Tableau 0.1). Chaque unité de calcul exécute un même code; le placement de l'application est implicite. Dans un modèle de programmation de type streaming, le parallélisme se fait au niveau des tâches⁵. Chaque unité de calcul exécute un ou plusieurs codes différents; le placement de l'application est explicite. Le placement fixe la position de la tâche sur l'architecture basée sur des fonctions objectives de coût. Ces fonctions tentent d'optimiser certaines métriques comme l'équilibrage de la charge et la communication entre chaque unité de calcul.

Tableau 0.1 Modèle de programmation

	Niveau du parallélisme	Exécution des techniques d'optimisation	Placement
SMP	Données	Sur l'ensemble de plusieurs unités de calcul	Implicite
Streaming	Tâches	Sur chaque unité de calcul séparément	Explicite

⁵ Tâches : unité de base dans la structure d'un programme streaming exécutant une action (i.e. ensemble de calcul) effectuée par le processeur.

L'optimisation mémoire dans le flot de conception des MPSoCs à modèle de programmation de type SMP

La deuxième contribution consiste en l'optimisation mémoire dans le flot de conception des MPSoCs à modèle de programmation de type SMP. Les techniques d'optimisations mémoire et la conception des MPSoCs à modèle de programmation de type SMP peuvent se combiner pour améliorer l'efficacité de la conception du système global. Comme indiqué par le Tableau 0.1, dans un modèle de programmation de type SMP, le placement de l'application est implicite, mais certaines techniques de base d'optimisation mémoire ne sont pas toujours adaptées à l'architecture MPSoCs. Les techniques d'optimisation s'appliquent sur un ensemble d'unités de calcul. Cette contribution entend résoudre le problème difficile de l'adaptation des techniques d'optimisation mémoire à des environnements fortement parallèles. Nous présentons une analyse approfondie de l'impact des environnements multiprocesseurs et des traitements multifil sur les techniques d'optimisation de mémoire. Nous décrivons les modifications à effectuer pour adapter ces différentes techniques. L'utilisation judicieuse des techniques d'optimisation et du traitement multifil pour des applications spécifiques, par exemple les applications multimédia, peut diminuer la taille des mémoires et le nombre des accès mémoires tout en augmentant les performances. Une évaluation des techniques proposées a été faite sur des applications multimédia telles que les applications de Détection d'images lacunaires et de Dématriçage.

L'optimisation mémoire dans le flot de conception des MPSoCs à modèle de programmations streaming

La troisième contribution consiste en l'optimisation mémoire dans le flot de conception des MPSoCs à modèle de programmation de type streaming. Les techniques d'optimisation mémoire et la conception des MPSoCs à modèle de programmation de type streaming peuvent également se combiner pour améliorer l'efficacité de la

conception du système global. Dans un modèle de programmation de type streaming, les techniques de base d'optimisation mémoire n'ont pas besoin d'être adaptées. Comme indiqué par le Tableau 0.1, les techniques d'optimisation s'appliquent sur chaque unité de calcul séparément. Le placement de l'application est explicite. Réussir l'intégration des techniques d'optimisation mémoire dans un flot de conception où l'étape de placement de l'application contribue énormément à la performance constitue tout un défi que nous entendons relever. Notre thèse démontre que dans un environnement streaming, l'étape de placement de l'application dans la conception de ce type d'architecture MPSoCs, doit tenir compte des techniques d'optimisation de mémoire qui peuvent, par la suite, être appliquées sur l'application. Nous proposons des approches pour combiner les techniques d'optimisation et l'étape de placement. Avec ces approches, on réalise que la mise en œuvre de ces techniques pendant l'étape de placement, augmente la possibilité de les réutiliser après cette étape. Des améliorations significatives sont obtenues au niveau du gain de la mémoire, mais aussi sur la communication et la charge des unités de calcul. Une évaluation de nos approches a été faite sur des tests de performance (de type multimédia) générées aléatoirement, mais aussi sur des applications réelles, par exemple application de Dématriçage.

Plan du document

Cette thèse compte 7 chapitres. Le chapitre 1 dresse un bilan des acquis scientifiques dans le domaine informatique qui nous concerne. Nous présentons les articles publiés les plus importants en la matière. Les uns survolent différentes problématiques, d'autres proposent diverses techniques et approches. Le chapitre 2 expose les concepts de base et l'environnement de travail de notre projet de recherche. Une introduction à la conception des multiprocesseurs sur puces, aux outils et modèles disponibles, aux techniques d'optimisation de mémoire permettra au lecteur d'avoir une meilleure compréhension de l'objectif générale de notre thèse. Le chapitre 3 reproduit le texte intégrale d'un premier article que nous avons publié et qui aborde les techniques d'optimisation mémoire pour

les MPSoCs à modèle de programmation de type SMP (partie 1 ou premier volet de notre thèse). Le chapitre 4 reproduit le texte d'un deuxième article sur les techniques d'optimisation mémoire pour les MPSoCs à modèle de programmation de type streaming (partie 2 ou deuxième volet de notre thèse). Le chapitre 5 est un chapitre de discussion générale sur l'ensemble de la thèse. Le chapitre 6 discute de l'orientation future des projets de recherche et des perspectives d'avenir; il énumère les objectifs qui ont déjà été atteints; et il se conclut par une récapitulation des travaux présentés dans cette thèse.

CHAPITRE 1. REVUE CRITIQUE DE LA LITTÉRATURE

Dans ce chapitre, nous présenterons une revue de la littérature sur les outils disponibles pour la conception de MPSoCs et pour les optimisations mémoire dans le flot de conception de MPSoCs.

1.1 L'intégration des techniques d'optimisation mémoire dans le flot de conception des MPSoCs

1.1.1 Outils de compilation

Les compilateurs sont des outils informatiques qui permettent entre autres d'analyser des programmes et d'optimiser les accès mémoire et par des techniques d'extension, d'automatiser le processus d'optimisation mémoire.

Les compilateurs actuels ne ciblent pas nécessairement les architectures MPSoCs. Certains compilateurs intègrent des optimisations mémoire, mais uniquement pour des environnements monoprocesseurs. Les techniques d'optimisation mémoire que nous proposons peuvent être implémentées dans ces outils.

1.1.1.1 Stanford University Intermediate Format (SUIF)[15]

SUIF[15] est un compilateur utilisé pour la recherche et le développement de techniques d'optimisation de compilation. L'environnement très modulaire de SUIF favorise l'élaboration de techniques d'optimisation qui s'intègrent facilement avec l'infrastructure. Des modules précompilés sont disponibles comme le module objet orienté OSUIF et le module orienté machine MachSUIF. Du fait de son infrastructure, ce compilateur se prête très bien aux travaux de collaboration en recherche et développement des techniques de compilation, techniques basées sur une représentation intermédiaire, d'où la dénomination SUIF (Stanford University Intermediate Format). Il permet une réutilisation optimale du

code en fournissant une abstraction utile et un cadre pour développer de nouvelles passes de compilateur.

SUIF est un compilateur très intéressant du fait que les techniques d'optimisation développées pour une architecture MPSoCs peuvent être intégrées et permettent d'automatiser le processus d'optimisation pour des applications multimédias.

1.1.1.2 PIPS [16]

L'objectif du projet PIPS était de développer un outil libre, ouvert et extensible pour l'analyse et la transformation automatique d'applications de traitement de signal. L'outil PIPS convient tout particulièrement aux personnes intéressées à la compilation, vérification ou optimisation source à source de programme, à la rétro-ingénierie, et à la parallélisation. Il permet de réduire les coûts et les délais d'exécution et d'optimisation.

Plusieurs équipes de recherche européennes l'utilisent pour le développement de nouvelles techniques d'analyse ou de transformation de programme (le Commissariat à l'Énergie Atomique de la Direction des Applications Militaires ou CEA-DAM, l'Université de Southampton, l'École Nationale Supérieure des Télécommunications de Bretagne/ENST Bretagne, et l'École Normale Supérieure de Cachan/ENS Cachan).

1.1.1.3 PLoTo [17, 18]

PLoTo est un outil de parallélisation automatique basé sur le modèle polyhédral. Le modèle polyhédral est une représentation géométrique pour les programmes qui utilisent l'algèbre linéaire et la programmation linéaire pour l'analyse et les transformations de haut niveau [18]. L'outil PLoTo permet de faire des optimisations et des transformations code source à source sur des séquencements de boucles imbriquées. Il sert à paralléliser l'application à gros grains et à optimiser la localité des données. Il repère les bouts de code dont les structures possèdent des boucles imbriquées avec des accès statiques affines et effectue des transformations pour du pavage et de la fusion de boucle efficace. L'outil

PLuTo, par l'intermédiaire de pragmas OpenMP, peut paralléliser automatiquement un code séquentiel. Bien qu'il soit entièrement automatique (C à C OpenMP), il dispose de quelques paramètres permettant des ajustements de la taille des tuiles, du facteur de déroulement de boucle et du niveau de fusion à appliquer. Il utilise l'outil CLoog pour la génération de code. Pour l'instant, l'outil PLuTo n'accepte qu'un très petit sous-ensemble de C - des séquences de boucles imbriquées arbitrairement à accès affine exclusif. Par contre, le modèle polyédral n'est pas limité à ce type d'applications

1.1.2 Environnement de conception

Les environnements de conception que nous allons examiner permettent d'intégrer et d'évaluer les différents concepts et approches présentés dans notre thèse. Ce sont des outils pour un flot complet de conception, ce que ne sont pas les compilateurs.

Les environnements de conception sont des éléments essentiels, mais non pas prépondérants de notre projet. Les environnements suivants ne visent pas l'intégration des concepts d'optimisation mémoire, mais restent complémentaires à nos travaux. Ces environnements facilitent l'implémentation des techniques et approches que nous proposons.

1.1.2.1 MultiFlex et FlexMP [19-21]

STMicroelectronics offre un environnement pour la recherche et le développement de système sur puce. Il procure une infrastructure de simulation matérielle, des outils SoCs et un environnement de développement pour des applications MPSoCs [1].

La conception initiale de cet environnement répond à quatre objectifs principaux. Il doit d'abord être utile pour les produits embarqués existants de STMicroelectronics et le développement des technologies de conception de systèmes, tout en étant utile pour les plateformes multiprocesseurs de haut niveau encore en développement. Il doit aussi être un véhicule à l'usage des scientifiques pour la recherche à long terme sur les architectures

multiprocesseurs au niveau de l'exploration, de la conception d'outils ainsi que des méthodes de développement. De plus, il doit être un environnement ouvert, facilement accessible et composée du plus grand nombre possible d'éléments du domaine public. Enfin, il doit être pour les concepteurs un environnement de référence permettant de concevoir facilement des architectures de processeurs réseau réalistes [1].

Pour satisfaire à ces objectifs, l'environnement dispose de plusieurs caractéristiques dont les principales sont (1) l'extensibilité du traitement à multiprocesseurs, (2) l'exploration des configurations des processeurs (3) la capacité matérielle de multiples fils d'exécution pour les modèles de processeurs embarqués (4) un support pour un grand nombre de différentes topologies d'interconnexions entre les composantes matérielles, comme les processeurs, les coprocesseurs, la mémoire, etc., (5) le développement rapide d'applications et (6) une infrastructure de travail extensible pour le développement des logiciels embarqués et pour le contrôle, le débogage et l'analyse du modèle [1].

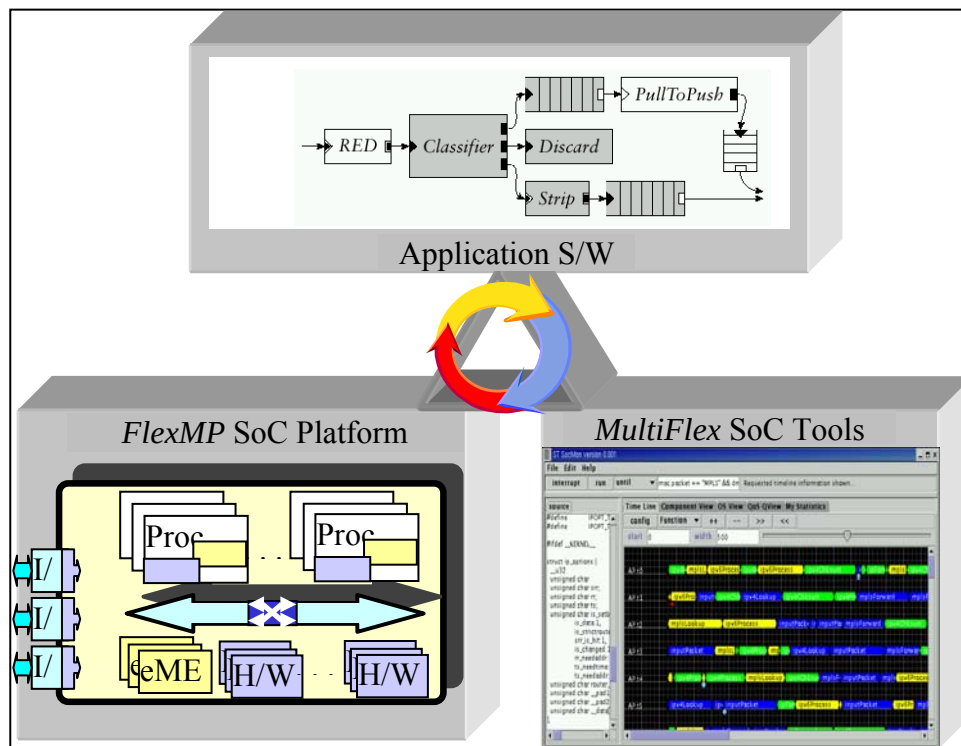


Figure 1.1 L'environnement de STMicroelectronics [1]

L'environnement de STMicroelectronics (Voir Figure 1.1) comprend trois composantes principales : la plateforme logicielle, la plateforme architecturale de processeurs et la plateforme des outils système-sur-puce [1]. L'environnement dispose de plusieurs outils visant à aider à la conception de MPSoCs.

Dans le cadre de ce projet, nous utilisons principalement deux outils : (1) l'outil Flexible MPSoC (FlexMP) pour l'évaluation des techniques d'optimisation de mémoire dans une architecture MPSoCs à modèle de programmation de type SMP (2) et l'outil MultiFlex pour l'évaluation des approches dans une architecture MPSoCs à modèle de programmation de type streaming.

1.1.2.2 Suite Chunky [22, 23]

La suite Chunky compte cinq outils: (1) Clan, (2) Candl, (3) PIP/PipLib, (4) Chunky et (5) CLooG. Clan (Chunky Loop Analyzer) est un outil pour extraire la représentation Polyhedral des parties avec contrôle statique d'application de haut niveau écrit en C, C++, C # ou Java. Candl (Chunky analyzer for dependancies in Loops) est un outil d'analyse de dépendance des données des parties avec contrôle statique. PIP/PipLib, (Parametric Integer Programming) est un outil de librairie qui trouve le minimum (ou maximum) lexicographique dans l'ensemble des points d'un Polyhedron. Chunky est un outil d'optimisation du code source. Il opère des transformations sur le code d'origine pour améliorer la localité des données. Les outils LooPo et LetSeE ont cette même capacité. CLooG (Chunky Loop Generator) est un outil qui génère le code pour l'exploration de Polyhedral. La plupart de ces outils sont encore en cours d'élaboration.

La complémentarité de la suite Chunky avec notre projet consiste à la génération de code pour l'exploration de Polyhedral des applications avec lesquels nous travaillons.

L'outil CLoog facilite l'analyse des applications de nouvelle génération pour l'adaptation des techniques d'optimisation.

1.1.2.3 Daedalus [24, 25]

Daedalus offre un environnement pour l'intervention rapide au niveau du système d'exploration architecturale de haut niveau de synthèse, de programmation et de prototypage d'application multimédia pour des architectures MPSoC. Daedalus repose sur l'hypothèse que les MPSoCs sont construits à partir de propriétés intellectuelles prédéterminées et prévérifiées. Ces propriétés intellectuelles comprennent une variété de processeurs programmables et dédiés, de mémoires et d'interconnexions, permettant ainsi la conception d'une grande diversité de plates-formes MPSoCs. À partir d'une application séquentielle en C ou C++, l'environnement Daedalus permet de paralléliser automatiquement l'application séquentielle en une application parallèle. Cette application parallèle est représentée par un Kahn Process Network (KPN[24, 25]). Les applications prises en charge par l'environnement se limitent à une structure ayant des boucles imbriquées avec des accès statiques affines. Ce type de structure est assez répandu dans le champ des applications multimédia et scientifiques. Par le biais de transformations automatisées au niveau source, l'environnement Daedalus se révèle capable de produire différentes entrées-sorties de KPNs équivalentes, dans lesquelles seul varie le degré de parallélisme. Cette aptitude permet l'exploration spatiale de différents niveaux de parallélisme. La Figure 1.2 illustre le flot de conception de l'environnement Daedalus.

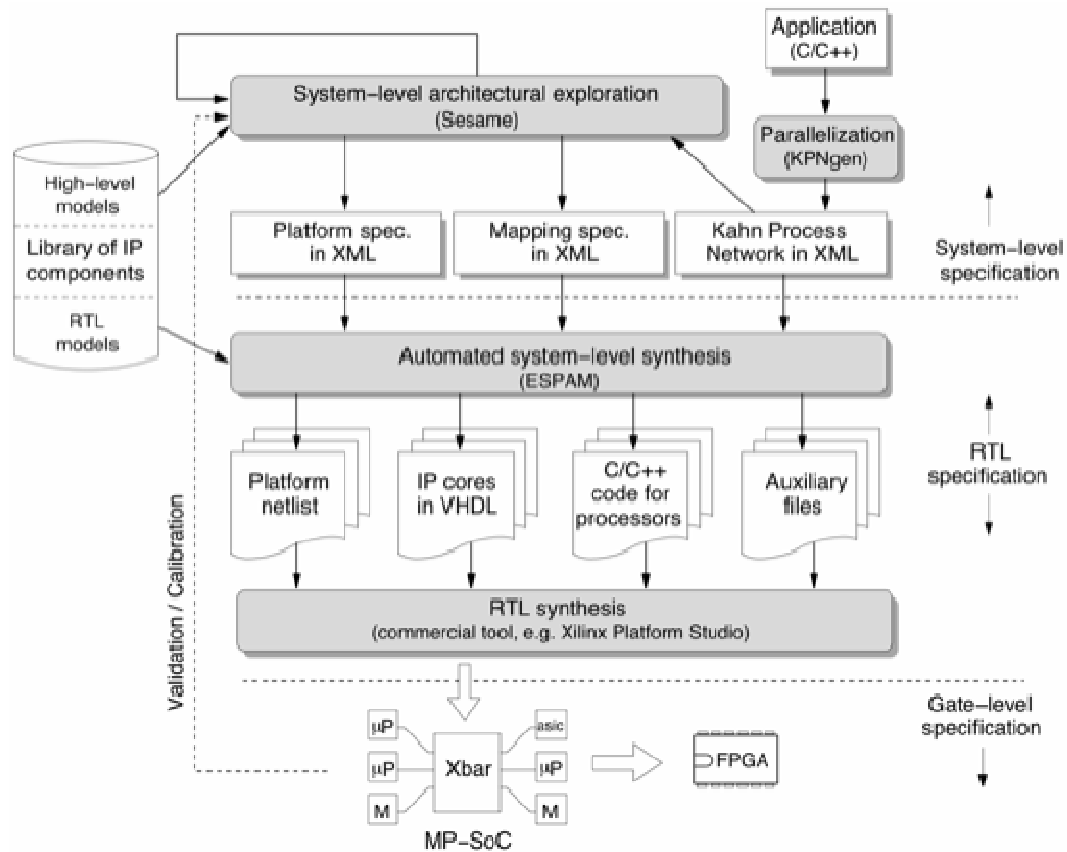


Figure 1.2 Daedalus[24, 25]

Les KPNs générés peuvent être ensuite utilisés pour la modélisation et la simulation au niveau système. L'environnement Daedalus permet d'évaluer rapidement la performance de différents mappages d'application à l'architecture et le partitionnement matériel et logiciel pour différentes plateformes architecturales cibles. Cette exploration de l'espace de conception devrait donner naissance à des spécifications système prometteuses (description du système au niveau plate-forme, description du placement de l'application, architecture et description de l'application) qui pourront être utilisées dans le flot de conception de Daedalus.

L'environnement Daedalus utilisant les spécifications et une version RTL des différentes composantes de la modélisation et de la simulation, génère automatiquement une version

VHDL synthétisable des différentes spécifications système pour la plate-forme MPSoC finale. L'environnement génère aussi le code C / C++ de l'application placé ensuite sur chaque unité de calcul. Avec l'utilisation d'outils commerciaux de compilation et de synthèse, la spécification de l'application générée par l'environnement Daedalus peut être facilement portée sur un FPGA pour le prototypage. Ces prototypes permettent de calibrer et de valider les modèles et les simulations de l'environnement Daedalus.

Finalement, l'environnement Daedalus permet d'aller d'une application séquentielle vers un prototype fonctionnant sur un FPGA tout en conservant tout le long des étapes de conception une grande souplesse d'expérimentation et d'exploration des différentes architectures MPSoCs.

La complémentarité de Daedalus avec notre projet consiste à la génération de Kahn Process Network des applications avec lesquels nous travaillons. L'outil KPNGen permet de générer cette représentation qui sert aux approches d'intégration des techniques d'optimisation mémoire avec l'étape de placement.

1.1.2.4 Compaan [26, 27]

Compaan est un outil qui permet de compiler automatiquement un sous-ensemble d'une application impérative en une représentation concurrente. Il utilise le langage Matlab comme langage impératif et compile des programmes Matlab dans une représentation concurrente. La représentation concurrente est une version particulière de la représentation Kahn Process Network.

Le développement de l'outil Compaan s'appuie sur les architectures MPSoCs. Vu l'ensemble des applications et la combinaison du matériel/logiciel à exécuter sur ce type d'architecture, Compaan est un outil pratique qu'utilise les concepteurs pour résoudre le difficile problème du partitionnement matériel et logiciel. Compaan regroupe trois outils différents schématisés dans la Figure 1.3: (1) MatParser, (2) DgParser et (3) Panda.

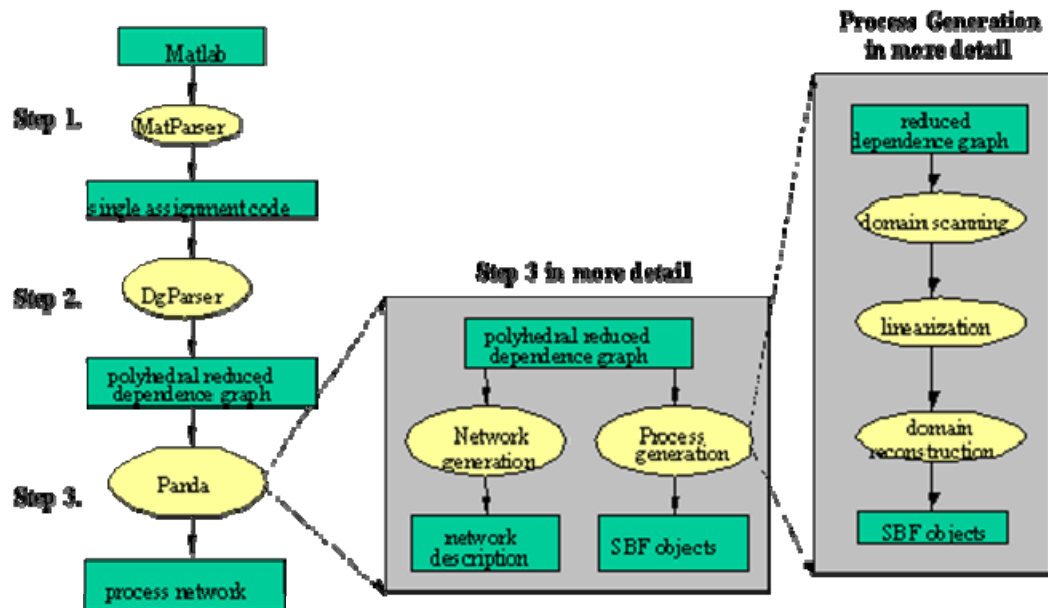


Figure 1.3 Compaan[26, 27]

MatParser est un compilateur de code séquentiel à code parallèle. Il extrait, par la technique Parametric Integer Programming (PIP), la totalité du parallélisme d'une application Matlab. L'application Matlab doit toutefois se limiter à des applications ayant des boucles imbriquées.

DgParser convertit le code généré par MatParser, en une représentation en graphe Polyhedral avec réduction de dépendance (PRDG). La représentation PRDG convient plus particulièrement à la manipulation mathématique.

Panda transforme la description PRDG d'un algorithme en un Kahn Process Network. Cet outil utilise la bibliothèque Polylib. La conversion de la représentation PRDG en KPN s'effectue en trois étapes: le balayage du domaine, sa reconstruction et sa linéarisation

Le KPN, créé par Compaan de l'UC Berkeley, est accessible dans une forme telle qu'il peut être simulé et analysé par les outils du projet Ptolemy II. Compaan génère la

description en langage MoML. C'est un langage de modélisation basé sur XML utilisé dans le projet Ptolemy II pour la spécification des interconnexions de composants configurables.

L'environnement Compaan se compare à l'environnement Daedalus; ils présentent tous deux une vue plus complète du flot de conception de MPSoC. Comme discuté brièvement auparavant, le type d'application pris en charge par ces environnements reste limité. L'environnement Compaan, à la différence de l'environnement Daedalus, supporte le langage Matlab comme entrée. On recourt souvent au langage Matlab pour une approche de base dans la conception de MPSoC.

La complémentarité de Compaan avec notre projet est la même que celle de Daedalus, par contre Compaan est utilisé lorsque l'application est écrite avec le langage Matlab.

1.2 L'optimisation mémoire

Cette section rapportent les articles publiés dont le sujet s'apparente à celui de notre projet et décrivent ce qui fait l'originalité de nos propres travaux.

1.2.1 Optimisation mémoire dans une architecture monoprocesseur

L'architecture traditionnelle monoprocesseur a été l'objet de nombreuses recherches visant à améliorer les données locales de l'antémémoire par diverses techniques et stratégies, telles que le remplacement par scalaire [12, 28], l'optimisation de l'ordre de stockage dans les tableaux [29, 30], la prélecture [30] et l'optimisation de la localité des données pour des codes utilisant principalement des tableaux comme structure de donnée [13, 31]. Un des groupes d'IMEC [32] fut le pionnier dans la transformation de code pour réduire la consommation d'énergie dans les applications embarquées traitant un grand nombre de données. Les techniques de transformation de boucle, comme la fusion de boucle et de l'allocation de tampon, ont été largement étudiées [33, 34]. Fraboulet et al. [35] et Marchal et al. [36] ont réduit l'espace mémoire dans les boucles imbriquées à l'aide de la

fusion de boucle. Kandemir et al. [37] ont étudié l'optimisation des interboucles en modifiant les modes d'accès de la boucle imbriquée afin d'améliorer la localité des données temporelles. Kadayif et al. [38, 39] introduisent le DST pour du pavage orienté espace de données et une optimisation des interboucles en divisant l'espace de données en plusieurs tuiles. Le travail de notre thèse se différencie des travaux précédents, car il cible une architecture avancée multiprocesseur sur puce (MPSoC). Les transformations que nous mentionnons ici peuvent être utilisées sur des systèmes distribués traditionnels comme les superordinateurs et les grappes. Elles ont, en outre, plus d'impact sur une architecture MPSoC. Les MPSoCs ont un environnement plus sensible, car leur superficie et leur consommation d'énergie sont limitées.

Comme énoncés par Li et al. [40], la plupart des efforts dans le domaine MPSoCs se concentrent sur des problématiques liées à la conception d'architecture et de circuits [30, 41]. Dans notre domaine, les techniques de compilation ne ciblent que les boucles imbriquées séparément (c'est-à-dire chaque boucle imbriquée indépendamment). Récemment, Li et al. [40] proposaient une méthode d'approche globale du problème. Cette méthode est limitée quand la taille des blocs de données partitionnés dépasse celle de l'antémémoire. Pour contourner ce problème, nous présentons une nouvelle méthode qui est une application de la fusion de boucle à toutes les boucles imbriquées et un partitionnement des données sur différentes unités de calcul. Van Achteren et al. [42] et Ilya et al. [43] décrivent une méthodologie pour l'exploration de la réutilisation des données. Des détails avec du formalisme et des fonctions de coût de compromis sont présentés. Ilya et al. [43] présentent une exploration et une analyse avec des mémoires bloc-notes (SPM) au lieu d'antémémoire. Le recours au SPM nécessite l'utilisation d'instructions spécialisées qui sont dépendantes de l'architecture. Il est possible aussi d'utiliser les améliorations architecturales pour augmenter les performances.

Catthour et al. [44] donnent un aperçu des connaissances sur les systèmes d'accès aux données et de gestion de stockage. Ces auteurs présentent différentes méthodologies sur les

transformations de boucle et sur le contrôle des flux de données, de la décision de réutilisation des données, de l'organisation des données dans la mémoire, de l'optimisation de type de données et de l'optimisation adressage. Ils fournissent une vue d'ensemble complète du flot de conception des SoCs sans donner d'informations détaillées sur les architectures multiprocesseurs et sur l'impact possible de ce flot sur les méthodologies. Certaines simulations sont effectuées sur une architecture multiprocesseur à modèle de programmation de type SMP; le temps d'exécution n'est pas indiqué.

Les travaux que nous avons cités portent essentiellement sur l'utilisation des techniques d'optimisation dans un environnement monoprocesseur. En comparaison, notre travail de thèse s'intéresse plus spécialement à l'effet qu'une architecture multiprocesseur et/ou avec traitement multifil peut avoir sur les techniques d'optimisation. Quelques techniques d'optimisation nécessitent certaines adaptations pour s'appliquer à une architecture multiprocesseur et/ou traitement multifil.

1.2.2 Optimisation mémoire dans une architecture multiprocesseur

Forsell et al. [45] qui étudient spécifiquement les architectures MPSoC, proposent une nouvelle classe d'antémémoire, nommée step-cache. Cette antémémoire peut servir à mettre en œuvre l'implémentation d'accès mémoire concurrent dans une architecture MPSoC avec mémoire partagée tout en évitant les problèmes de cohérence d'antémémoire. L'accent est mis sur la concurrence dans l'antémémoire avec l'aide d'un nouveau type d'antémémoire et non, comme nous le faisons [46-49], sur l'optimisation d'antémémoire et du temps d'exécution sous l'effet du traitement multifil sans modifier l'architecture existante.

Lors de la conception de système performant, certains opérateurs, tel que l'opérateur modulo, doivent être évités. Ces opérateurs prennent du temps à s'exécuter et consomment donc beaucoup d'énergie. Des développeurs proposent des composantes matérielles dédiées pour l'exécution efficace de ces opérateurs. Malheureusement, leur emploi demande une

surface de conception plus grande or cette surface est souvent limitée dans les architectures MPSoCs. La technique logicielle décrite par Ghez et al. [50] pour éliminer l'opérateur modulo utilise des instructions de condition qui introduisent des surcoûts supplémentaires sur certaines architectures SoC. Notre propre technique pour supprimer l'opérateur modulo diffère de celle-ci par le fait qu'elle utilise la transformation unimodulaire pour éviter ces surcoûts. Toutes les transformations que nous proposons contribuent à améliorer de façon significative les performances sans nécessiter des modifications architecturales.

Les travaux que nous avons cités portent essentiellement sur le gain de performance en ayant recours à des modifications architecturales. En comparaison, notre travail de thèse ne nécessite aucune modification architecturale. Il se concentre sur l'effet d'architecture multiprocesseur et/ou avec traitement multifil sur des techniques d'optimisation mémoire. Ces techniques d'optimisation mémoire requièrent parfois des adaptations pour être appliquées dans ce type d'architectures. Ces adaptations sont inspirées par l'étude et la comparaison de différentes techniques de parallélisation. Un examen attentif de l'effet de ces architectures permet aux concepteurs de faire de meilleurs choix au niveau des optimisations de code.

1.2.3 Évaluation des performances des techniques d'optimisation mémoire

Le traitement multifil a été très populaire ces dernières années, car c'est un moyen simple et rentable d'augmenter le parallélisme d'un système ou d'une application. Nous avons retenu, en ce qui nous concerne, l'environnement StepNP. Les unités de calcul de cet environnement supportent le traitement multifil [20]. Chacune de ces unités de calcul possède des banques de registres séparées pour chaque fils d'exécution dans le but d'une économie des surcoûts lors de la commutation de fils. Les recherches sur l'architecture avec un ou, ce qui est exceptionnel pour les MPSoCs, plusieurs processeurs ont permis de concevoir diverses techniques et stratégies pour maximiser la capacité du traitement

multifil. La plupart de ces travaux se focalisent sur l'architecture et analysent très peu l'effet du traitement multifil sur les techniques de compilation pour l'optimisation mémoire. L'espace et la taille de la mémoire sont des éléments cruciaux dans le MPSoCs, ce qui n'est pas le cas dans toutes les architectures multiprocesseurs.

L'architecture multiprocesseur embarquée et le modèle de programmation à base de multifil proposés par Schaumont et al. [51] permettent de réduire considérablement l'énergie pour un quad-processeur par rapport à un single-processeur. L'effet du traitement multifil sur quatre différents modèles de mémoire partagée a été étudié par Chong et al. [52]. Ces chercheurs mesurent les performances de ces différents modèles de mémoire. Notre travail de thèse vise le même but que les travaux déjà existants mais s'en distingue par le fait que l'amélioration des performances s'effectue sans avoir à modifier l'architecture.

D'autres recherches s'intéressent au temps d'exécution et à la mémoire; elles concernent de grands systèmes comme les grappes. La performance de l'antémémoire est étudiée à l'aide d'un processeur virtuel avec traitement multifil (MVP) [14]. La performance se trouve améliorée par la tolérance de latence de mémoire, mais aussi par la diminution du taux d'échec dans l'antémémoire en raison de l'exploitation de la localité des données. Le travail de Haines et al. [14] évalue les coûts et les avantages du traitement multifil logiciel sur un multiprocesseur à mémoire distribuée. La méthodologie de partitionnement matériel/logiciel proposée par Dimitroulakos et al. [53] vise, quant à elle, à améliorer les performances d'application embarquée dans les SoCs [53].

Les auteurs de [54] ont étudié une méthodologie d'ordonnancement pour la programmation de système sur puce avec traitement multifil. Leurs efforts portent principalement sur la réduction de la consommation d'énergie et sur le moyen de faciliter la gestion de la mémoire. Des travaux similaires ont été publiés plus récemment par IMEC [36]. Toutefois, ces travaux mettent l'emphasis sur les MPSoCs. Le traitement multifil

devient un facteur essentiel dans les MPSoCs et l'industrie estime, aujourd'hui, qu'il est important d'observer l'impact du traitement multifil sur l'optimisation mémoire.

Les travaux que nous avons cités portent sur l'exploitation du traitement multifil sur divers systèmes. En comparaison, notre travail de thèse s'intéresse uniquement aux architectures MPSoCs. Leurs performances dépendent essentiellement de la mémoire. Pour augmenter les performances, il nous faut exploiter au maximum les techniques de compilation offrant le meilleur rendement et, ce faisant, étudier l'effet qu'une architecture multiprocesseur et/ou avec traitement multifil ont sur elles.

1.3 L'optimisation mémoire dans le flot de conception des MPSoCs

L'application d'optimisation mémoire dans un modèle de programmation de type streaming, comme mentionné précédemment, est fortement influencée par l'étape de placement dans le flot de conception. Il est temps maintenant d'examiner les stratégies de placement avec les différentes techniques d'optimisation mémoire pour les MPSoCs. Nous montrons dans cette thèse que l'on peut, en adaptant certaines techniques d'optimisation mémoire, améliorer une architecture MPSoC avec un modèle de programmation de type SMP. À noter que nous proposons une approche originale pour combiner les techniques d'optimisation mémoire avec le placement dans une architecture MPSoC pour un modèle de programmation de type streaming. Dans un modèle de programmation de type SMP, les mêmes données sont traitées par différentes unités de calcul. Lors de l'étape du placement, la même tâche est répartie sur différentes unités de calcul dépendant du partitionnement des données. Les techniques d'optimisation mémoire doivent dès lors prendre en compte le parallélisme. Les techniques d'optimisation doivent donc être adaptées au parallélisme. Elles n'influencent en rien le placement car elles s'appliquent sur l'ensemble des unités de calcul. Par contre, dans un modèle de programmation de type streaming, chaque tâche est répartie sur une unité de calcul différente. Les techniques d'optimisation s'appliquent indépendamment à chaque unité de calcul comme elles le font dans une architecture

monoprocasseur. L'algorithme de placement doit, en conséquence, incorporer les techniques d'optimisation potentiellement appliquées sur les tâches pour maximiser les optimisations sur chaque unité de calcul.

À l'origine, les algorithmes de placement se concentraient uniquement sur l'optimisation des performances en termes de puissance de calcul. Aujourd'hui, on ne peut ignorer certaines contraintes liées à l'arrivée de nouvelles architectures telles que les MPSoCs. Ces architectures sont dévoreuses d'énergie et de mémoire d'où la nécessité d'optimiser au maximum les algorithmes de placement [55]. Le mode de placement proposé par Guangyu et al. [56] tend à placer les données dans une unité de calcul proche d'une autre unité de calcul ayant accès à ces mêmes données. Celui de Pastricha et al. [57] s'applique à une architecture de communication ayant le moins de bus possible et un coût réduit d'espace mémoire. Notre travail de thèse diffère par le fait que nous privilégions les techniques d'optimisation mémoire et que notre algorithme favorise le placement de tâches pouvant potentiellement se fusionner sur la même unité de calcul. Ceci permet de réduire la quantité de mémoire requise par chaque tâche. Notre approche réduit donc le coût de l'espace mémoire et celui de communication entre les unités de calcul.

Diverses autres approches du problème de placement ont été décrites [58-60]. Pour faciliter l'étape de placement, de nouveaux langages ont été élaborés. Le langage Brook est une extension du standard ANSI C; il est conçu pour intégrer l'idée du calcul parallèle et du calcul intense dans ce langage [61]. Le langage StreamIt est un langage qui facilite la programmation de grandes applications de type streaming et la mise en place de ces applications sur différentes architectures [62]. Ces différentes approches et langages facilitent le placement en exploitant le parallélisme d'une application, mais ne tiennent pas compte de l'optimisation potentielle des mémoires présentes dans l'application.

L'algorithme d'ordonnancement de Hu et al. [55] répond à une nécessité d'économie d'énergie pour un réseau sur puce (NoC [63]). Cet algorithme distribue automatiquement les tâches sur différentes unités de calcul et ordonnance leur exécution sous des contraintes

temps-réel. Hu et al. montrent comment se différencie une topologie de réseau de plusieurs ordinateurs d'une topologie de réseau de plusieurs unités de calcul comme dans un NoC. L'approche de Ruggiero et al. [64] tient compte du coût de communication pour la répartition et l'ordonnancement de graphes de tâches pour la conception de MPSoCs. Elle combine des algorithmes de programmation linéaire en entier (ILP) et de programmation par contrainte (CP), tout en indiquant que l'ILP convient uniquement aux petits problèmes de placement. Pour remédier à cette limitation, elle recourt aux systèmes avec ordonnancement statique. Le problème de placement est décomposé en deux sous-problèmes: (1) l'attribution de chaque tâche à une unité de calcul et (2) l'ordonnancement temporel des tâches sur chaque unité de calcul. Ruggiero introduit également le concept d'application de type streaming. Chez Markus et al. [65] et Tang et al. [66], c'est une approche heuristique qui est proposée et les outils présentés pour régler le problème de placement et d'ordonnancement font appel à des algorithmes génétiques. Les approches heuristiques ne garantissent pas une solution optimale, mais elles donnent des résultats acceptables dans des délais raisonnables. Thiele L. et al. [67] présentent un environnement appelé Distributed Operation Layer (DOL). Cet environnement permet l'analyse des performances au niveau système et un placement à l'aide d'un algorithme multi objectif. L'algorithme de placement, basé sur les algorithmes d'évolutions, optimise les temps de calcul et de communication.

Les travaux cités portent essentiellement sur l'utilisation d'algorithmes de placement pour l'optimisation d'ordonnancement ou de la consommation d'énergie et s'intéressent peu à l'optimisation mémoire. En comparaison, notre travail de thèse traite surtout de l'optimisation de l'espace mémoire sans ignorer les récentes études sur les algorithmes de placement pour les objectifs standard (coûts de charge et de la communication) et nous proposons une approche différente pour l'optimisation mémoire.

CHAPITRE 2. CONCEPTS DE BASE ET VUE GLOBALE DU PROJET

Ce chapitre présente une vue d'ensemble de notre projet et décrit brièvement les quelques concepts de base sur lesquels il s'appuie.

2.1 Concepts de base

La section « Concepts de base » porte sur les sujets suivants: les systèmes multiprocesseur sur puce, la conception niveau système, les modèles de programmation, l'architecture mémoire et les optimisations MPSoCs.

2.1.1 Système multiprocesseur sur puce (MPSoC)

Un système sur puce (SoC) implémente la plupart des fonctions d'une carte électronique. Un SoC inclut des mémoires, un processeur, des bus et des circuits logiques spécialisés. Les systèmes multiprocesseurs sur puce (MPSoC) sont semblables aux SoCs, mais comptent plusieurs processeurs souvent spécialisés. Les MPSoCs sont généralement hétérogènes et comportent des protocoles de bus plus compliqués; et ils doivent se plier à des contraintes de temps réel, d'espace et d'énergie [5].

Les travaux de recherche sur les MPSoCs ont pour objectif d'en multiplier les fonctionnalités, d'en augmenter la fiabilité et d'accélérer la bande passante tout en réduisant la consommation de puissance et le coût. Cet objectif suppose une très forte intégration des MPSoCs sur leur support en silicium [5].

Bien que partageant certaines caractéristiques avec les super ordinateurs, les MPSoCs ont des propriétés spécifiques qui les en différencient; ils ont des ressources beaucoup plus limitées et pour minimiser les coûts de mise en œuvre, ils doivent être conçus avec très peu de surcoûts d'espace.

Contrairement à la plupart des supers ordinateurs dont le seul objectif est d'atteindre le plus haut rendement possible en termes de traitement de calcul, la plupart des MPSoCs sont limités par une contrainte de consommation d'énergie maximale. Comme le marché des produits mobiles continue de se développer et que le coût de refroidissement de puce ne cesse d'augmenter, les MPSoCs à faible consommation d'énergie sont de plus en plus intéressants. Pour certains produits (ex. appareils portables multimédias, téléphones mobiles), la consommation d'énergie est le principal objectif d'optimisation, tandis que la performance est généralement perçue comme une contrainte de conception.

Contrairement aux super ordinateurs qui permettent d'exécuter une grande variété d'applications, les MPSoCs ciblent une application spécifique ou un type d'applications. En règle, le concepteur connaît bien les propriétés de l'application et il peut dès lors personnaliser (ou particulariser) la conception du MPSoC [55].

Traitement multifil dans les architectures MPSoCs

Les MPSoCs utilisent souvent un grand nombre d'unités de calcul pour tirer profit du haut niveau de parallélisme. Pour accroître encore davantage le niveau de parallélisme et améliorer la performance des processeurs sans augmenter la fréquence de l'horloge, on peut utiliser le traitement multifil matériel. Cette façon de faire améliore l'efficacité du pipelining. Deux applications indépendantes peuvent s'exécuter simultanément sur le même pipeline. Puisque les applications sont indépendantes, elles acceptent la parallélisation de leurs instructions, car il n'y a pas de correspondance entre elles. Si une des applications s'arrête, faute d'informations dans l'antémémoire par exemple, l'autre application continue dans le pipeline, car l'absence d'informations dans l'antémémoire n'influence pas son exécution. La technique de traitement multifil matériel est considérée comme une vraie alternative pour améliorer le pipelining [5].

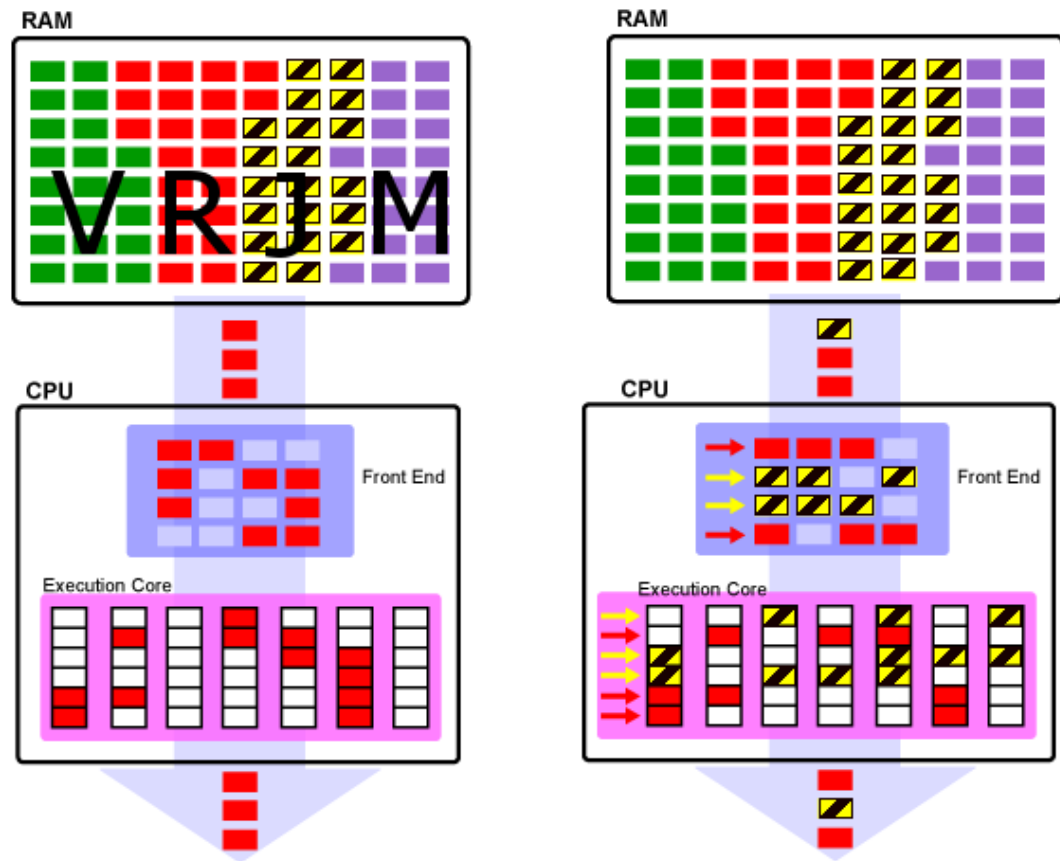


Figure 2.1 Traitement classique (gauche) et traitement multifil (droite)[68].

Dans la partie supérieure gauche de la Figure 2.1, les boîtes colorées représentent les instructions pour quatre programmes courants différents. On peut voir que seules les instructions pour le programme R sont en cours d'exécution, alors que les autres attendent que le processeur puisse les exécuter. Les boîtes blanches représentent des états vides du pipeline. Le processeur n'arrive pas à ordonnancer le code à exécuter[68].

Comme indiqué précédemment, le traitement multifil résout le problème des boîtes blanches dans le mécanisme du pipelining. La partie droite de la Figure 2 compte moins

de boîtes blanches, car le processeur ordonnance deux applications avec deux fils d'exécution [68].

Les processeurs avec traitement multifil peuvent réduire les temps de latence entre la mémoire et le processeur. Considérons par exemple le cas d'un processeur avec traitement multifil exécutant deux fils d'exécution, les programmes R et J. Si le fil d'exécution rouge requiert du processeur des données absentes dans l'antémémoire, il va devoir attendre, et parfois longtemps, pour les obtenir. Entre temps, grâce au traitement multifil, le processeur exécute le fil d'exécution jaune (bar oblique noir) ce qui permet de maximiser le pipeling pendant des périodes d'attente [68].

2.1.2 Conception niveau système

Les MPoCs sont des systèmes complexes. Dans le domaine des systèmes sur puce, les progrès consistent à mettre au point de nouvelles méthodologies pour, en dépit de la complexité, augmenter la productivité et réduire les délais de mise en marché.

Une méthodologie très répandue est la conception au niveau système. Le concepteur élève le niveau d'abstraction de la plateforme. Le système s'organise comme un ensemble de blocs communicants. Le concepteur doit faire des projections exactes pour respecter les concepts et les aspects de bas niveau au moment de la conception du MPSoC. Cette abstraction lui permet de se concentrer avec plus de précision sur les enjeux ou problématiques de conception énoncés dans l'introduction de cette thèse.

Dans une conception au niveau système, le modèle du système se situe au cœur du processus de la saisie des spécifications de la conception jusqu'à l'exécution des tests [69, 70].

Le modèle est une spécification de l'exécutable qui est en continuel ajustement tout au long du processus de développement. La simulation montre si le modèle fonctionne correctement.

Une fois les spécifications matérielles et logicielles saisies, comportement de la synchronisation par exemple, on peut générer automatiquement du code pour les systèmes embarqués et créer des bancs d'essai pour la vérification de leur fonctionnement Ceci permet d'économiser du temps et d'éviter les erreurs possibles d'un codage manuel[69].

2.1.3 Modèle de programmation

Dans cette thèse, nous utilisons deux modèles de programmation. (1) Le premier modèle de programmation de type symmetric multi-processor (SMP), dans l'esprit de POSIX[71], se fonde sur des ressources de traitement symétriques ayant accès à une mémoire partagée. (2) Le second modèle de programmation de type streaming, dans l'esprit de StreamIt[62] et Brooks[61], se rapproche du modèle producteur et consommateur, qui englobe des applications dans des composants logiciels bien définis. Il opère selon une sémantique de communication de flux de données axées statique ou dynamique. Le contrôle en est généralement assez simple.

Ces deux modèles de programmation ont leurs avantages et désavantages (voir Table 1). Le modèle de programmation de type SMP est souvent utilisé pour les plateformes multimédia sur SoC. Dans ce modèle, le créateur de l'application doit identifier et exprimer explicitement le parallélisme qui est bien compris dans la plupart des cas. Le modèle de programmation de type streaming bénéficie d'un regain d'intérêt grâce aux nouvelles applications multimédia et le sans-fil. Ce modèle de programmation est basé sur les flux de données (streaming). Il était peu utilisé ou réservé à des applications très spécifiques. Il était en effet difficile de placer une application écrite avec ce modèle de programmation sur une architecture classique. Grâce à des progrès techniques récents, un développeur peut aujourd'hui programmer avec ce type de modèle de programmation pour ensuite le placer efficacement sur une architecture classique (i.e. autre que l'architecture MPSoC). La programmation par flux de données accentue l'expression du parallélisme en divisant le calcul et les accès-mémoires. Le parallélisme et la localité des données explicites dans un

programme de flux de données facilitent l'utilisation des techniques d'optimisation traditionnelles lors de la compilation. Le modèle de programmation basé sur les flux favorise un style programmation avec récupération, traitement et distribution des données [72]. L'industrie informatique étudie les possibilités de l'intégrer aux outils de conception. Les recherches à venir vont tenter d'explorer les techniques de placement de ce modèle de programmation sur des architectures plus spécialisées (MPSoCs par exemple).

Le Tableau 2.1 présente les avantages et désavantages des différents types de modèle de programmation.

Tableau 2.1 SMP vs Streaming

	Avantages	Désavantages
SMP	▪ Support des systèmes d'exploitation actuels. ▪ Support des codes patrimoniaux⁶. ▪ Équilibrage des charges.	▪ Maintien nécessaire de la cohérence des données locales et partagées. ▪ Nécessité d'une large bande passante pour les communications inter-processeurs. ▪ Extensibilité limitée. ▪ Absence de support pour les systèmes hétérogènes. ▪ Synchronisation du contrôle et des données.
Streaming	▪ Surcoût de communication faible. ▪ Réduction de la bande passante des canaux de communication. ▪ Communication et calcul orthogonal. ▪ Estimation facile des caractéristiques de communication nécessaires à l'application.	▪ Support faible pour application orienté contrôle.

2.1.4 Architecture Mémoire

Une hiérarchie de mémoire est nécessaire pour soutenir la vitesse d'accès à la mémoire des processeurs actuels. Cette caractéristique des architectures traditionnelles explique la

⁶ « Legacy codes » en anglais

mise en place de plusieurs niveaux de mémoire entre la mémoire principale et les registres des processeurs. Le choix hiérarchique des niveaux de mémoire consomme beaucoup d'énergie. Pour les architectures classiques, l'utilisation des mécanismes matériels tels que la prédiction de l'antémémoire et les ajouts de tampons contribue en grande partie aux pertes de performance globale. Les applications de type multimédia dont sont surtout dotés les dispositifs portables et mobiles peuvent tirer profit des mécanismes logiciels et d'une meilleure distribution de la mémoire. Les niveaux hiérarchiques de la mémoire exigent également des modes d'accès et de partitionnement des données différents [32].

Antémémoire (cache)

Les techniques d'optimisation présentées dans cette thèse s'appliquent principalement à l'antémémoire et à ses performances.

L'antémémoire est une mémoire très rapide mais de faible capacité qui retient les données le plus fréquemment utilisées pour en accélérer le traitement à chaque nouvelle utilisation. Le stockage dans l'antémémoire de ces données récurrentes réduit le nombre des transferts entre l'unité de traitement et la mémoire centrale. L'accès rapide à l'antémémoire améliore les performances. Si la rapidité d'accès est un avantage, leur faible capacité et leur prix sont des désavantages. Il est important d'avoir une bonne optimisation de la localité des données, car l'absence de données dans l'antémémoire est très coûteuse [5].

Dans une application spécialisée, il est souhaitable de laisser le plus longtemps les données actives dans l'antémémoire pour diminuer le temps d'exécution. L'accès aux données situées dans l'antémémoire peut prendre environ 1 cycle d'horloge. Celui des informations absentes de l'antémémoire peut prendre environ 10-20 cycles [6].

2.1.5 Optimisation MPSoC

Il est important que le processus d'optimisation se fasse lors de la conception du MPSoC. La plateforme étant limitée en ressources, toute optimisation apporte un gain de performance non négligeable.

Il existe différentes approches pour optimiser les applications MPSoCs. Elles se font soit au niveau de l'architecture, soit au niveau logiciel. Les prochaines sections en donneront une brève description.

2.1.5.1 Métriques d'optimisation

Parallélisme

L'optimisation du parallélisme est très importante, car c'est la raison principale de l'emploi des MPSoCs. La parallélisation détermine le mode d'utilisation de la mémoire par plusieurs unités de traitement. Un parallélisme excessif risque d'entraîner des problèmes majeurs de dépendance des données et de surcharger les interconnexions entre les différentes unités de traitement. Les coûts importants d'une surcharge sont un élément de contre performance. Le compilateur doit prendre en compte ces considérations et faire un compromis en sachant quelle sera la puissance de consommation du système entier. À noter que certains codes ne peuvent pas s'adapter à la parallélisation. Parfois, il faut réécrire le code d'une tout autre manière avant de le compiler. Le compilateur doit d'abord estimer les performances, la consommation de puissance et l'espace mémoire, puis optimiser le code en conséquence [5].

La localité des données et des instructions

L'optimisation des données est très importante puisqu'elle tend à limiter le nombre de transferts entre la mémoire principale et l'antémémoire. Les instructions étant en lecture seule, la localité des instructions n'est pas vraiment un problème. Il n'en est pas de même

des données; lors d'une création d'application, le programmeur porte peu d'attention à la localité des données et n'utilise pas l'antémémoire à son maximum potentiel. Lorsque plusieurs unités de traitement utilisent la même ligne de l'antémémoire, une mauvaise gestion de la mémoire entraînera des mises à jour et des validations de ligne de l'antémémoire inutiles ou superflues. Pour éviter, dans les systèmes à unités de traitement multiples, que plusieurs unités se situent sur une même ligne de l'antémémoire sans nécessairement partager les mêmes données, le compilateur utilise principalement des transformations de boucles. Des problèmes de dépendance entre boucles et données peuvent empêcher la transformation de certains codes. Dans certains cas, mais non pas dans tous, il suffit d'un simple ajustement du code pour que les transformations s'opèrent. L'optimisation de la disposition des données à l'intérieur de la mémoire constitue une autre solution au problème [5].

La consommation en puissance

L'optimisation des performances est importante, mais on doit faire attention à la consommation en puissance du système. Cette consommation est une des caractéristiques qui différencient les MPSoCs des systèmes parallèles traditionnels. Les MPSoCs ne consomment que très peu de puissance. Les compilateurs et les théories pour les systèmes parallèles traditionnels ne sont pas adaptés aux applications pour les MPSoC; ils ne tiennent pas toujours compte de la consommation en puissance. Un bon compilateur trouve un juste équilibre entre l'augmentation de la puissance de calcul et la diminution du temps de traitement. Pour optimiser la consommation en puissance, il est souvent plus avantageux de remplacer une unité de traitement de forte puissance par plusieurs unités de moindre puissance. On augmente ainsi la parallélisation tout en gardant un bas niveau de consommation en puissance [5].

L'espace mémoire

L'espace mémoire est aussi une des caractéristiques particulières des MPSoCs. L'optimisation mémoire diminue les interactions entre des composants ne se trouvant pas sur la puce. La localité des données rapproche les données importantes, tandis que l'optimisation d'espace mémoire rapproche l'ensemble des données. L'optimisation d'espace mémoire passe par le partage de données pour des fonctions et de la durée de vie de certaines données ou structures (tableau, liste...). Il est parfois possible d'éliminer des structures de données inutilisées. Le compilateur, mais aussi le programmeur doivent s'assurer que le code soit optimal au niveau de l'espace. Il faut faire attention à ne pas trop réduire la durée de vie de certaines données au risque de diminuer la localité des données [5].

2.1.5.2 Techniques d'optimisation mémoire

Nous consacrerons cette section aux techniques de compilation et, plus spécifiquement, à celles des transformations de programme. Il existe de nombreuses techniques, mais la fusion de boucles et l'allocation de tampons sont les deux seuls techniques de transformations de boucles que nous passerons en revue. Elles présentent les caractéristiques décrites dans la section précédente et s'appliquent aux optimisations qui seront traitées ultérieurement. Elles agissent sur toutes les profondeurs de la boucle imbriquée quand celle-ci dépend de toutes les boucles imbriquées précédentes. Cette situation se présente souvent dans des applications de type multimédia et sans-fil.

Tout au long de notre thèse, le **modèle polyhédral** [73, 74] est utilisé pour représenter les calculs de boucles imbriquées. Par exemple, la boucle imbriquée de profondeur 2 dans le code de la Figure 2.2 (a) peut être illustrée par un domaine en deux dimensions (Figure 2.2 (b)). L'axe i correspond à l'axe de la boucle i et l'axe j correspond à la boucle j . À chaque itération (i, j) , trois instructions S_1 , S_2 et S_3 sont calculées. L'ordre des calculs dépend de

l'ordre lexicographique : l'itération (i, j) sera calculé avant l'itération (i', j') , uniquement si $(i, j) \prec (i', j')$.

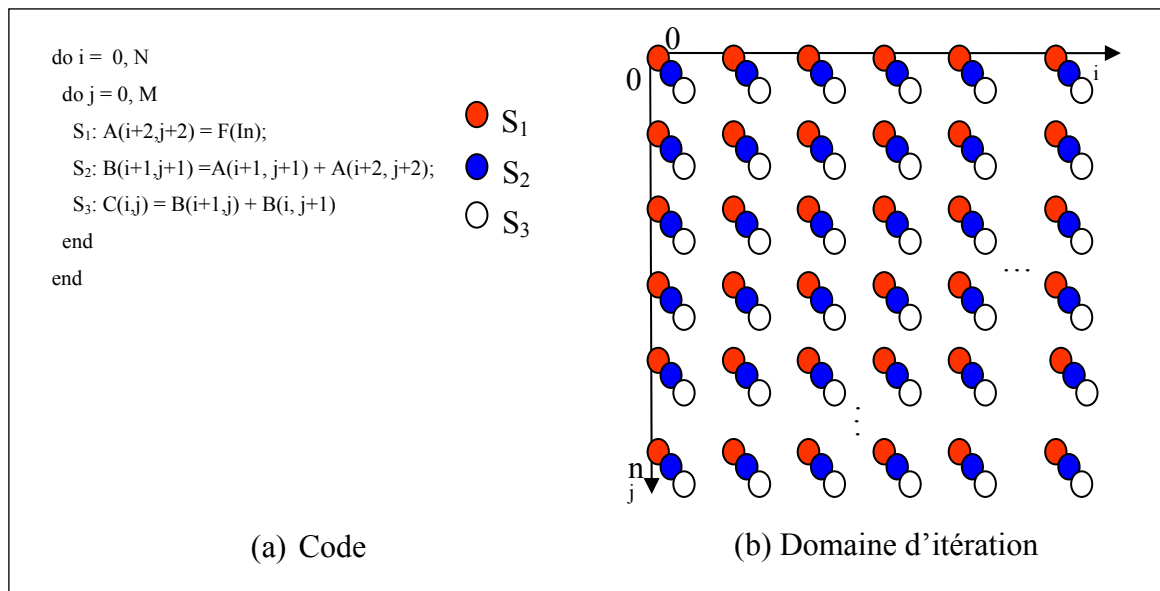


Figure 2.2 Exemple d'un code et de son domaine d'itération

Dans le code de la Figure 2.2, chaque instruction de la boucle imbriquée peut-être représentée par un vecteur d'itération vecteur $\vec{i}$ dont chaque élément représente une boucle. L'itération $\vec{j}$ de la boucle imbriquée L_k , est dite dépendante de l'itération $\vec{i}$ de la boucle imbriquée L_{k-1} , si $\vec{i}$ produit un élément du tableau A_{k-1} utilisé par $\vec{j}$. La différence entre ces deux vecteurs $(-)$ est appelée le vecteur de dépendance de données. Notre thèse se concentre surtout au cas où toutes les entrées du vecteur de dépendance de données sont des constantes. Dans ce cas, le vecteur est communément appelé vecteur de distance. Lors d'une représentation vectorielle, on utilise souvent les opérandes exprimant l'ordre lexicographique $(\prec, \preceq, \succ, \succeq)$ et les opérandes plus traditionnels tels que $(\leq, \geq, >, <)$.

L'**ordre lexicographique** est un ordre total, tandis que les opérandes traditionnels définissent un ordre partiel. Cette distinction a son importance, quand il s'agit d'ordonner les calculs; comme les éléments sont indexés par des vecteurs, on sait quel calcul précède un autre calcul [47].

La technique de **fusion de boucle** est souvent utilisée pour des applications ayant de nombreuses boucles imbriquées. Par cette technique, une seule boucle imbriquée en remplace plusieurs. Elle est d'usage courant pour l'optimisation puisqu'elle augmente la localité des données dans un programme. Elle permet l'emploi immédiat des données déjà en place dans l'antémémoire. Le Chapitre 3 (i.e. article 1) apportera des informations plus détaillées

La technique de l'**allocation de tampon** est souvent utilisée pour des applications avec des tableaux temporaires qui stockent des calculs intermédiaires. L'allocation de tampon réduit la taille des tableaux temporaires. Elle diminue l'espace mémoire et réduit le ratio de l'absence d'informations dans l'antémémoire. La taille du tampon est définie par les dépendances entre les instructions. Le tampon ne conserve que les éléments utiles (également appelés les éléments vivants). Un élément d'un tableau est considéré vivant au temps t , s'il est utilisé (écrit) au temps t_1 et une dernière fois (lu) au temps t_2 où les temps $t_1 < t < t_2$. Le Chapitre 3 (i.e. article 1) décrit plus en détail la technique de l'allocations de tampon.

2.2 Vue globale du projet

Cette section porte sur le flot de conception des MPSoCs qui est à la base des fondements de cette thèse. Elle comprend aussi une sous section décrivant le type d'applications touchée par nos travaux de recherche.

2.2.1 Flot de conception

La Figure 2.3 illustre le flot de conception adopté dans le cadre de ce projet. Notre thèse est centrée sur l'optimisation mémoire. Les différentes approches présentées exploitent les caractéristiques des techniques d'optimisation mémoire dans le flot de conception des systèmes MPSoCs.

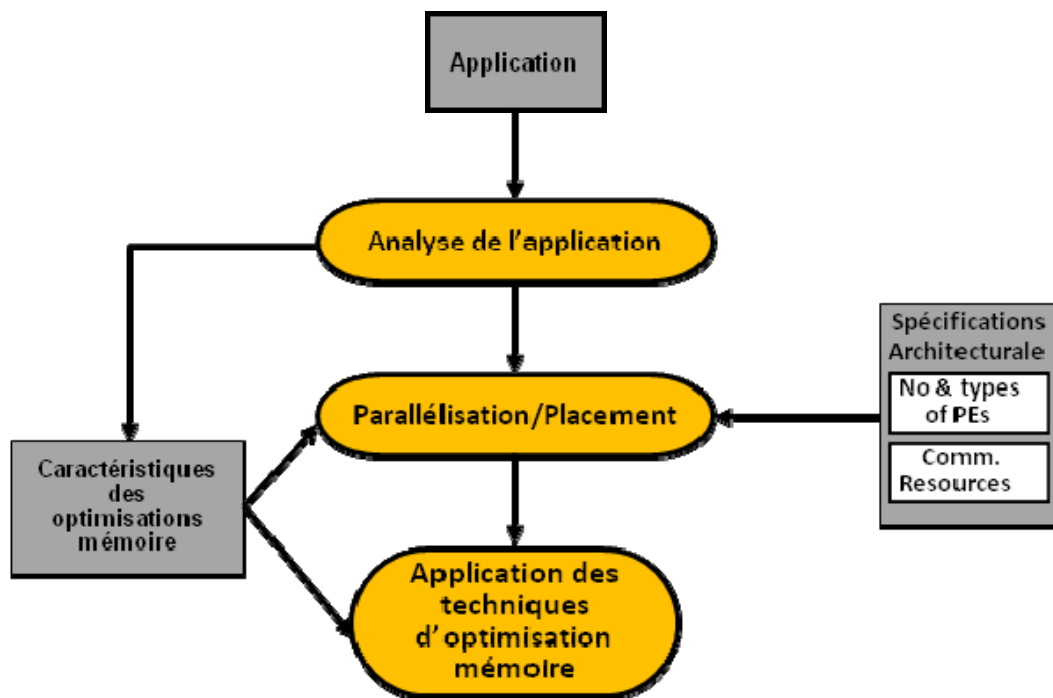


Figure 2.3 Flot de conception général

Nous privilégions deux types de modèle de programmation dont les flots de conception diffèrent d'un modèle à l'autre. Dans un environnement SMP, les optimisations mémoire doivent être adaptées, car elles s'appliquent à différentes unités de calcul. Dans un environnement streaming, les optimisations mémoire n'ont pas besoin d'être adaptées, car elles s'appliquent sur chaque unité de calcul séparément. Pour plus d'informations sur le flot spécifique à chaque modèle de programmation, on peut consulter le Chapitre 3 et le Chapitre 4. La suite traite du flot de manière générale.

Application

L'application est une des principales entrées du flot de conception. L'application peut être modélisée ou décrite en utilisant différents formalismes bien établis. Lors de la conception des MPSoCs à modèle de programmation de type SMP, nos travaux s'appliquent sur des applications décrites sous la forme de polyédral et de code source. Lors de la conception des MPSoCs à modèle de programmation de type streaming, nos travaux s'appliquent sur des applications décrites sous la forme de graphes de tâches.

Spécifications architecturales

Les spécifications architecturales sont une des entrées du flot de conception qui permet de capturer les détails architecturaux du MPSoC. On y retrouve entre autres les informations sur le type d'interconnexions, le nombre et le type de processeurs et les informations sur la mémoire. Cette capture permet de définir les limites imposées par l'architecture.

Analyse de l'application

À cette étape, l'analyse de l'application détecte quelles sont les techniques possibles d'optimisation de mémoire. L'analyse se fait sur la structure du code, les boucles imbriquées, les données et les dépendances de données.

Caractéristiques des optimisations mémoire

Après avoir identifié les techniques possibles d'optimisation de mémoire, il faut encore identifier les conditions nécessaires à la mise en œuvre de ces techniques. L'étape du placement et de la parallélisation peut empêcher l'utilisation de certaines techniques. Voir le Chapitre 3 et le Chapitre 4 pour de plus amples informations.

Placement et parallélisation de l'application

À cette étape s'effectuent le placement et la parallélisation de l'application. Sont prises en compte les informations collectées dans l'étape précédente et les informations sur la spécification de l'architecture. Les stratégies de placement et parallélisation diffèrent selon le modèle de programmation. Pour un modèle de programmation de type SMP, le placement est implicite et la parallélisation de l'application se fait au niveau des données. Pour un modèle de programmation de type streaming, le placement est explicite et la parallélisation de l'application se fait au niveau des tâches. Les caractéristiques des optimisations mémoire doivent être prises en considération au niveau de la parallélisation dans un modèle de programmation de type SMP et au niveau du placement pour un modèle de programmation de type streaming.

Optimisation mémoire

Les techniques d'optimisation mémoire s'appliquent différemment selon le modèle de programmation. Les concepts de base restent les mêmes, mais dans certains cas les techniques d'optimisation mémoire ont besoin d'adaptation ou d'étapes préliminaires. Notre thèse applique manuellement les techniques sur les applications. Dans un modèle de programmation de type SMP, l'automatisation des techniques d'optimisation est plus compliquée; l'étude de ces techniques dépasse le cadre de cette thèse. Dans un modèle de programmation de type streaming, l'automatisation des techniques d'optimisation de mémoire est plus concevable, puisqu'elles opèrent dans un contexte monoprocesseur sur chaque unité de calcul séparément. PIPS [16], SUIF [15] ou Chunky [22] sont trois des divers outils d'optimisation existants que l'on peut utiliser pour l'automatisation.

2.2.2 Types d'application

Dans le domaine des MPSoCs, l'application est une composante très importante de la conception et du produit final. L'application définit l'architecture. Cette section présente

deux types d'applications fortes répandues dans le domaine de MPSoC. Ce sont celles que nous utiliserons.

2.2.2.1 Hypothèse sur le type d'application

Les applications sur lesquelles nous opérons respectent les conditions suivantes :

- Absence de dépendance entre les boucles imbriquées.
- Dépendance entre deux boucles consécutives ou entre une boucle et toutes celles qui la précèdent.
- Même profondeur de toutes les boucles.
- Boucles ayant des limites constantes.
- Mêmes dimensions de tous les tableaux de données
- Accès uniforme aux tableaux.

Nos techniques peuvent agir sur des structures de codes d'applications plus générales, mais cela risque de compliquer l'automatisation et ajoute des surcoûts sans gain supplémentaire. Notre thèse cible des applications multimédia se conformant aux conditions énoncées. Et quand une application ne respecte pas l'une de ces conditions, on peut généralement la modifier pour la rendre conforme.

Les raisons pour lesquelles la majorité des applications multimédia partagent les caractéristiques requises sont les suivantes :

- Les applications multimédia sont souvent composées de plusieurs calculs pipelinés (chaque calcul correspondant à une boucle imbriquée). Chaque boucle imbriquée lit l'image produite par le calcul précédent, traite cette image, et produit une nouvelle image. C'est pour cela que les dépendances de données se font entre boucles consécutives.
- Il n'y a pas de dépendances dans une boucle imbriquée; chaque boucle imbriquée ni ne lit, ni n'écrit dans la même image.

- En général, chaque calcul (boucle imbriquée) prend en entrée les images produites dans la boucle imbriquée précédente et produit une image intermédiaire (la première boucle imbriquée prend en entrée l'image d'entrée et la dernière boucle imbriquée produit l'image de sortie). Puisque toutes les images ont les mêmes dimensions (n), les tableaux auront les mêmes dimensions (n). Une boucle a accès à chaque dimension du tableau ce qui explique que chaque boucle imbriquée ait n boucles (profondeur n).
- Habituellement, chaque boucle imbriquée travaille sur la totalité de l'image. C'est pourquoi, les limites des boucles sont des constantes.
- Les accès sont uniformes. Cette caractéristique est fixée dans la plupart des travaux sur les transformations de programme. La majorité des applications multimédia utilisent un accès uniforme qui se présente généralement sous la forme $A(a * i + b, c + d * j, \dots)$ où a, b, c et d sont des constantes (i, j sont les boucles) [47].

2.2.2.2 Applications Multimédia illustrées dans le cadre de la thèse

Avec l'apparition dans l'industrie d'applications embarquées de type multimédia, les caractéristiques de conformité deviennent essentielles. Ces applications manipulent une masse de données et emploient souvent des tableaux multidimensionnels pour stocker des résultats intermédiaires pendant le traitement des tâches multimédia.

Cette section présente les deux applications sélectionnées dans le cadre de la thèse. La première application relève du domaine médical et la deuxième du domaine de la photographie.

Détection d'images lacunaires⁷

La première application de type multimédia opère sur des images médicales (détection d'images lacunaires). Elle effectue cinq traitements correspondant chacun, d'une manière

⁷ « Cavity detection » en anglais

algorithmique, à une boucle imbriquée. Le premier traitement se fait sur l'image d'entrée. Chaque traitement, opère sur l'image fournie par le précédent traitement; l'image de sortie est le résultat final de ces traitements successifs. Les effets sur l'image sont, selon l'ordre d'opération de chacun des cinq traitements, (voir Figure 2.4) un flou vertical, un flou horizontal, un surlignement des côtés, une inversion des couleurs noires et blanches et une détection des contours.

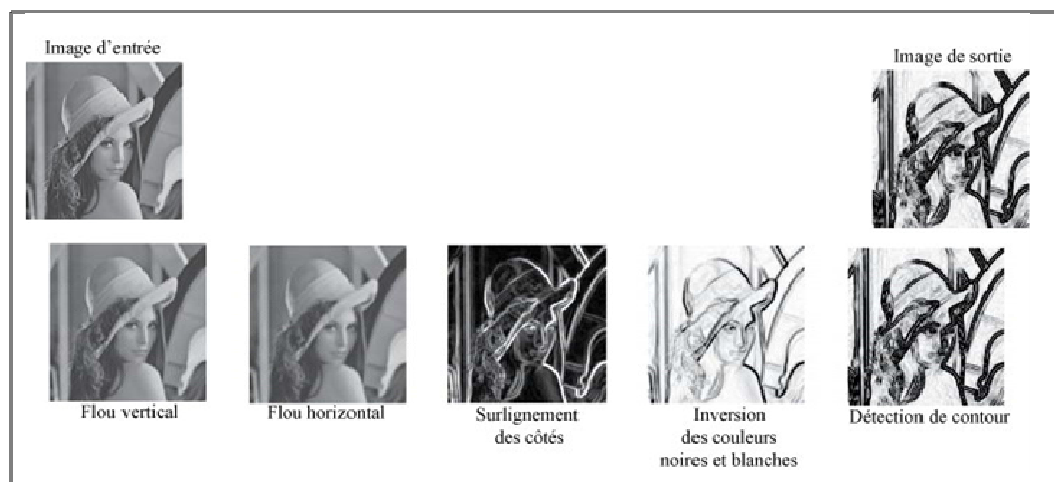


Figure 2.4 Détection d'images lacunaires

Dématriçage⁸

La deuxième application de type multimédia traite les images produites par les caméras numériques. Cette application est dénommée Dématriçage [75](Figure 2.5). Chaque pixel d'une image-couleur se compose de trois couleurs de base: le rouge, le vert et le bleu. L'interpolation des rangées de filtres couleur ou dématriçage constitue un maillon important de la chaîne du traitement des images numériques. Il s'effectue trois interpolations sur l'image d'entrée: (1) l'interpolation verte, (2) l'interpolation bleue, et (3) l'interpolation rouge. L'interpolation verte calcule les pixels verts et compte de manière

⁸ « Demosaicing » en anglais

algorithmique deux boucles imbriquées. L'interpolation rouge suit l'interpolation verte et compte cinq boucles imbriquées. L'interpolation bleue suit également l'interpolation verte et se fait en parallèle avec l'interpolation rouge; elle compte cinq boucles imbriquées.

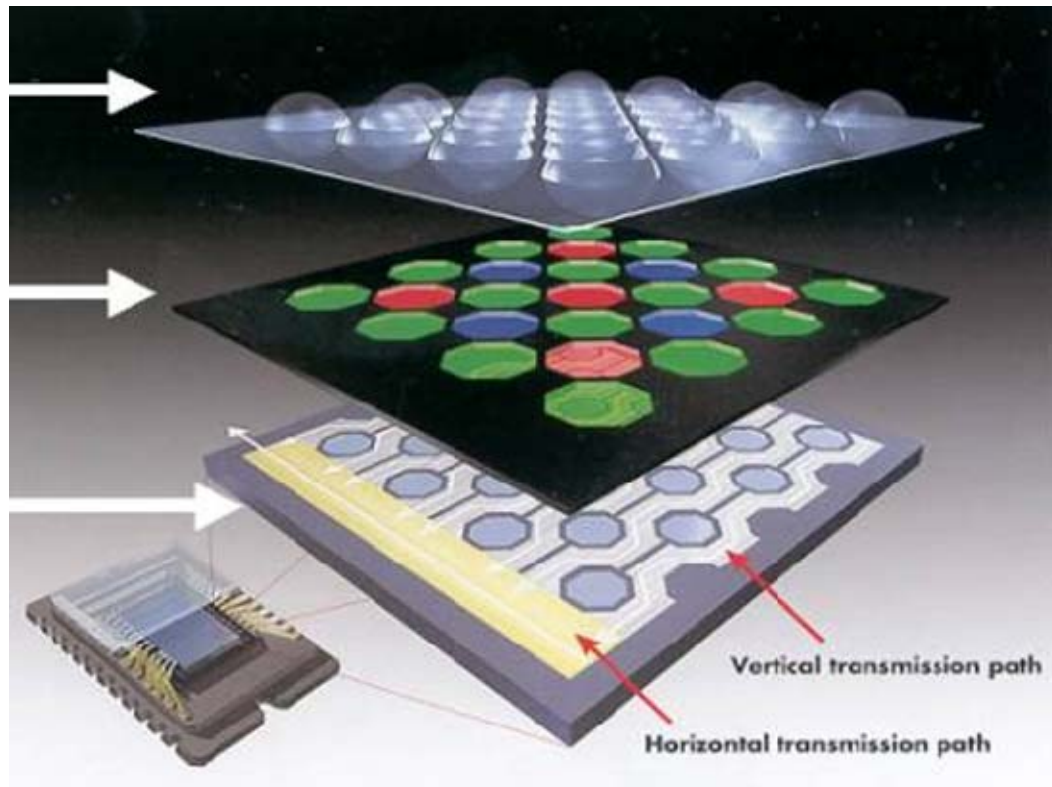


Figure 2.5 Dématriçage[76]

CHAPITRE 3. ARTICLE 1: MULTIPROCESSOR, MULTITHREADING AND MEMORY OPTIMIZATION FOR ON-CHIP MULTIMEDIA APPLICATIONS

B. Girodias¹, Y. Bouchebaba¹, G. Nicolescu¹,

E.M. Aboulhamid², P. Paulin³, B. Lavigueur³

¹École Polytechnique de Montréal, ²Université de Montréal, ³STMicroelectronics

Abstract

Multiprocessor System-on-Chip is one of the main drivers of the semiconductor industry revolution by enabling the integration of complex functionality on a single chip. The techniques for processor design and application optimizations can be combined together for more efficient design of these systems. Thus, the memory optimization techniques improving the data locality can be combined with multithreading technology, improving the overall processor efficiency. The combination of these techniques is mainly challenged by the adaptation of memory optimization techniques to the high parallelism offered by the multithreading environments. This paper presents an in-depth analysis of the impact of multiprocessor and multithreading environments on memory optimization techniques. A discussion is provided on the different types of parallelization (fine and coarse grain) and their influence on memory optimization technique. Some improvements on existing memory optimization techniques are presented as well some adaptation necessary to use them in this type of environment.

3.1 Introduction

The International Technology Roadmap for Semiconductors (ITRS) defines Multiprocessor Systems-on-Chips (MPSoC) as one of the main drivers of the semiconductor industry revolution by enabling the integration of complex functionality on a single chip. MPSoC are increasing in popularity in today's high performance embedded systems. Given their combination of parallel data processing in a multiprocessor system with the high level of integration of System-on-Chip (SoC), they are great candidates for systems such as network processors and complex multimedia platforms[1]. The important amount of data manipulated by these applications requires a large memory size and a significant number of accesses to the external memory for each processor node in the MPSoC architecture[2, 3]. Therefore, it is important to optimize, at the application-level, the access to the memory and it is important to observe the effect of multithreading in order to improve processing time and power consumption.

Embedded applications are often described as streaming applications which is certainly the case of multimedia applications involving multi-dimensional streams of signals such as images and videos. In these applications, the majority of the area and power cost is due to the global communication and memory interaction[4-6]. Indeed, a key area of concentration to handle both real-time and energy/power problems is the memory system [2, 3]. The development of new strategies and techniques to decrease memory space, code size and shrink the number of accesses to the memory is necessary.

Multimedia applications often consist of multiple loop nests. Unfortunately, today's compilation techniques for parallel architectures (not necessarily MPSoC) consider each loop nest separately. Hence, the key problem associated with these techniques is that they fail to capture the interaction among different loop nests.

The main contribution of the paper is to present an in-depth analysis of the impact of multiprocessor and multithreading environments on memory optimization techniques. A discussion is done on the different type of parallelization (fine and coarse grain) and their influence on memory optimization techniques. Some improvements on existing memory optimization techniques are presented as well as some adaptation necessary to use them in this type of environment. Observations on two illustrative applications (cavity detection and demosaicing) are obtained by varying the number of processors and the number of threads on a real-life industrial MPSoC platform (MultiFlex[7]).

The MultiFlex[7] platform provides a set of tools which address the modeling, the architecture exploration and end-user programming of MPSoC systems.

In our case we used the platform to instantiate architectures including ARM multithreaded processors with one level local cache and connected through a STBus NoC to a shared memory.

The objective of this work isn't to create tools or to feign major novelty in compiling optimization domain. The objective is to facilitate the process of MPSoC application design and to observe the impact of standard optimization techniques in MPSoC environments.

The remainder of this paper is organized as follows: section 3.2 discusses related work, section 3.3 overviews techniques used in memory optimization, section 3.4 presents these techniques in application to MPSoC, section 3.5 proposes a novel approach to improve performance, section 3.6 gives an overview of the different parallelism approaches, section 3.7 discusses experimental results for a real case study, section 3.8 gives the analysis on multithreading and section 3.9 presents the concluding remarks.

3.2 Related work

In single processor environments (ex. SoC), there has been extensive research in which several techniques and strategies have been proposed to improve cache data locality.

Among them, one can point out scalar replacement[4, 8], intra array storage order optimization[9, 10], pre-fetching [10] and locality optimizations for array-based codes [6, 11]. One of IMEC groups [5] pioneered code transformation to reduce the energy consumption in data dominated embedded applications. Loop transformation techniques like loop fusion and buffer allocation have been studied extensively [12, 13]. Fraboulet *et al* [14] and Marchal *et al* [15] minimize the memory space in loop nest by using loop fusion. Kandemir *et al* [16] studied inter-nest optimizations by modifying the access patterns of loop nest to improve temporal data locality. Kandemir *et al* introduced DST [17] for data space-oriented tiling, which also aims at optimizing inter-nest locally by dividing the data space into data tiles. The work presented in this paper is different from prior efforts as it targets MPSoC. Transformations discussed in this article can be used on a large scale system; however they have more impact on a MPSoC environment. MPSoC is a more sensitive environment, because it has limited resources and is more constrained in area and energy consumption.

As proposed by Feihui L. *et al* [18], most efforts in the MPSoC domain focus on architecture design and circuit related issues [12, 19]. Compilation techniques in this domain target only single loop nests (i.e. each loop nest independently). Recently, Kandemir *et al* [18] proposed a method with a global approach to the problem. However, it is limited when the partitioned data block sizes are larger than the cache. To circumvent this problem, our research paper proposed a new method, which consists in applying loop fusion to all loop nests and partitioning the data space across the processors. In [20, 21] present a methodology for data reuse exploration. It gives great detail with formalism and presents some cost function and trade-offs. [21] presents an exploration and analysis with scratchpad memories (SPM) instead of caches. Using SPM requires the use of special instructions that are architecture dependant. Some work might also use additional architectural enhancements for performance purposes.

A technique to remove modulo operator is proposed in [22]. The solution used conditional statements which may introduce overhead cost on certain SoC architecture. Our work proposes a similar solution to remove modulo operator in conjunction with a unimodular transformation to eliminate any overhead cost. All transformations presented in our current work require no changes to the architecture and can still obtain significant performance enhancement.

In [23], an overview of the state-of-art in system-level data access and storage management is given. The authors present global loop and control flow transformations, data reuse decision, memory data layout organization, data type optimization, address optimization and illustrate them on real-life applications. The book gives a great overview on the complete design flow. It does not go into detail about multiprocessor environment and the effect it can have on its methodology. Some simulations are done on a multiprocessor SMP environment; however, only processing time is presented. Our current work concentrates on the effect that multiprocessor and multithreading can have on optimization technique. Some optimization techniques require some adaptation to be applied in a multiprocessor and multi-threading environment.

Multithreading has been very popular these past years, since it can be a simple and cost effective way to increase parallelism. Different types of multithreading have been studied. This paper uses StepNP's hardware multithreaded processors [7]. Each hardware multithreaded processor has separate register banks for different threads to keep low-overhead switching between threads. In single and multiple processor environments (but rarely in MPSoC), there has been extensive research in which several techniques and strategies have been proposed to maximize the multithreading capabilities. Most of the existing work concentrates more on the architecture side and few analyze on the effect of multithreading on existing memory optimization compilation techniques. Memory space and size are very crucial in MPSoC, which is not really the case in all multiprocessors environments.

An embedded multiprocessor architecture and an associated thread-based programming model are proposed in [24]. They were able to reduce considerably the energy for a quad-processor compared to a single-processor. The effect of multithreading on four different shared memory models is evaluated in [25]. The authors in [25] analyzed the performance of these memory models and shared their results. As all these works try to improve performance by altering the architecture, this paper concentrates on improving the performance without changing the architecture.

Other works focus on processing time and memory but only for large systems like clusters. The performance of cache is studied with a Multithreaded Virtual Processor (MVP)[3]. The performance was improved by tolerating memory latency but also lower cache miss rates due to exploitation of data locality. In [3], the paper quantifies the costs and benefits of software multithreading on a distributed memory multiprocessor. A hardware/software partitioning methodology is proposed for improving the application's performance in embedded SoC[26]. This paper looks only at MPSoC architecture where memory is a key player for performance. It looks to increase performance by exploiting compilation techniques at their best, therefore studying the effect of multithreading on them.

One of IMEC [27] research groups is studying a scheduling methodology for multithreaded system on chip. Their efforts focus primarily on reducing the power consumption and facilitating memory management. More recent work by IMEC is found in [15]. However, this work focuses on MPSoC and multithreading. Multithreading is becoming an important factor in MPSoC and the industry feels it is important to observe the impact of multithreading on memory optimization.

On the MPSoC side, a novel class of caches, named step caches is proposed [28]. It can be used to implement concurrent memory access in shared memory MPSoC without cache coherency problems. As this work concentrates on cache concurrency with a new type of

cache, the authors' work [29-32] looks more into cache and processing time optimization under the effect of multithreading without changing the existing architecture.

The main difference with our previous work is the analysis and comparison of fine and coarse grain parallelization with memory optimization transformation. Our previous work introduced new adaptations to memory optimization techniques (fusion, buffer allocation and tiling) for multiprocessor environments; however the study was only done with a coarse grain parallelization.

This paper discusses the effect of multithreading and multiprocessing on memory transformations with fine and coarse grain parallelization. Better understanding of the effect of multithreading will help designers in making better optimizations.

3.3 Memory optimization techniques

This section will review different techniques used in the compilation field, particularly in program transformations. As described earlier, loop transformation techniques such as loop fusion and buffer allocation have been studied extensively. More techniques exist but three techniques are the focus of the optimization presented further in this paper.

The following section will also demonstrate how to combine these loop transformations in an MPSoC environment. These techniques apply to any sequence of loop nests where loop nest k depends on all previous loop nests.

3.3.1 Loop fusion

3.3.1.1 Monoprocessor environment

Loop fusion is often used in applications with numerous loop nests. This technique replaces multiple loop nests by a single one. It is widely used in compilation optimization since it increases data locality in a program. It enables data already present in the cache to

be used immediately. Figure 3.1 illustrates the loop fusion technique. Details on the requirements needed to accomplish a loop fusion will be presented subsequently.

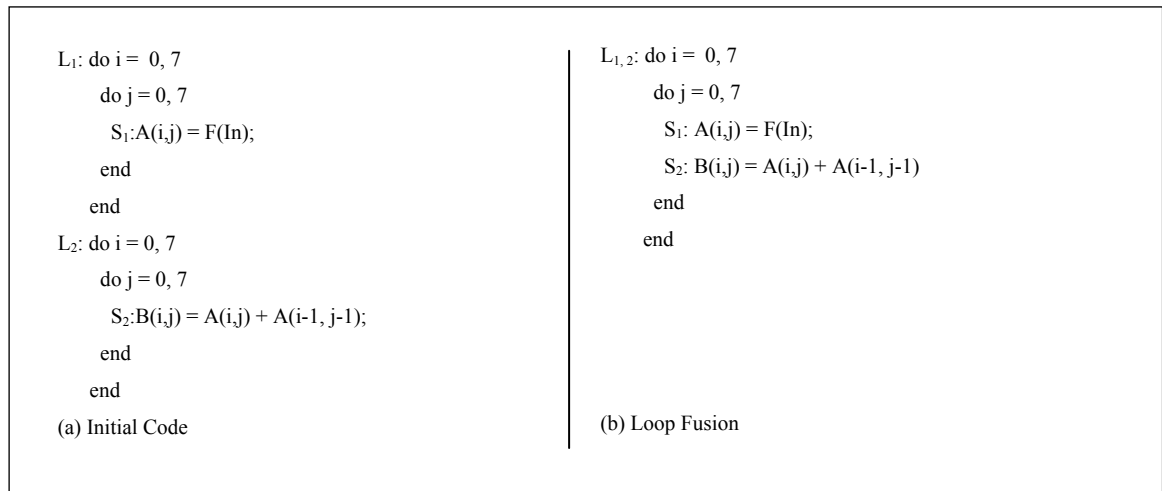


Figure 3.1 An example of loop fusion

Loop fusion cannot be applied directly to a code. All dependencies among loop nests must be positive or null. Therefore, a loop shifting technique must be applied to the code.

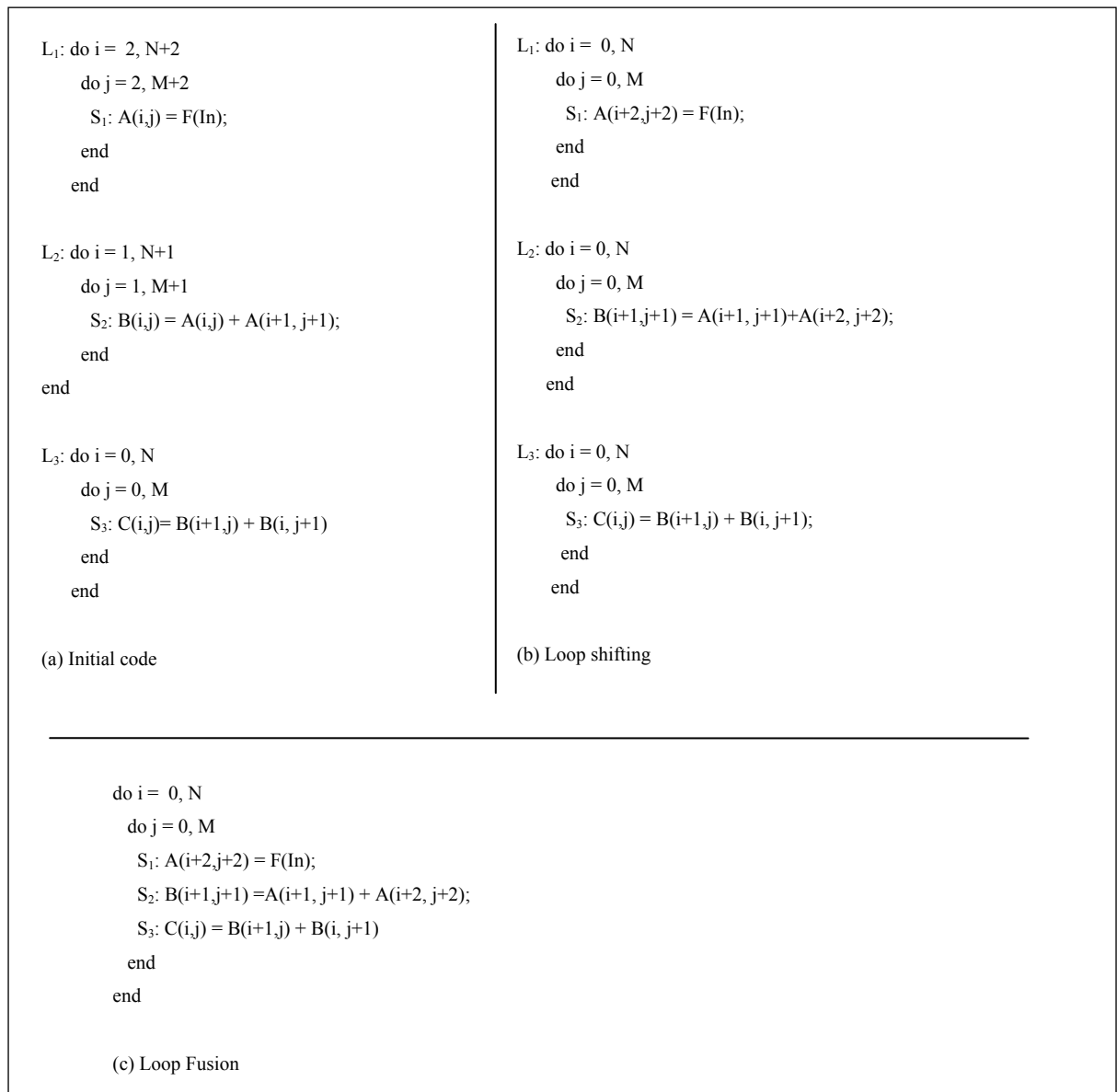


Figure 3.2: An example of 3 loop nests

Figure 3.2 illustrates the sequence of loop transformations going from the initial code (a) to a loop shifting (b) and finally a loop fusion (c). As shown in this example, one must shift the iteration domain of the loop nest L1 by a vector $(-2, -2)$ and the iteration domain of loop nest L2 by a vector $(-1, -1)$ in order to make all dependencies positive or null (≥ 0). A

lexicographically positive dependency is a satisfactory condition to apply loop fusion; however, in a parallel application it is advantageous to have positive or null dependencies.

More detail on fusion can be found in [8].

3.3.1.2 Multiprocessor environment

The code generated after a loop fusion cannot be automatically parallelized. Border dependencies appear when partitioning the application. To avoid these dependencies, elements at the border of each processor block must be pre-calculated before each processor computes its assigned block (initialization phase).

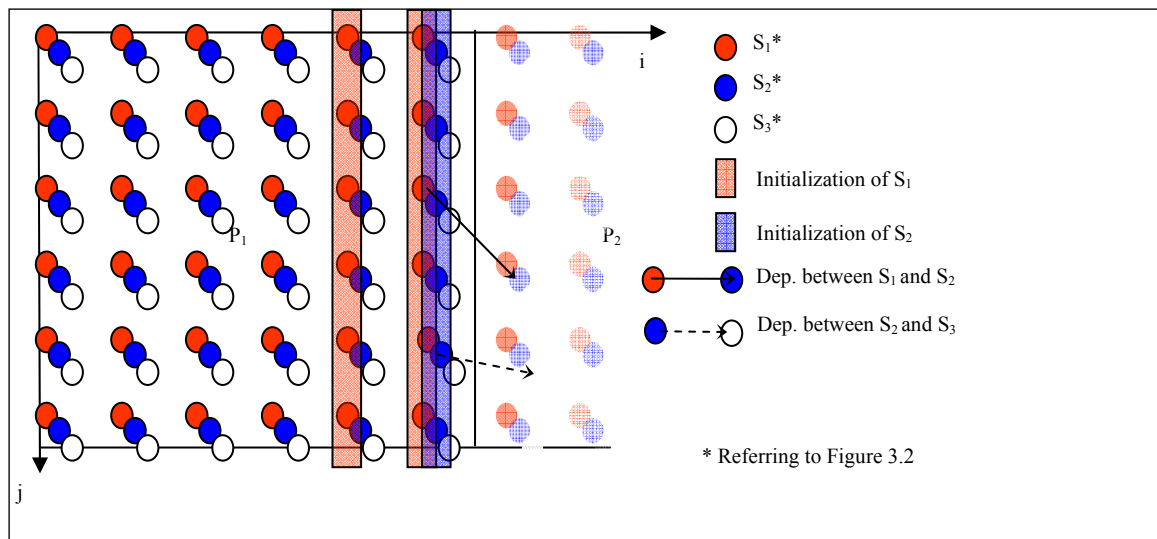


Figure 3.3: Partitioning after loop fusion

Figure 3.3 illustrates the partitioning of the code in Figure 3.2 (c) across two processors where the left-hand side of array calculations is assigned to P1 and the right-hand side to P2. As shown in Figure 3.3, dependencies between S_1 and S_2 and dependencies between S_2 and S_3 do not allow one to parallelize the code directly. In the border, processor P2 cannot compute statements S_3 and S_2 before P1 computes statements S_1 and S_2 . Therefore this

paper proposes an initialization phase where statement S_1 and S_2 on the border of processor P_1 block must be pre-calculated before processing concurrently each processor assigned block. This solution is possible since S_1 does not depend on any statement which is the general scenario in the types of applications studied in this paper. More detail on fusion in a multiprocessor environment can be found in [30, 31].

3.3.2 Buffer allocation

3.3.2.1 Monoprocessor environment

Multimedia applications often use temporary arrays to store intermediate computations. To reduce memory space in this type of application, several techniques are proposed in the literature like scalar replacement, buffer allocation and intra array storage order optimization. Nevertheless, these techniques are used in monoprocessor architectures. It decreases memory space and reduces the cache miss ratio. The buffer size is defined by the dependencies among statements. The buffer will contain only useful elements (also called live elements). An element of an array is considered live at time t , if it is assigned (written) at t_1 and last used (read) at t_2 whereas $t_1 \leq t \leq t_2$. Figure 3.4 illustrates an example of the buffer allocation technique where array A is replaced by the buffer BUF of size 10. More detail on buffer allocation can be found in [29].

<pre> do i = 0, 7 do j = 0, 7 S₁: A(i,j) = F(In); S₂: B(i,j) = A(i,j) + A(i-1, j-1); end end </pre>	<pre> do i = 0, 7 do j = 0, 7 S₁: BUF[(8*i+j)%10] = F(In); S₂: B(i,j) = BUF[(8*i+j)%10] + BUF[(8*(i-1)+(j-1))%10]; end end </pre>
(a) Initial Code	(b) Buffer Allocation

Figure 3.4: An example of buffer allocation

3.3.2.2 Multiprocessor environment

In a monoprocessor architecture, each array is replaced by one buffer containing all live elements. However in a multiprocessor architecture, the number of buffers replacing each temporary array depends on: (1) the number of processors, (2) the depth of the loop nest, (3) the division of the iteration domain and (4) the dependencies among loop nests. Two types of buffers are needed: (a) buffers for inner computation elements of blocks assigned to each processor and (b) buffers for the computation of elements located at the border of these blocks. The last buffer type is needed for the initialization phase as seen in the previous sub-section.

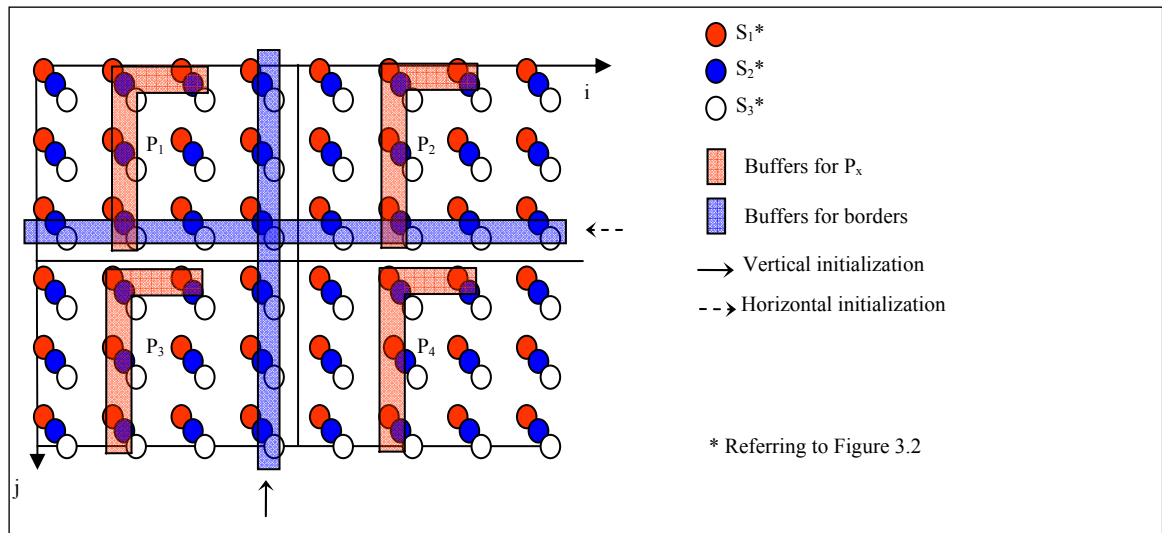


Figure 3.5: Buffer allocation for array B

Figure 3.5 illustrates the buffer allocation for just one temporary array (array B) of the code in Figure 3.2 (c) partitioned across 4 processors. The initialization phase is needed to compute in parallel these blocks (the vertical initialization for processors P_2 and P_4 and horizontal initialization for processors P_3 and P_4). In this example, six buffers are needed in

total, one for each of the 4 processors and two buffers for the initialization phase. More detail on buffer allocation in a multiprocessor environment can be found in [30, 31].

Since type (a) buffers can be seen as circular structures, modulo operator (%) is used to manage them. Using buffer allocation reduces memory space but increases processing time. This issue will be revisited later.

3.4 Techniques impacts

Optimizing, a specific aspect of an application, does not come for free. The techniques described in the last section have increased tremendously the cache hit ratio in a multiprocessor architecture, but computation time and code size have increased as well. It is certainly a major concern since MPSoCs are often chosen for their high data processing and have small memory space for applications.

3.4.1 Computation time

Buffer allocation uses extensively modulo operators which are very time consuming operations for any processor. Every time one must read or write into a buffer, it uses one or more modulo operators. In SoC, these instructions are often done with co-processors.

As seen in the previous section, two types of buffers are needed in a multiprocessor architecture. This means that when a processor is processing a statement, it must be aware of the location of the elements that will be used for computation. If the computation is located close to the border of the processor's block, some data needed to compute will be located in one of the buffers used to pre-calculate borders' elements. If the computation is not located close to a border, data will be taken from the buffer for inner computation of the block. Each processor must add extra operations to test which buffer it will get the data for the computation. These operations are "if statements" which are also known to be time consuming.

Using fusion and buffer allocation in a multiprocessor increases data locality, but requires the consideration of border dependencies between paralleled processor data blocks. Some code must be added to take into account these dependencies. Therefore, the code size is increased. The size increase depends on the size of the application. For instance, with the Cavity Detection application (see section 3.7.2), we have observed an increase of 200%.

3.4.2 Code size increase

In the buffer allocation technique, the size of the application is increased by adding extra code for tests seen in the last sub-section. However fusion and buffer allocation in multiprocessors, the code size is mostly increased by all the code needed to manage and partition the parallel application across processors and extra code to pre-calculate the elements located in each border of the processor's block (initialization phase).

Using buffer allocation decreases memory space, but requires modulo operators for buffer management. Using modulo operators increases processing time especially on platforms like MPSoC, where the embedded processor are more limited and where co-processors may be used for special instruction like modulo. Therefore, the processing time is increased. For instance, with the Cavity Detection application (see section 3.7.2), we have observed an increase of 10%.

3.5 Improvement in optimization techniques

As discussed in the previous section, the optimization techniques described earlier significantly improve the cache hit ratio, but this is done at the expense of processing time. This section depicts improvements to save significantly in processing time by (1) changing the partitioning, (2) eliminating modulo operators and (3) changing the order of the iterations with a unimodular transformation.

3.5.1 Data partitioning

This section presents a novel manner to partition the code across the processors to eliminate supplementary tests needed to manage and locate data in multiple buffers. Besides that this new block assignment gives great advantage at the level of the processing time, it also makes the code easier to parallelize. This is important if these techniques are automated in a parallel compiler.

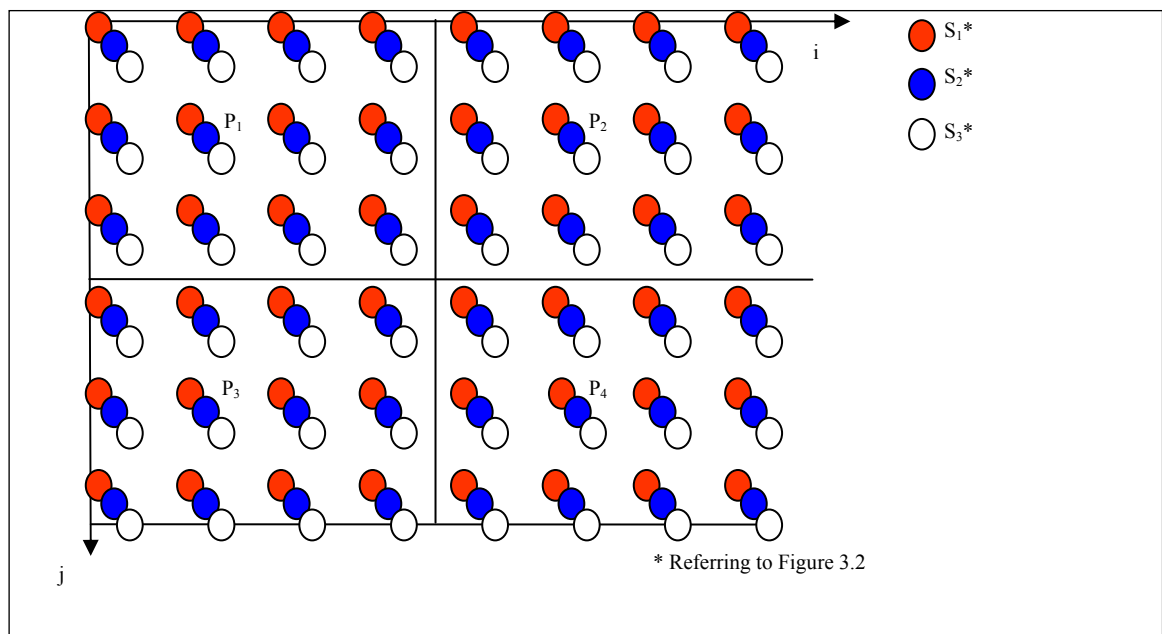


Figure 3.6: Classic partitioning

Figure 3.6 illustrates what one can expect to see in the literature. This division affects the processing time since each processor may interact with several others. This fact is even more significant in a buffer allocation scenario, where broad computation is needed to manage buffers.

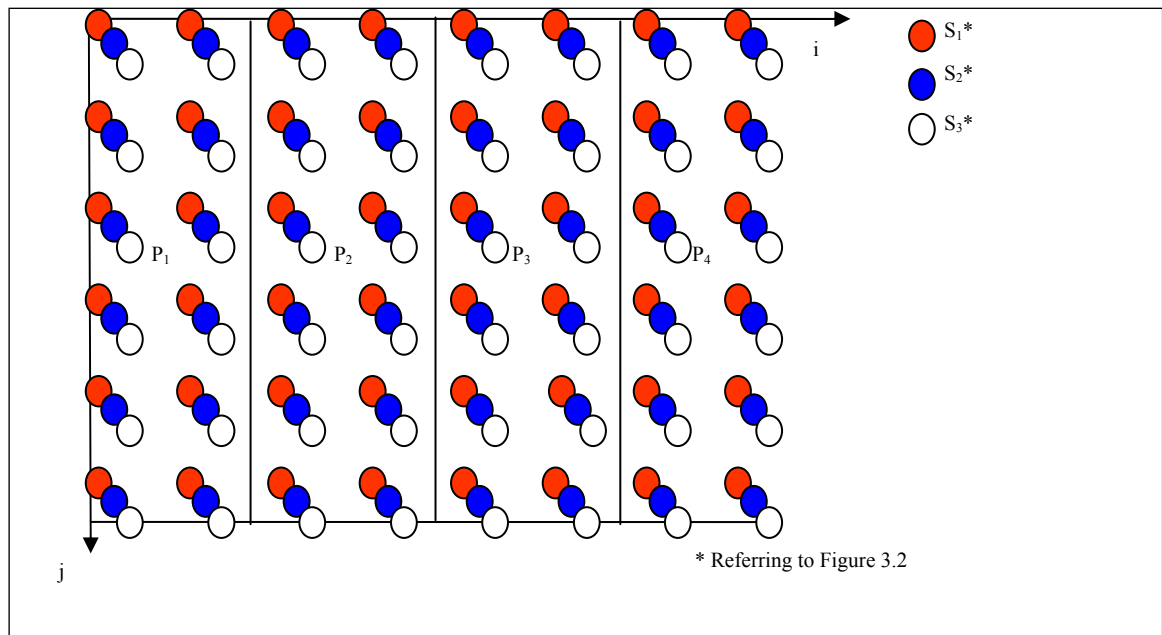


Figure 3.7: Paper's partitioning

Figure 3.7 illustrates the partitioning proposed in this section (along one axis). This block assignment reduces processing time by decreasing the number of interactions needed among processors. Furthermore, it eliminates vertical dependencies introduced by using a partitioning technique as shown in Figure 3.7. By dividing the iteration domain along one axis, the buffers used to store the elements located in the processor's block borders are not required any longer. A single type of buffer is used for both the border and block computations. This partitioning is more intuitive and the calculation of the processor's block boundary remains the same regardless of the number of processors.

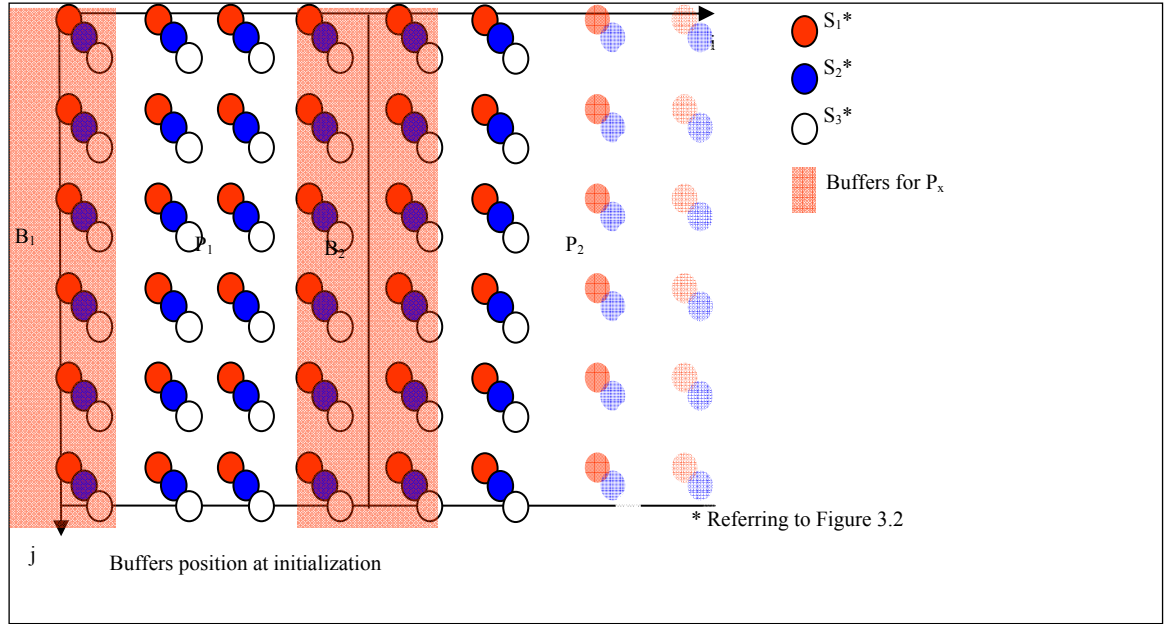


Figure 3.8: Buffer allocation for array B with new partitioning

Figure 3.8 illustrates the buffer allocation for just one temporary array (array B) of the code in Figure 3.2 (c) divided across 2 processors. Only one buffer is required for each processor (B_1 for P_1 and B_2 for P_2). The total number of buffers is strictly equal to the number of processors. The size of each buffer is equal to $M_j \times (d+1)$ where M_j is the number of iterations of axis j and d is the highest dependency along the i axis. As seen in Figure 3.8, buffers are located at the border of each processor's block at the initialization phase, and then shift along the i axis during computation. No further supplementary tests are needed to determine from which buffers data should be recovered. Each processor recovers data from only one buffer only since there is presently just one type of buffer.

Using different data partitioning may reduce the code size and facilitate the parallelization of the data; however if one restricts oneself to one type of data partitioning, data locality may decrease depending on the application, image size and cache size. Using a 2 axes data partitioning assures a better data fit in the memory cache, however it necessitates more work for border dependencies. Using a 1 axis data partitioning facilitates

Figure 3.9 demonstrates the division of processor P_1 's block into sub-blocks of equal size as the buffer. The loop z which goes across the sub-block is completely unrolled. Here, the unrolling technique is used to optimize the time spent computing the modulo operator.

<pre> for (i =6; i < 12, i++) for (j = 0, j < 6; j++) S1:A(i%2,j) = ... S2:B(i,j-1) = A((i-1)%2,j-1) + A(i%2,j) end end </pre> NOTE: Modulo of number which is a power of 2, can be done by shifting, but this technique works with any numbers. (a) Example with modulo	<pre> for (i = 6 ; i < 12; i+2) // z = 0 for (j = 0; j < 6; j++) S1: A(0,j) = ... S2: B(i,j-1) = A(1,j-1) + A(0,j) end // z = 1 for (j = 0; j < 6; j++) S1: A(1,j) = ... S2: B(i+1,j-1) = A(0,j-1) + A(1,j) end end </pre> (b) Example without modulo
--	---

Figure 3.10: Elimination of modulo operators

Figure 3.10 illustrates the elimination of the modulo operators. The loop i in Figure 3.10 (b) executes an equivalent of 2 loops at each iteration (computing sub-blocks). By unrolling the loop scanning the sub-blocks, the access to the buffer is done with constants which are defined by dependencies (elimination of modulo operators).

Traditionally, loop unrolling is used to exploit data reuse and explore instruction parallelism; however this paper uses this technique to eliminate the modulo operator.

Using the modulo operator elimination technique reduces the processing time, however the technique proposed uses loop unrolling which, depending on the unrolling factor (in our case the modulo factor), increases the code size. For this reason, we have proposed to combine the next optimization to correct this potential problem. With the Cavity Detection application (see section 3.7.2), we were able to reduce the processing time by 30% with insignificant code size increase.

3.5.3 Unimodular transformation

In conjunction with the previous optimization, a final optimization can be done on the code (Figure 3.10 (b)) obtained after the fusion and the buffer allocation without modulo. This optimization is the fusion of the two innermost j loops which will decrease processing time, and increase the cache hit ratio. However, this transformation cannot be applied directly.

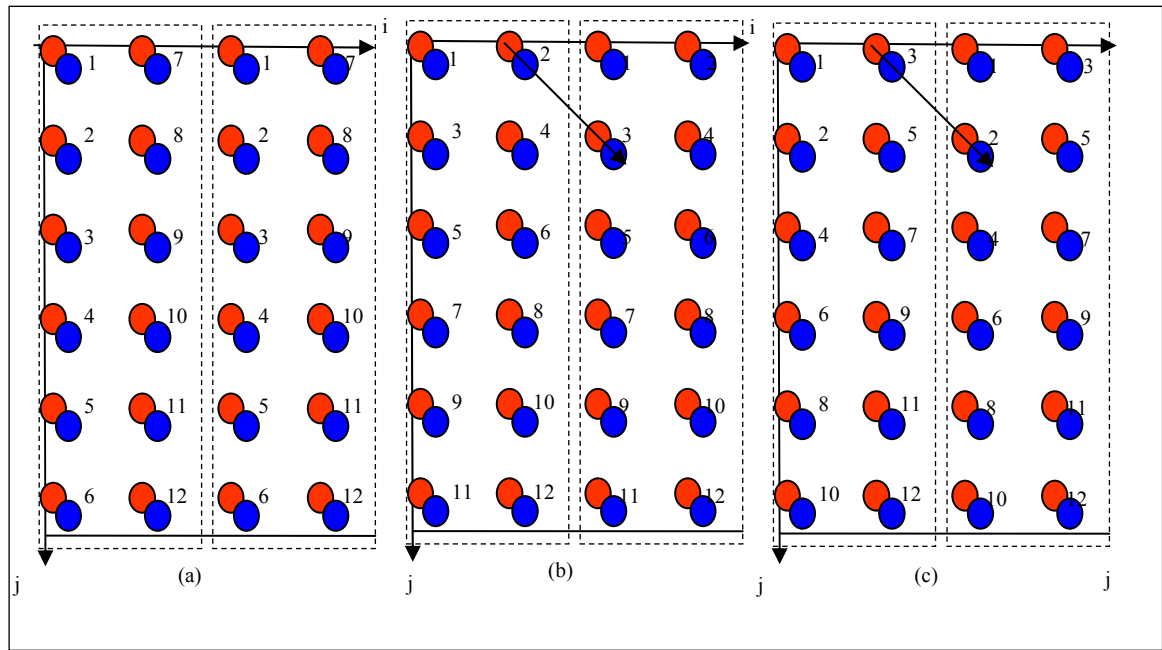


Figure 3.11: Execution order (a) without fusion (b) after fusion and (c) after unimodular transformation

Merging j loops in Figure 3.10 (b) changes the execution order of the statements inside a sub-block which corresponds to the application of a unimodular transformation $T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ on each sub-block. Figure 3.11 (b) illustrates the issue by applying fusion directly. Elements generated by iteration 2 in sub-block 1 are needed by iteration 3 of the second sub-block. However, this element will have been erased by iteration 2 in the second sub-block since all

sub-blocks share the same buffer. Figure 3.11 (c) illustrates the execution order to avoid this issue. One must apply a unimodular transformation $T = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ to each sub-block. This matrix is a function of dependencies. Through this transformation, the processing time has decreased and the cache hit ratio has increased. This optimization eliminates any overheads emerging from the modulo operators elimination technique.

Using the unimodular transformation reduces the code size lead by the previous modulo operator elimination technique, however for some applications; it may require more effort to find the appropriate unimodular transformation to respect all dependencies.

3.6 Parallelization

This section describes two different approaches used for parallelization. In this paper, the parallelization is again studied for performance in a multiprocessor environment, but with the additional consideration of multithreading. Multithreading helps to hide memory latency, but thread management costs are added [33]. There is often interference between threads, consequently stressing the memory hierarchy [33]. Having multiple threads at the same time increases the number of cache misses. Having a good parallelization approach can help increase performance with multithreading and avoid these disadvantages. Two approaches will be used for parallelization: fine and coarse grain.

The first approach is the fine grain parallelization. In a code containing several embedded loops (e.g. i1, i2... in), fine grain parallelization consists in parallelizing the inner loops (e.g. in). The second approach is a coarse grain parallelization. The main objective is to divide the computation domain into several big blocks. In a code containing several loops (i1, i2... in), coarse grain parallelization consists of parallelizing the outer most loop (i1).

Granularity of the parallelization influences metrics like cache hit ratio and processing time. The number of data partitioned will increase and their size will decrease. Useful data will have a better fit in the cache and the number of memory access will be reduced. However, like we will see in the following sections, the granularity may limit some memory optimization techniques.

Our work starts with the sequential version of the application. Implementing a fine grain parallelization is facilitated by the fact that there are no border dependencies to take into account. For the coarse grain, we have to take into account the border of the parallelized block which makes it harder to implement and necessitated an initial phase.

Fine grain is easier to implement, however a unimodular transformation is needed to respect the order of execution. Finding this transformation may be a very hard task.

The following sub-sections will demonstrate each different parallelization for each previous optimization done on a multimedia application. The application simulated consists of an imaging application used in medical applications (see section 3.7.2).

3.6.1 Initial code

This is the initial code of the application which hasn't been transformed for any optimization. This code is used as a reference. It is composed of many computations ($c_1, c_2 \dots c_n$) where each computation corresponds to a loop nest of depth 2 (i.e. composed of two loop (i, j)). The first computation is done on an input image. Then, the computation results are passed to the following computation. This code uses temporary arrays for each computation results and therefore uses a lot of memory space. Figure 3.12 illustrates 3 of the n computations of the application (3 loop nests) partitioned on 2 processors each with 2 threads. Each dot in a loop nest represents a pixel computation. The objective is to start computing the loop nest of c_1 in parallel, and then c_2 in parallel etc. This parallelization is simple to implement, but the data locality is not taken into account, resulting in poor

performance or high data bandwidth. As proposed in the previous section, to circumvent this problem, a possible solution is to use the technique of loop fusion.

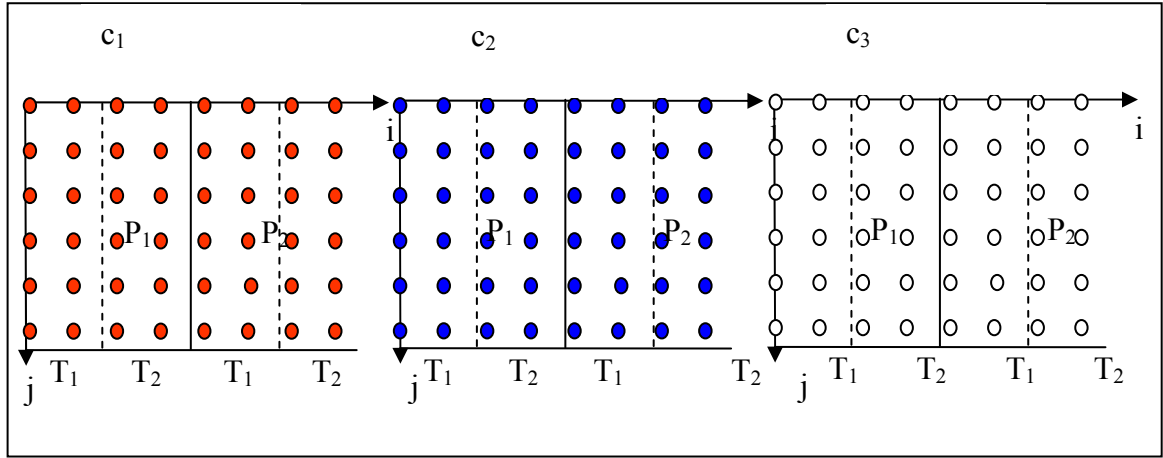


Figure 3.12: Part of the initial code partitioned on 2 CPUs and 2 threads

3.6.2 Code with fusion

This is the code after the transformation for optimization of data locality. All loop nests which can be merged have been fused together to optimize the reuse of data in the cache. The loop nests c_1 , c_2 and c_3 are depicted in Figure 3.13 with the three colored dots. The merged code will be parallelized using two different approaches. The first approach consists of using fine grain parallelization and the second approach consists of using coarse grain parallelization.

3.6.2.1 Fine grain parallelization

The code generated after a loop fusion cannot be directly parallelized because of data dependencies. Figure 3.13 a) illustrates the execution order of dots in the merged code. It is not possible to parallelize the loop i or j , because of the vector dependency $(1, 1)$. To avoid the dependency problem shown by the arrow in Figure 3.13 a), a unimodular

transformation is used to change the order of execution (the execution order is represented by the numbers). Figure 3.13 b) illustrates this unimodular transformation. All dots of each diagonal line (strikethrough dots) are executed in parallel and all different diagonal lines are computed sequentially. In code generation, the merged code with two loops i and j will be transformed to another merged code with two loop i' and j' , such that j' will be parallelizable.

This technique corresponds to fine grain parallelization. Partitioning the code will consist in distributing all dots under a diagonal line across different processors or threads. For example, the set of dots 4, 5 and 6 can all be done in parallel after the computation in parallel of the sets 2 and 3.

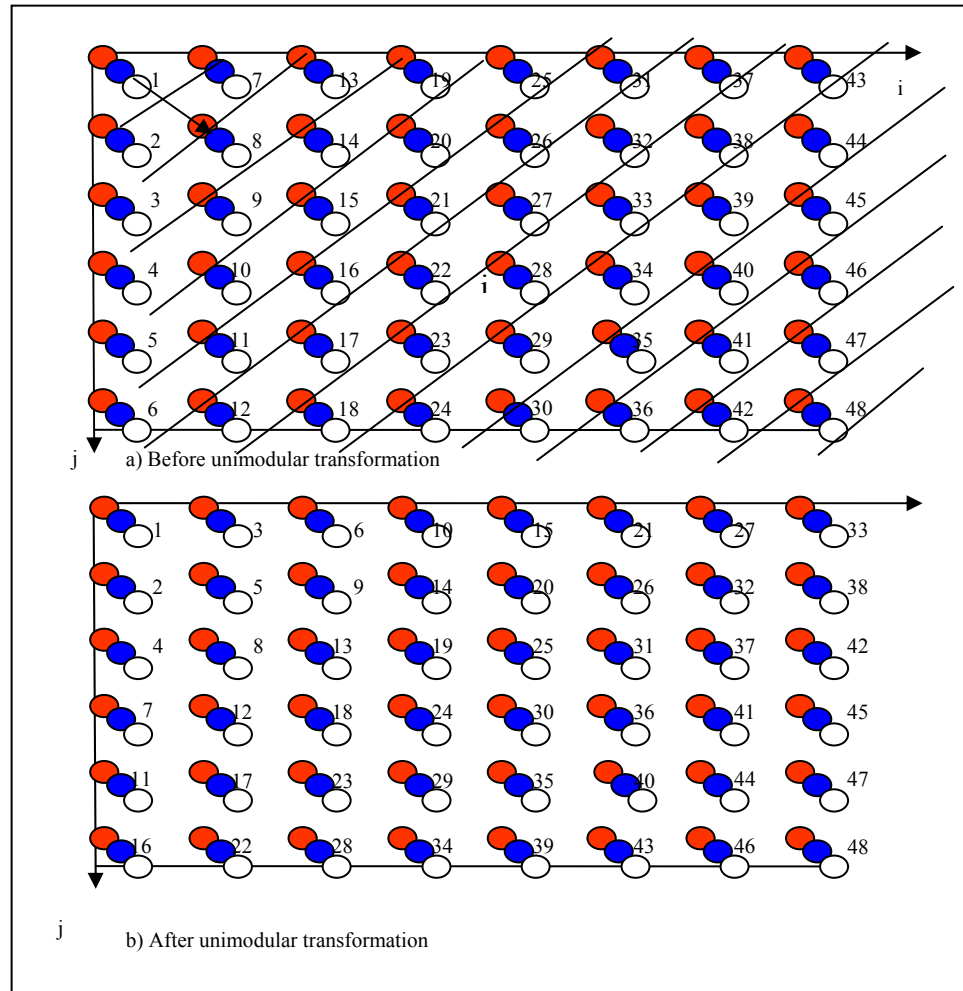


Figure 3.13: Unimodular transformation

3.6.2.2 Coarse grain parallelization

The second parallelization is the coarse grain approach. The objective of coarse grain is to parallelize the outer most loop (loop i) which was not the case in a fine grain parallelization where the inner most loop was parallelized (loop j). In a coarse grain approach, the iteration domain is divided into several blocks which will be computed in parallel. However, as seen earlier, border dependencies appear when partitioning the application with a coarse grain approach.

With the coarse grain approach, the iteration domain is divided into equal blocks. The number and size of the partitioned blocks depend on the number of processors and/or the number of threads. Figure 3.14 illustrates some examples of parallelization: 2 CPUs and 2 threads and 4 CPUs and 2 threads. The dotted lines represent the limitation of the partitioned block of the threads and the full lines represent the partitioned block of the processors. For all these different partitionings, the initialization phase shown in Figure 3.3 is always needed.

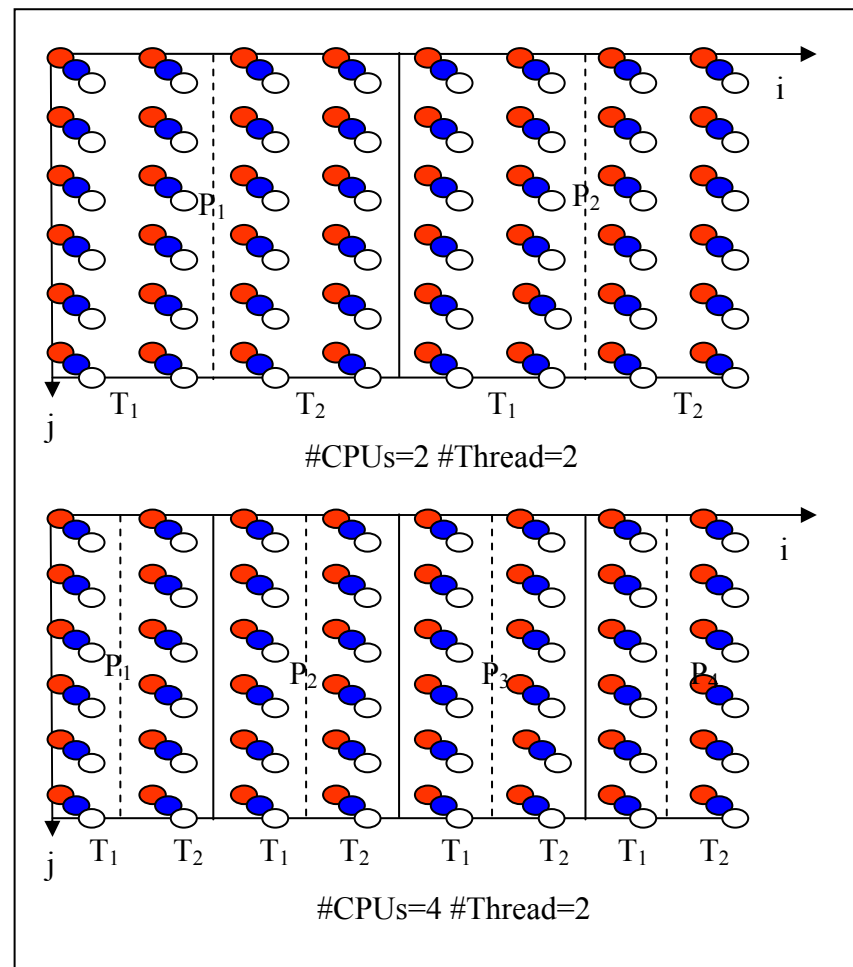


Figure 3.14: Examples of partitioning

3.6.3 Code with fusion and buffer allocation

Buffer allocation is often used for applications with temporary arrays which store intermediate computations. The paper's buffer allocation differentiates itself from the standard array contraction/privatization and storage mapping methods by the fact that it's adapted to multiprocessor. The standard techniques cannot be applied directly to a parallelized code. Extra buffers have been added to satisfy data dependencies between processing units. In the array contraction, scalars are used as buffers for dependencies of depth size equal to 1. This adjustment supports any size of dependency.

3.6.3.1 Buffer allocation in a fine grain approach

To apply the buffer allocation in a fine grain parallelization approach, one must replace each temporary array by a smaller buffer which will be common to all processors. This will create a coherency problem and will complicate code generation; this will be studied in future work. Figure 3.15 illustrates the case where operations 1, 2 and 3 have been computed and have stored their results in the buffer A which is shared by all processors. When the operations 4, 5 and 6 will be computed in parallel, the processors associated to each operation will get their data in the same buffer (A) which will create some coherency problems. The cache coherency is due to the lack of cache coherency support in the architecture that we used. This problem does not occur for the coarse grain parallelization approach. In this approach, each processor is processing a different data block, therefore not working on the same memory space. In the fine grain parallelization, all processors are working in the same memory zone, therefore creating some coherency problem. Coarse grain parallelization guarantees cache coherency without having a system with hardware cache coherency support. It is often the norm in SoC system not to have hardware support

for cache coherency. For cost and space issue, designers prefer to handle cache coherency at the software level.

Therefore, the fine grain approach is not suitable in this case.

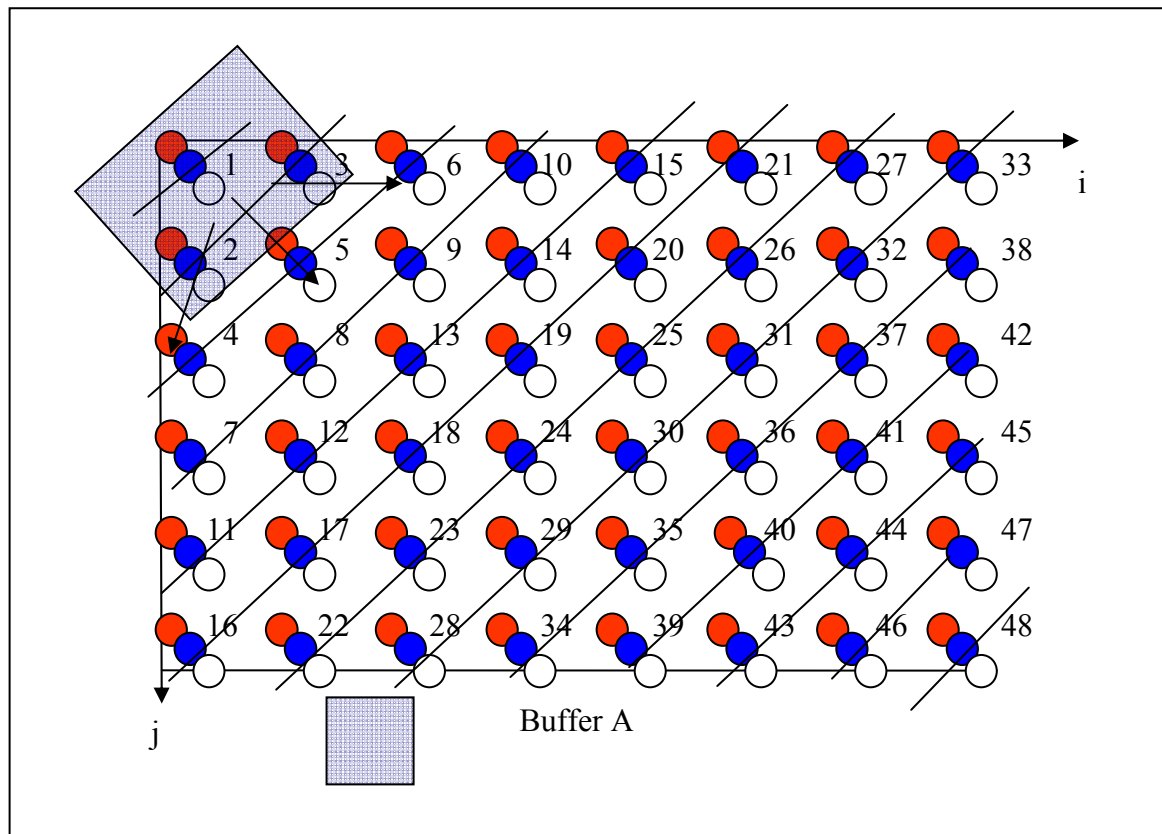


Figure 3.15: Buffer allocation in fine grain approach

3.6.3.2 Buffer allocation in a coarse grain approach

To apply the buffer allocation in a coarse grain parallelization approach, one must replace each temporary array by multiple smaller buffers where each buffer is associated to a processor. The coherency problem is avoided since each processor now has its own buffer. In this paper's experimentations, two different versions of the buffer allocation are

used. The first version has been applied to the merged code without optimization (modulo operators are used to manage the buffers). This technique reduces memory space but using modulo operators increases processing time. The second version is similar to the last one, however optimization to eliminate the modulo operators has been done. This optimization is based on loop unrolling and unimodular transformation.

3.7 Applications

This section presents briefly the general type of applications used for these type of transformations and describes two specific application used in this article.

3.7.1 General

Our work is targeted to signal processing and multimedia applications which are data dominated. They satisfy theses following conditions:

- They are sequences of nested loops: each loop nest reads the image/frame produced in the preceding computation, makes some computation, and produces (writes) a new image/frame. For that reason, the dependencies are between consecutives loop nests.
- There are no dependencies in the same loop nest, because in general, each loop nest of the pipeline does not read and write in the same image. This is a consequence of the preceding characteristic.
- Each computation (loop nest) of the pipeline takes as input the images produced in the preceding loop nests and produces an intermediate image. As all these images have the same dimensions, the corresponding arrays will also have the same dimensions. Each dimension of this array will be accessed by one loop.
- Each loop nest will work on the entire image or frame. For this reason the loop bounds are constants.
- The access functions are uniform.

We have selected two applications: (1) Cavity Detection and (2) Demosaicing which satisfy these characteristics. Our industrial partner is working on new applications which share these common characteristics (e.g. Temporal Noise Reduction Algorithm).

More detail on the reasons why these techniques can be applied to the majority of multimedia applications can be found in this paper [30, 31].

3.7.2 Cavity Detection

Cavity detection is used in medical imaging and it is composed of 5 computations (5 loop nests: L_1 , L_2 , L_3 , L_4 and L_5) on an input image. Each computation (loop nest) takes as input the image computed in the previous nest, carries out its computation, and provides an image to the following loop nest. The images produced by L_1 , L_2 , L_3 and L_4 are stored in temporary arrays. All the loop nests are parallel and they have the same depth (2 loops). The arrays manipulated by the different loop nests are 2-dimensional arrays and they have the same access function (except for the constants). Figure 3.16 gives the data flow graph of this application.

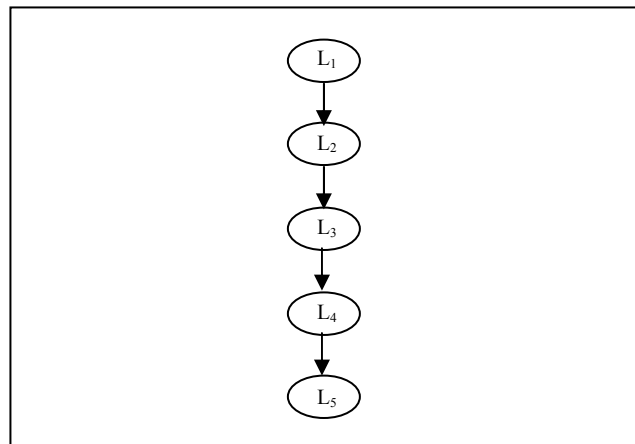


Figure 3.16: Cavity Detection Application

3.7.3 Demosaicing

Demosaicing application is a digital camera application that performs three interpolations on the input image: (1) the green interpolation, (2) the blue interpolation, and (3) the red interpolation. The green interpolation computes the missing green pixels and it is composed of two loop nests L_1 and L_2 . The red interpolation is done after the green interpolation and it is composed of five loop nests L_3 , L_4 , L_5 , L_6 and L_7 . The Blue interpolation is also done after the green interpolation and it is composed of five loop nests L_8 , L_9 , L_{10} , L_{11} and L_{12} . All these loop nests have the same characteristics as those of cavity application, except that each loop nest can use the elements produced in all preceding loop nests. Figure 3.17 gives the data flow graph of this application.

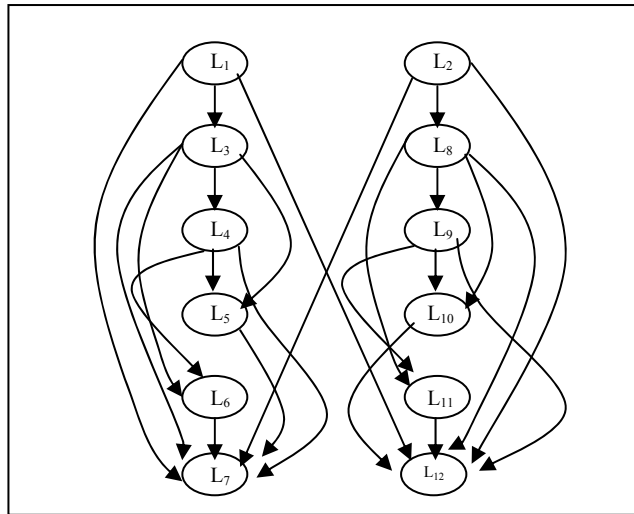


Figure 3.17: Demosaicing Application

3.8 Experimental results

Experiments were carried out on the StepNP multiprocessor SoC programming environment [7] which focuses on two programming models: a distributed system object component (DSOC) message passing model and a symmetrical multi-processing (SMP)

model using shared memory. The StepNP tools map these models onto the Multiflex multiprocessor SoC platform[7]. StepNP is a cycle accurate simulation tool that integrates heterogeneous parallel components (H/W or S/W) into a homogeneous platform programming environment. The platform supports different types of processing elements (ARM, XTensa ...), interconnects (buses, bridge and crossbars), memories (cache, shared memory...), coprocessors... The platform is fine grain parametrisable. The platform's compiler doesn't do any optimizations.

The platform used for our experiments has multiple ARM (200MHz) with hardware threads, each processor with one level local cache (4-way set-associative, 4 bytes blocks) and connected to a shared memory by modern interconnects (i.e. NoC) (32bit, 10 cycles latency + jitter). The programming model selected is the SMP model using shared memory. This paper will look at three metrics: the results of cache hit ratio, processing time obtained by each simulation and the code size.

3.8.1 Memory optimization technique improvements

The experiments for memory optimization techniques consisted of four different simulations: (1) the initial code without any transformations, (2) the initial code with fusion and buffer allocation using modulo operators, (3) the initial code with fusion and buffer allocation without using modulo operators and (4) the initial code with fusion and buffer without using modulo, but with a unimodular transformation.

The results are obtained by changing the cache size (0.5, 1, 2, 4, 8, 16, 32 and 64 KiB). Different processors configuration have been used, but since similar results are observed, this paper presents only the 4 CPUs configuration. For these experiments the cavity detection application was selected.

3.8.1.1 Cache ratio and memory space

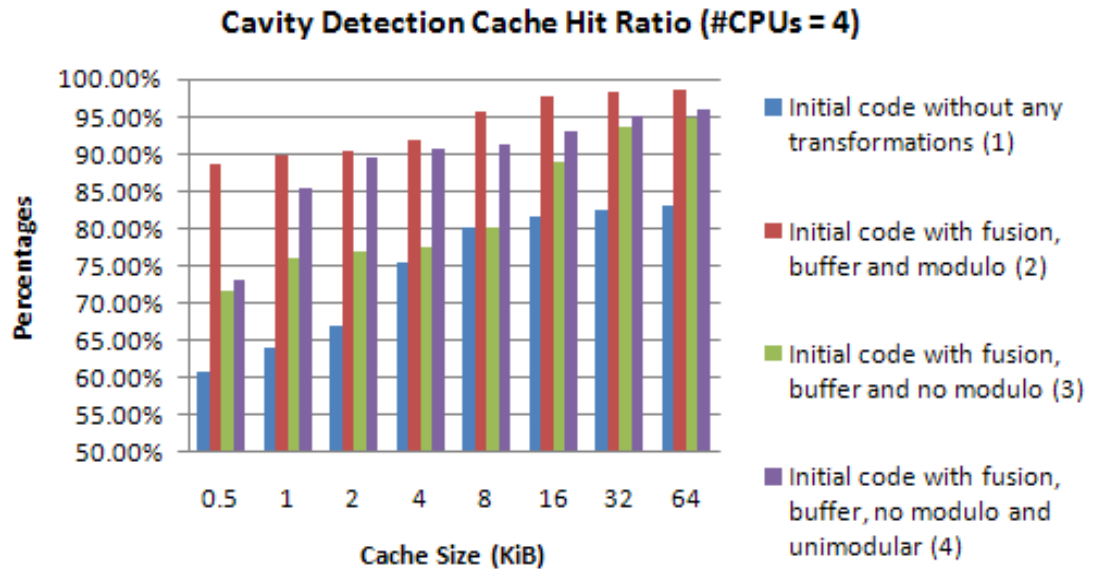


Figure 3.18: DCache hit ratio #CPU=4

Figure 3.18 shows the data cache hit ratio of the multimedia application on a multiprocessor architecture (4 CPUs) with a 4-way set-associative cache with a block size of 4 bytes. As one can see, most of the techniques presented in this paper considerably increase the cache hit ratio compared to the initial application. The best results are obtained with the fusion with buffer allocation using modulo operators. One can observe an average increase of 20% of the data cache hit ratio.

The combination of the loop fusion, buffer allocation and mainly the partitioning reduce the memory space by approximately 80%.

3.8.1.2 Processing time and code size

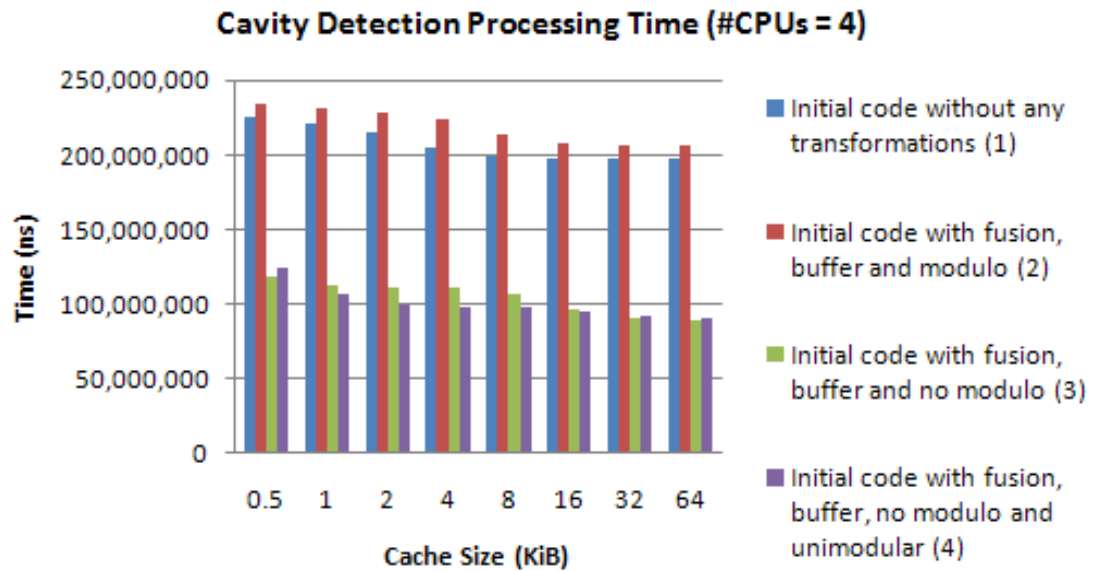


Figure 3.19: Processing time #CPU=4

Figure 3.19 shows the processing time of the multimedia application on a multiprocessor architecture (4 CPUs) with a 4-way set-associative cache with a block size of 4 bytes. As discussed in the previous section, the fusion with buffer allocation using modulo operators (2) is great for cache hit but at the expense of prolonging processing time (see Figure 3.19). However, the two others techniques (3) and (4) show great improvements in processing times while still increasing the data cache hit ratio. The best results are seen with the fusion with buffer allocation using no modulo operators with a unimodular transformation. One can observe an average decrease of 50% of the processing time.

The partitioning, proposed here, reduces the code size by approximately 50% (compared to the classical partitioning presented earlier) in the case of the fusion.

3.8.1.3 Power consumption

By optimizing key metrics (processing time, cache misses, channel transactions, memory space, and code size) and by avoiding any coprocessor for instruction like modulo, we can expect to significantly reduce power consumption. The IMEC research group in [5] pioneered the work on program transformations to reduce the energy consumption in data dominated embedded applications and they have stated that, in these applications, memory accesses can account for 70% of the global energy consumption. Based on this result, and in our cache miss reduction result (20%), we can estimate an energy consumption reduction of $70 * 20 / 100 = 14\%$. This estimation is based only on data transfer reductions. Other parameters will also play a role in energy consumption reduction.

Our present work uses coarse grain calculation based on IMEC's observation [5]. However, we may explore power model similar to model found in [34] to greatly improve the granularity of the power consumption estimation.

3.8.2 Multiprocessor and multithreading effect

For the effect of multithreading, the experiments consisted of five different transformations: (5) the initial code without any transformations, (6) the initial code with fusion and coarse grain parallelization, (7) the initial code with fusion and fine grain parallelization, (8) the initial code with fusion and buffer allocation with modulo operators (coarse grain) and (9) the initial code with fusion and buffer without using modulo operator (coarse grain). Concerning the buffer allocation in versions 8 and 9, the difference is that modulo operators were removed from the buffer allocation technique. As seen earlier, this improves the performance by reducing the execution time and by increasing the data locality. Since the Demosaicing application does not require modulo operators, the initial code with fusion and buffer without using modulo operator (9) was not applied to the application

The results are obtained by changing the number of processors (1, 2, 4, 8 and 16) and changing the number of threads (1, 2, 4, 8, 16 and 32). Given that there would be too many results to present, this paper focuses on the results obtain by the configurations 1/16, 4/4 and 16/1 (CPU/Thread-per-CPU). These configurations each give 16 partitioned blocks processed by either processors or threads, and therefore a comparison can be done. For these experiments the Cavity Detection and Demosaicing applications were use.

3.8.2.1 Cache ratio

This sub-section present the different results obtain for the Data Cache Hit Ratio with the Cavity Detection and Demosaicing applications.

3.8.2.1.1 Cavity Detection

Figure 3.20 presents the results of the cache hit ratio obtained with each transformation with the configuration 16/1, 4/4 and 1/16 (CPU/Thread-per-CPU).

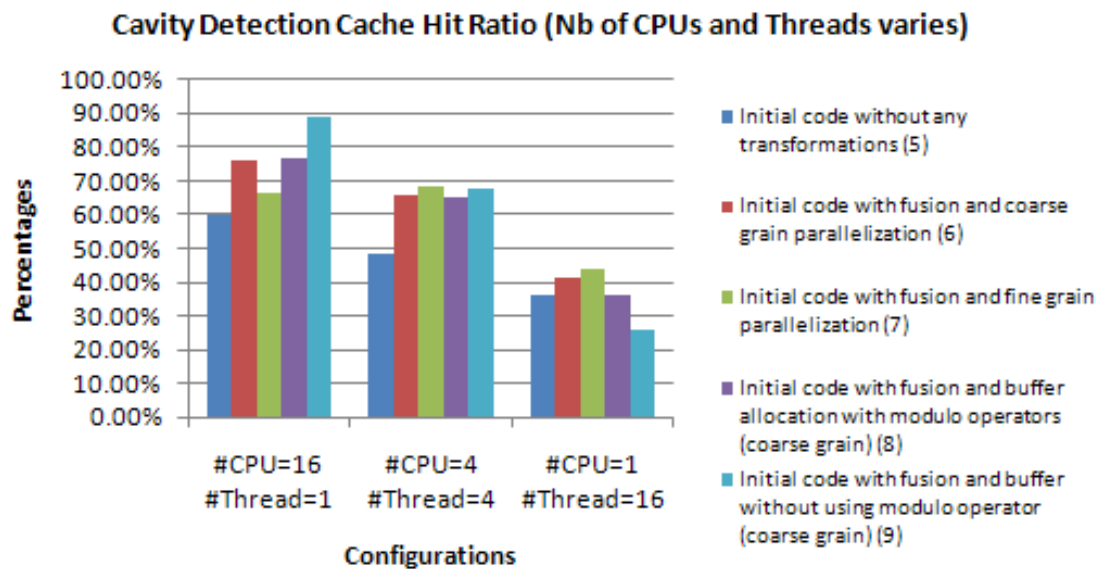


Figure 3.20: Cavity Detection DCache hit ratio

Looking at the different transformations, for a configuration of 1 processor and 16 threads (last column), the transformation with fusion and fine grain parallelization (7) gives the best result. The worst result is obtained by the last transformation: initial code with fusion and buffer allocation optimized (9).

For the configuration of 4 processors and 4 threads, the transformation with fusion and fine grain parallelization (7) still gives the best result, but the transformation (9) which was the worst at the previous configuration is now the second best. The worst result is now obtained with the initial code (5).

For the last configuration of 16 processors and 1 thread, the best result is obtained by the previous configuration's second best: the transformation with fusion and buffer allocation optimized (9). The transformation, fusion and fine grain parallelization (7), which was the best at the other configurations, is now the second worst result, the worst result still being the initial code (5). The two other transformations (6) and (8) follow similar changes as the transformation (9), since they use similar transformations.

From Figure 3.20, we can see that it is more advantageous to have more processors than threads for the cache hit ratio.

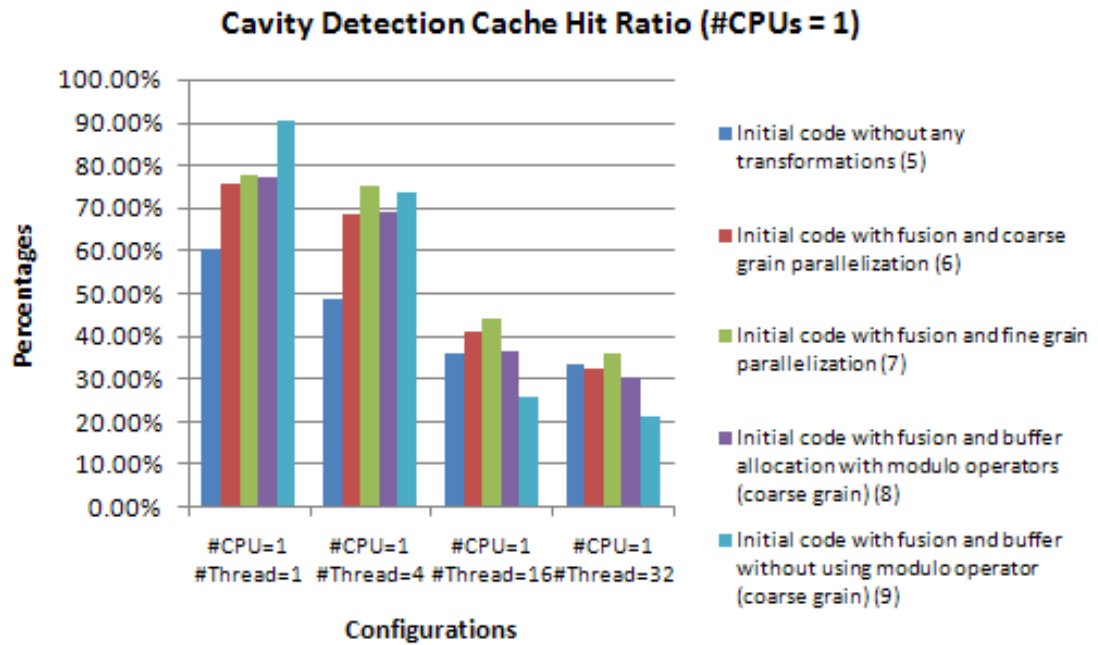


Figure 3.21: DCache hit ratio #CPUs=1

Figure 3.21, Figure 3.22 and Figure 3.23 demonstrate the effect of multithreading on the cache hit ratio. Increasing the number of threads decreases the cache hit ratio. However, increasing the number of processors does not affect the results much. The results obtained with 1 processor or 16 processors are relatively similar. Threads share the same memory space while processors each have their own memory space.

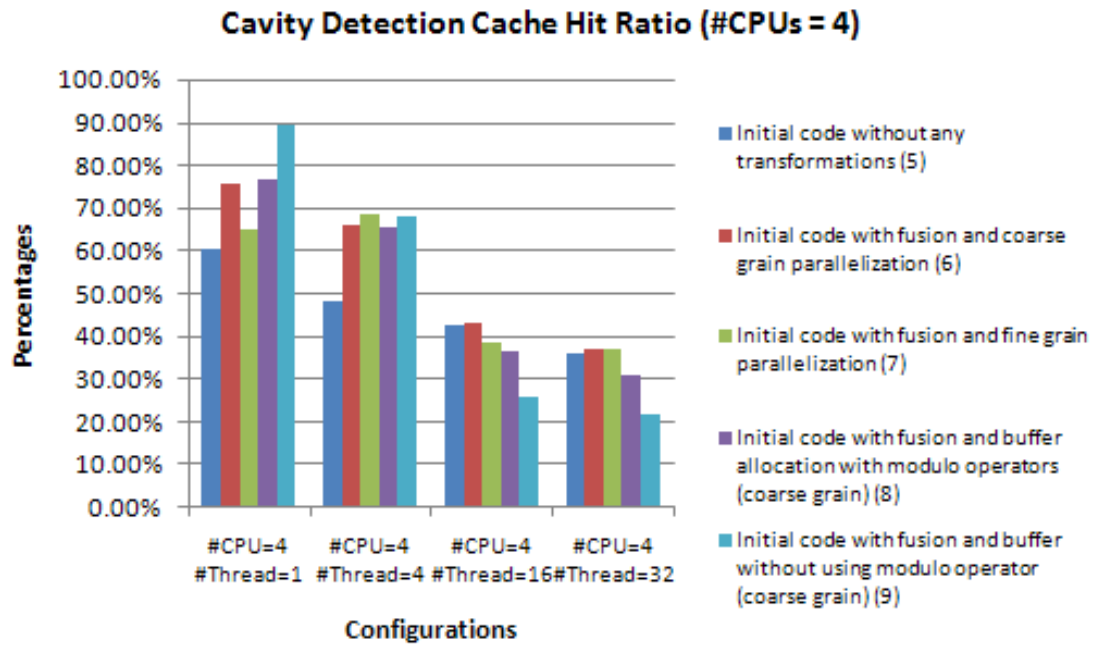


Figure 3.22: DCache hit ratio #CPUs=4

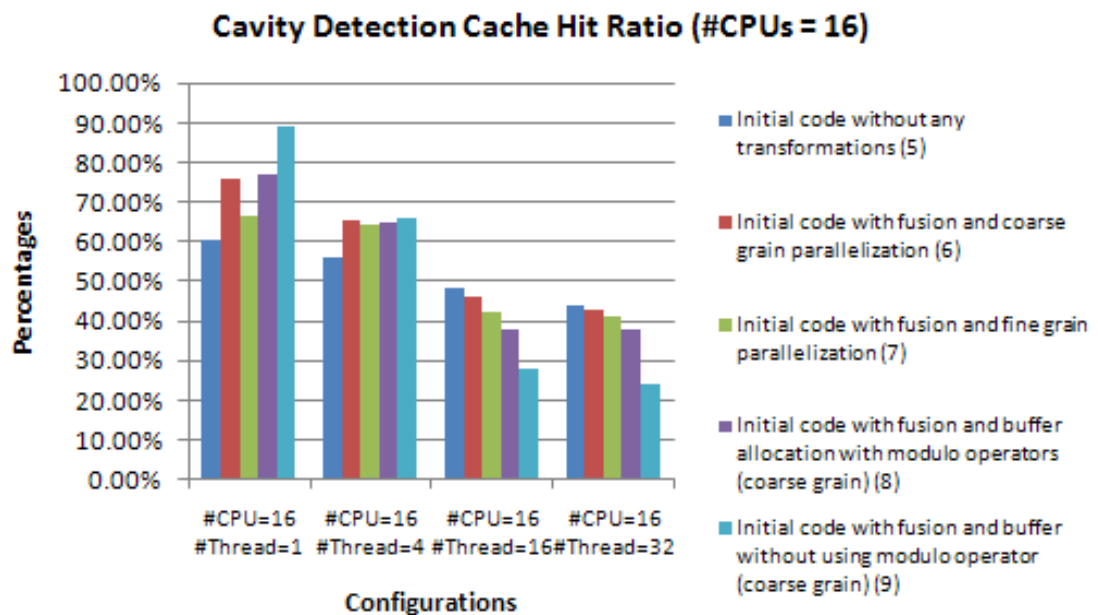


Figure 3.23: DCache hit ratio #CPUs=16

Figure 3.24 demonstrate the effect of increasing the number of CPUs on the cache hit ratio. Increasing the number of CPUs doesn't affect the cache hit ratio. Like said previously, each processor have their own memory space, therefore the cache hit ratio is not affected.

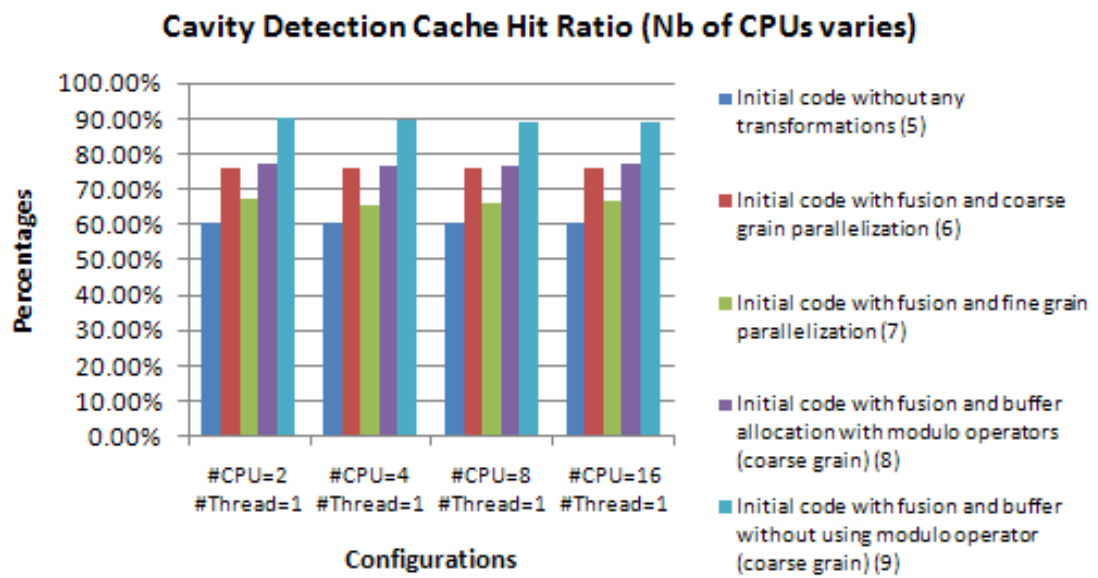


Figure 3.24: DCache hit ratio #CPUs varies

3.8.2.1.2 Demosaicing

Figure 3.25 presents the results of the cache hit ratio obtained with each transformation with the configuration 16/1, 4/4 and 1/16 (CPU/Thread-per-CPU).

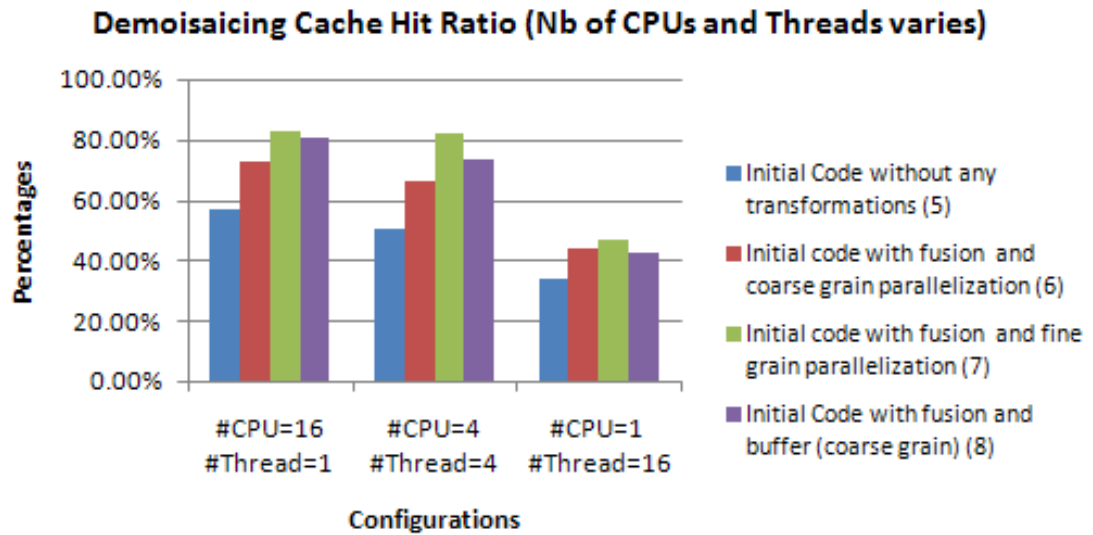


Figure 3.25: Demoisaicing DCache hit ratio

Looking at the different transformations, for each configuration, the transformation with fusion and fine grain parallelization (7) gives the best result. The worst result is obtained by the initial code (5). The tendency is maintain all across the different configurations. Once again, one can see that it is more advantageous to have more processors than threads for the cache hit ratio.

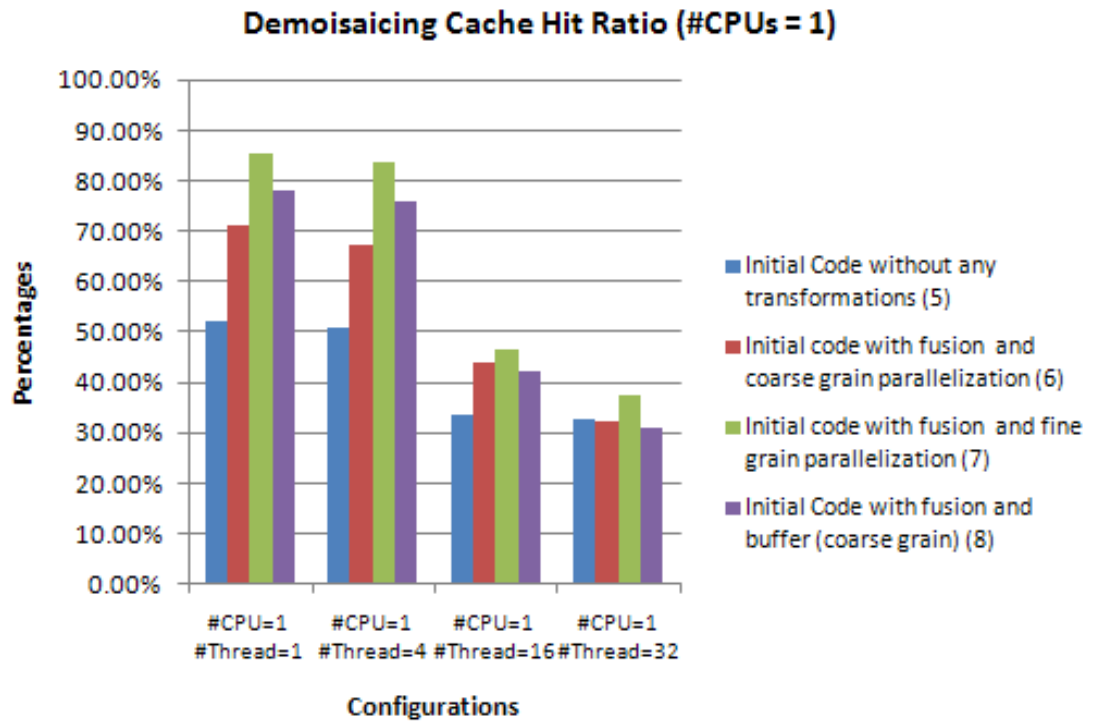


Figure 3.26: DCache hit ratio #CPUs=1

Figure 3.26, Figure 3.27 and Figure 3.28 demonstrate the effect of multithreading on the cache hit ratio. Increasing the number of threads decreases the cache hit ratio. Threads share the same memory space while processors each have their own memory space. One can see that the cache hit ratio of the Initial Code without transformation (5) increase at the configuration 4/32 and 16/32. Since the number of processing elements is increasing, the data block size is decreasing and may fit better into the cache.

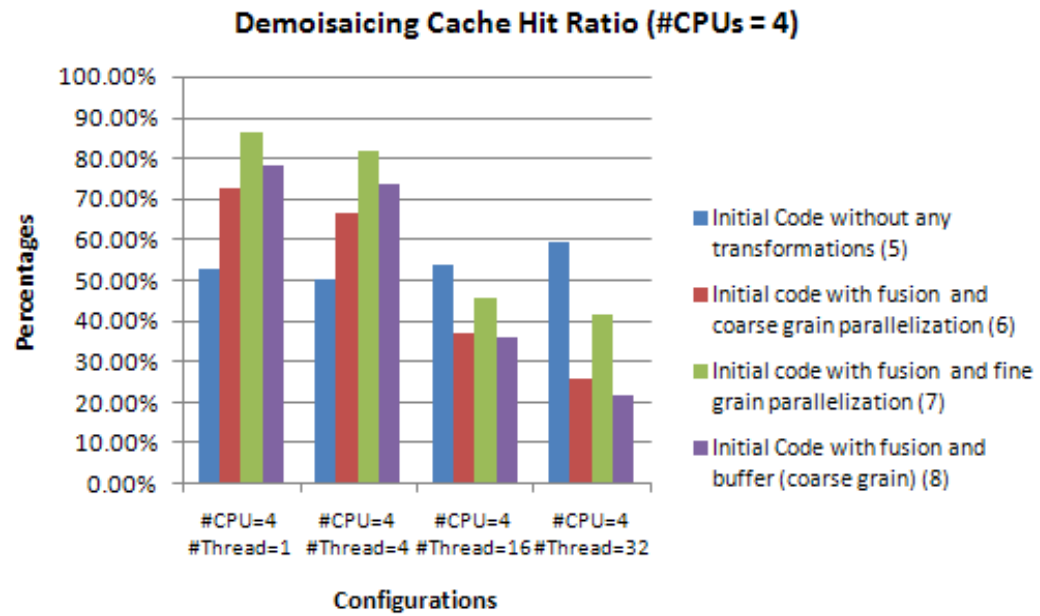


Figure 3.27: DCache hit ratio #CPUs=4

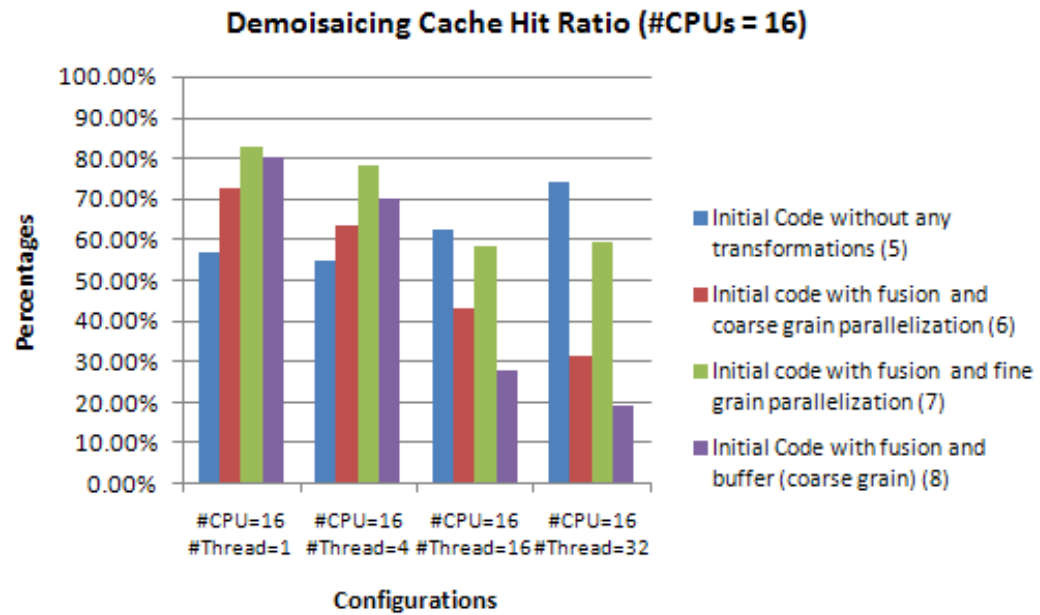


Figure 3.28: DCache hit ratio #CPUs=16

Figure 3.29 demonstrate the effect of increasing the number of CPUs on the cache hit ratio. Increasing the number of CPUs doesn't affect in general the cache hit ratio. Like said previously, each processor has their own memory space therefore the cache hit ratio is not affected. However the Initial code with fusion and fine grain parallelization (7) is a bit affected. In a fine grain parallelization, each processing unit is working on the same data area, which may affect the cache hit ratio. In the coarse grain parallelization, each processing unit is working on different independent data area.

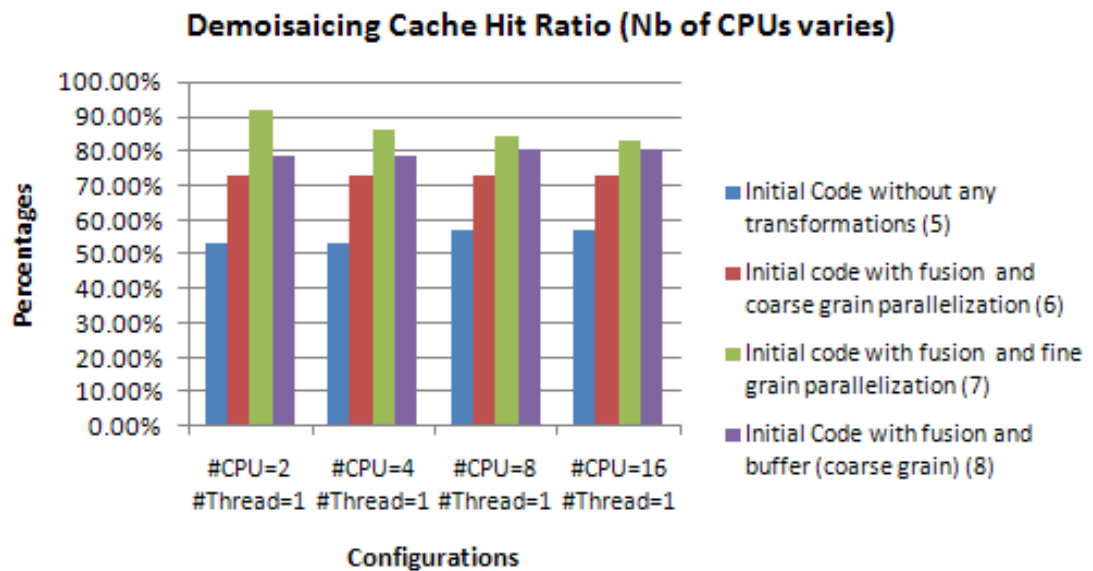


Figure 3.29: DCache hit ratio #CPUs varies

3.8.3 Processing time

This sub-section present the different results obtain for the processing time with the Cavity Detection and Demosaicing applications.

3.8.3.1.1 Cavity Detection

Figure 3.30 presents the results of the processing time obtained with each application with the configuration 16/1, 4/4 and 1/16 (CPU/Thread-per-CPU).

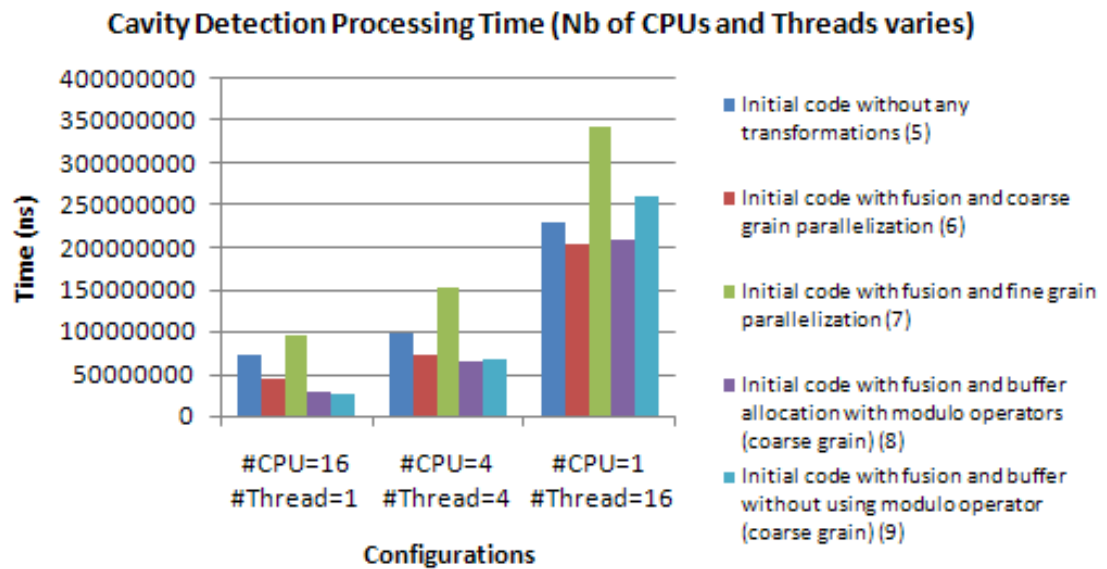


Figure 3.30: Processing time (ns)

Looking at the different transformations, for a configuration of 1 processor and 16 threads, the application with fusion (6) gives the best result. The worst result is obtained by the application with fusion and fine grain parallelization (7). For the configuration of 4 processors and 4 threads, the application with fusion and buffer allocation (8) obtains the best result. The worst result is still obtained with the application with fusion and fine grain parallelization (7). For the last configuration of 16 processors and 1 thread, the best result is now obtained by the application with fusion and buffer allocation optimized (9). The worst result is still obtained by the application with fusion and fine grain parallelization (7). It is not surprising to see the application (9) having the best result since this technique was optimized for processing time in a multiprocessor environment.

From Figure 3.30, one can see that it is more advantageous to have more processors than threads for the processing time.

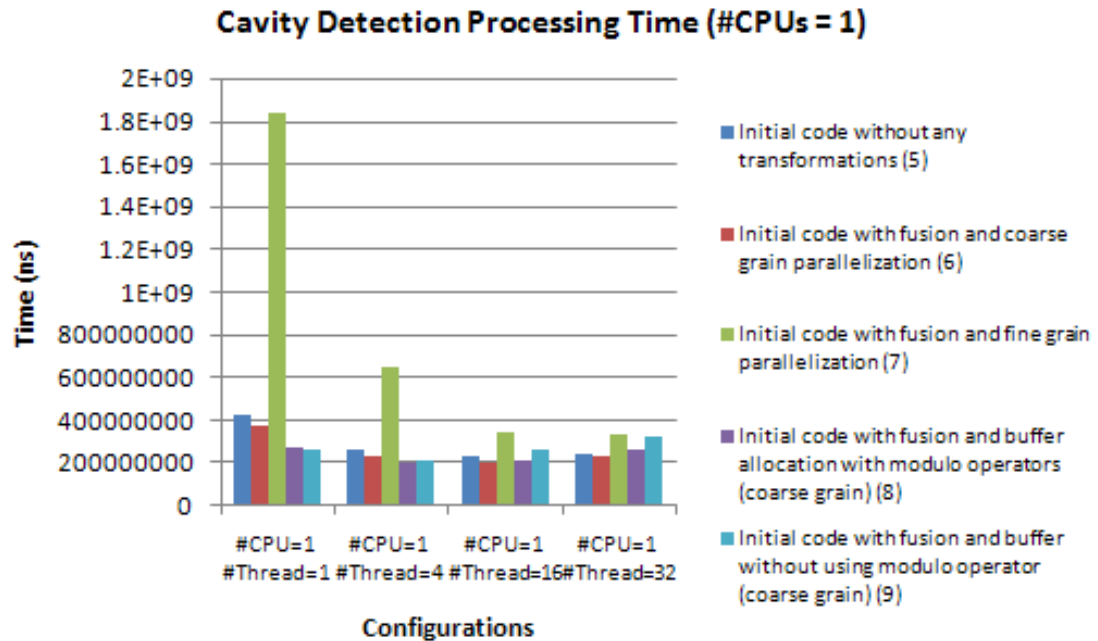


Figure 3.31: Processing time (ns) #CPUs=1

Figure 3.31 and Figure 3.33 demonstrate the effect of multithreading on processing time. In a monoprocessor architecture, increasing the number of threads decreases the processing time, and in a multiprocessors architecture the same observation should be expected, however the results obtained do not yield this conclusion. In fact, increasing the number of threads decreases the processing time, however if the application does not require a great number of threads, a lot of processing time will be spent managing the threads. Therefore, it is important not to saturate or over parallelize the application. The application must define the architecture.

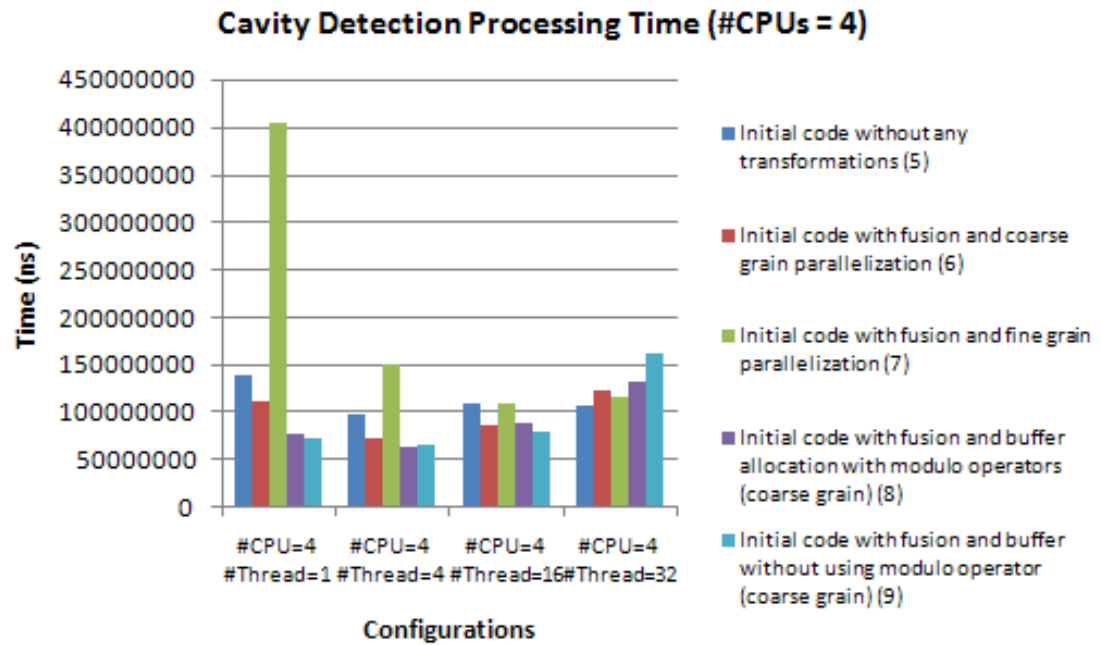


Figure 3.32: Processing time (ns) #CPUs=4

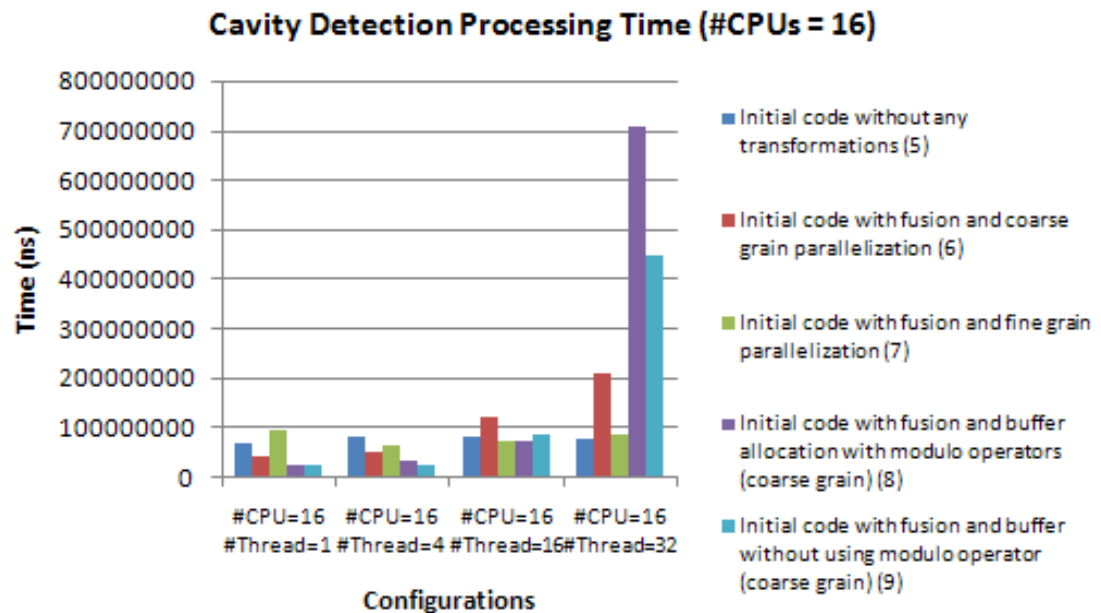


Figure 3.33: Processing time (ns) #CPUs=16

Figure 3.34 demonstrate the effect of increasing the number of CPUs on the processing. Increasing the number of CPUs decreases the processing time.

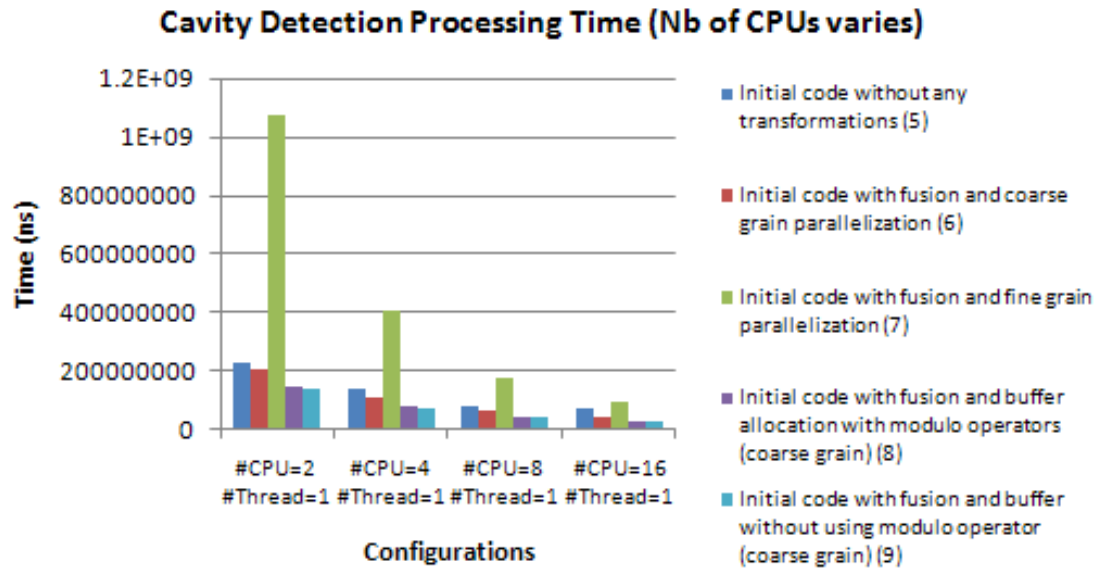


Figure 3.34: Processing time (ns) #CPUs varies

3.8.3.2 Demosaicing

Figure 3.35 presents the results of the processing time obtained with each application with the configuration 16/1, 4/4 and 1/16 (CPU/Thread-per-CPU).

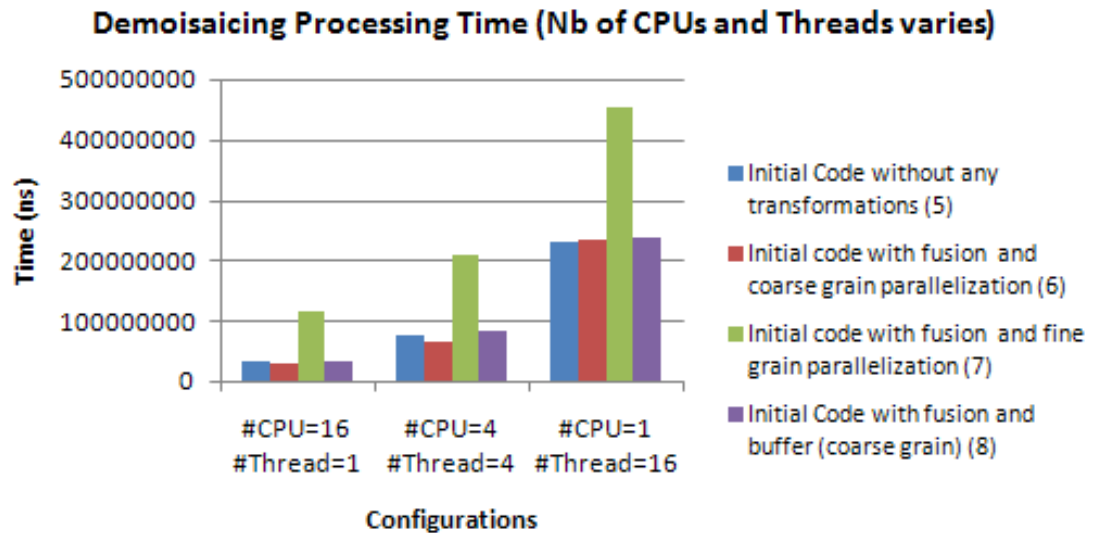


Figure 3.35: Processing time (ns)

Looking at the different transformations, for a configuration of 1 processor and 16 threads, the application with fusion and coarse grain parallelization (6) gives the best result. The worst result is obtained by the application with fusion and fine grain parallelization (7). For the configuration of 4 processors and 4 threads, the application with fusion (6) still gives the best result. The worst result is still obtained with the application with fusion and fine grain parallelization (7). For the last configuration of 16 processors and 1 thread, the best result is now obtained by the application without any transformation (5) and closely followed by the application with fusion and coarse grain parallelization (6). The worst result is still obtained by the application with fusion and fine grain parallelization (7). Since the number of processing elements is increasing, the data block size is decreasing and may fit better into the cache. One can see that the cache hit ratio of the Initial Code without transformation (5) increase at the configuration 4/32 and 16/32. Therefore the number of access to the main memory is reduced which decrease processing time.

Once again, one can see that it is more advantageous to have more processors than threads for the cache hit.

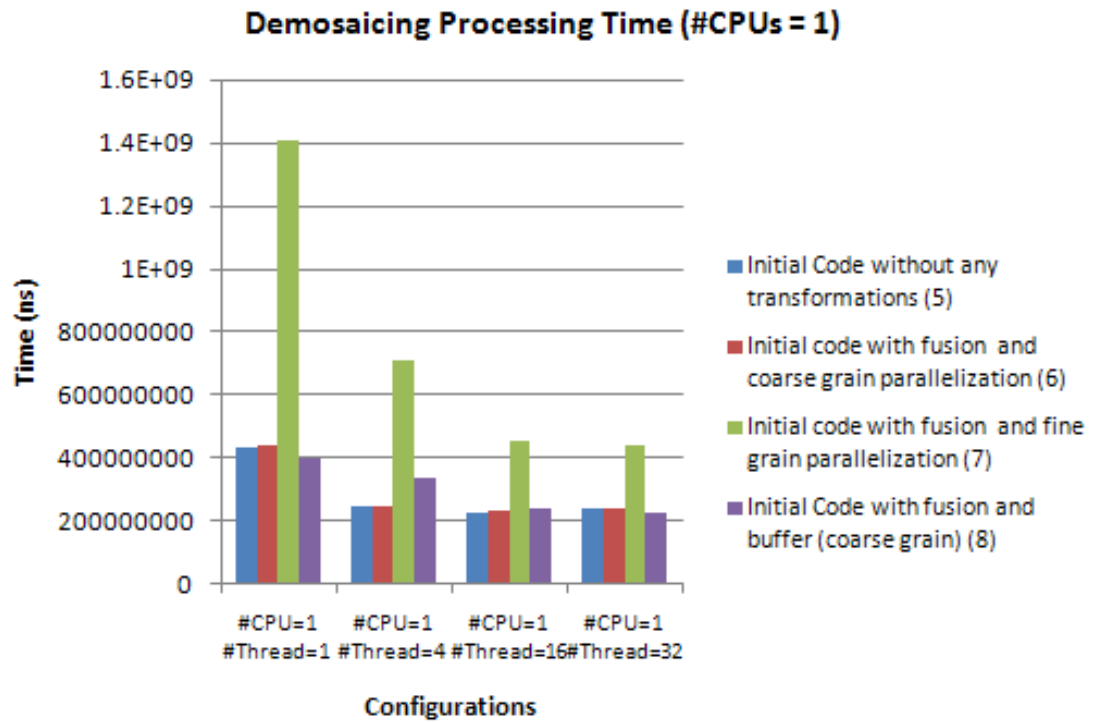


Figure 3.36: Processing time (ns) #CPUs=1

Figure 3.36, Figure 3.37 and Figure 3.38 demonstrate the effect of multithreading on processing time. Once again, it is important not to saturate or over parallelize the application. The application must define the architecture.

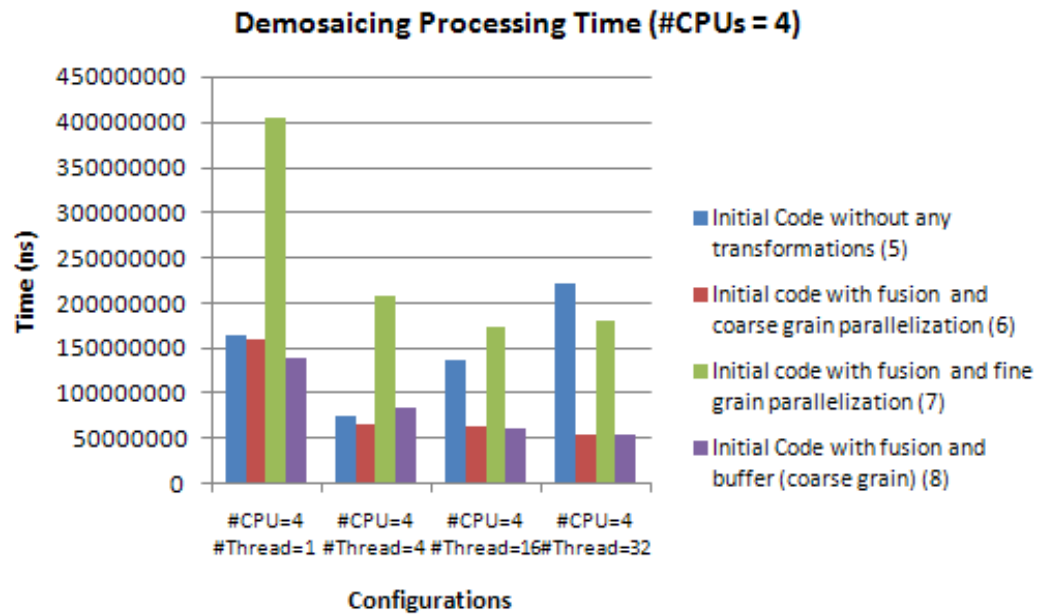


Figure 3.37: Processing time (ns) #CPUs=4

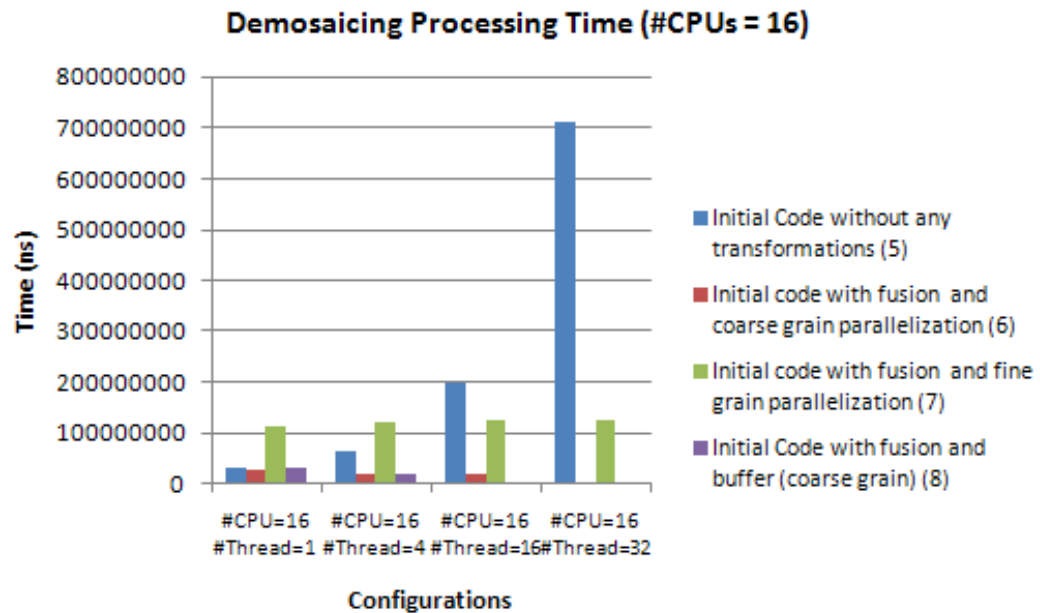


Figure 3.38: Processing time (ns) #CPUs=16

Figure 3.39 demonstrates the effect of increasing the number of CPUs on the processing.

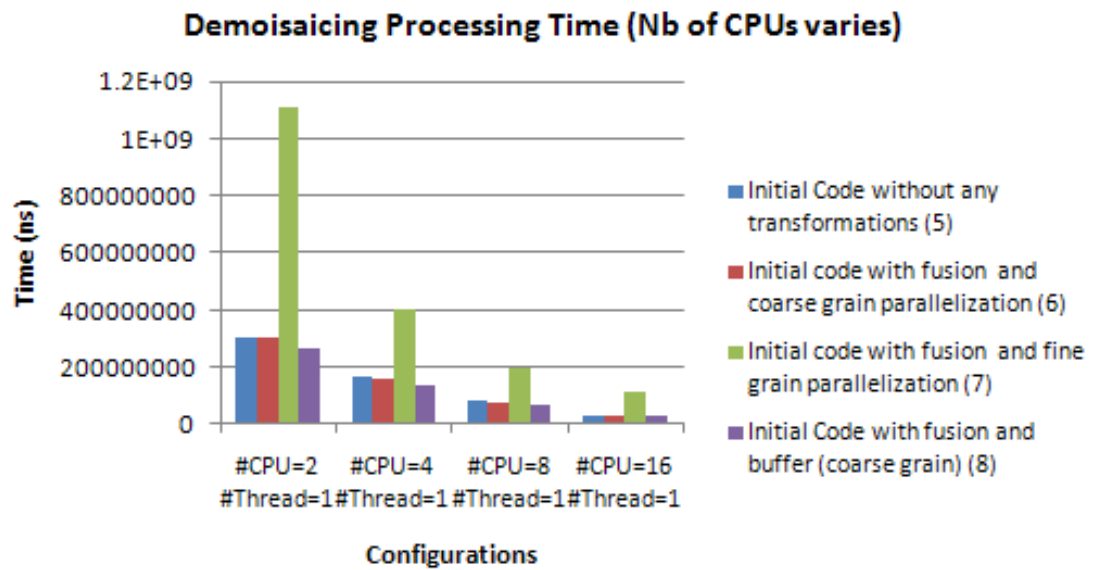


Figure 3.39: Processing time (ns) #CPUs varies

3.9 Analysis summary

Here are some extracted observations from the study of the effect of multithreading on the memory optimization techniques:

- Multithreading has a better effect on fine grain parallelization than on coarse grain parallelization.
- Too many threads available can increase processing time, since too much time will be spent managing threads.
- Having a large number of threads or processors, reduces the size of the data block being processed, therefore making it fit better in the cache and consequently increasing the cache hit ratio.

- Increasing the number of threads decreases the ratio of cache hit, since threads share the same memory. In a multiprocessor environment, every processor has its own memory (cache), therefore reducing the interference between each other.
- Increasing the number of processors has shown no effect on the cache performance.
- Increasing the number of threads greater than two decreases significantly processing time (without over parallelizing).

Overall multithreading reduces the processing time; however it decreases the cache hit ratio. Therefore, multiprocessor demonstrates the best balance between cache hit ratio and processing time. These observations show and confirm that investigating on new memory management techniques for multithreaded environment would be of great interest.

3.10 Conclusions and future work

This paper presented an overview of techniques' adaptation to significantly reduce the processing time while increasing the data cache hit ratio for a multimedia application running on an MPSoC.

From the results presented, one can see that the best results are obtained with the fusion with buffer allocation using no modulo operators with a unimodular transformation. This technique displays excellent balance among data cache hit, processing time, memory space and code size. Using fusion and buffer allocation in a multiprocessor environment necessitates an initial phase for border dependencies between partitioned blocks. A different data partitioning was demonstrated for eliminating border dependencies. Using buffer allocation require modulo operator, which in MPSoC environment are processing intensive. A modulo operator elimination was presented to eliminate modulo operators; however this technique may in some cases increase code size. For this reason, we have presented the final adaptation which is the unimodular transformation. It decreases the code size lost by using the modulo operator elimination. With these adaptations, we demonstrate

a global approach to the problem of data locality in MPSoC, averaging a 20% cache hit increase and a 50% processing time decrease.

This paper mainly presented a study on the effects of multithreading on different optimization techniques. From the results presented above, it is more advantageous to have more processors than threads.

In the parallel field, fine grain parallelization may have some advantages on the coarse grain approach. A fine grain approach maximizes the parallelization of a given application and implementing fine grain parallelization is easier. In contrast, a fine grain approach restrains the utilization of certain compilation techniques. Loop fusion may still be applied to an application, since it can be applied before the parallelization. However, the buffer allocation technique is difficult to apply in a fine grain approach.

Future works will focus on automating these optimizations by using and adapting tools like SUIF [35] and CLooG[36] and integrate it into the design process while considering these observations. Special attention will be paid to cache coherency for the fine grain parallelization approach.

References

- [1] A. A. Jerraya, and W. Wayne, *Multiprocessor Systems-on-Chips*, Elsevier ed., United States of America: Morgan Kaufmann, 2005.
- [2] W. Wolf, "The future of multiprocessor systems-on-chips," *Design Automation Conference 2004*. pp. 681-685, 2004.
- [3] M. Haines, and W. Bohm, "An evaluation of software multithreading in a conventional," *Parallel and Distributed Processing, 1993. Proceedings of the Fifth IEEE Symposium on*. pp. 106-113, 1993.
- [4] F. Catthoor, F. Franssen, S. Wuytack *et al.*, "Global communication and memory optimizing transformations for low," *VLSI Signal Processing, VII, 1994., [Workshop on]*. pp. 178-187, 1994.

- [5] F. Catthoor, S. Wuytack, E. D. Greef *et al.*, *Custom Memory Management Methodology -- Exploration of Memory Organisation for Embedded Multimedia System Design*, Boston: Kluwer Academic Publishers, 1998.
- [6] M. E. Wolf, and M. S. Lam, "A data locality optimizing algorithm," *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. pp. 30-44, 1991.
- [7] P. G. Paulin, C. Pilkington, M. Langevin *et al.*, "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 7, pp. 667-680, 2006, 2006.
- [8] S. Carr, and K. Kennedy, "Scalar replacement in the presence of conditional control flow," *Software - Practice and Experience*, vol. 24, no. 1, pp. 51-77, 1994/01/, 1994.
- [9] E. D. Greef, "Storage Size Reduction for Multimedia Application. Ph.D. Thesis," Katholieke Universiteit, Leuven, 1998.
- [10] K. Olukotun, B. A. Nayfeh, L. Hammond *et al.*, "The case for a single chip multiprocessor," *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. pp. 2-11, 1996.
- [11] M. Cierniak, and W. Li, "Unifying data and control transformations for distributed shared-memory machines," *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. pp. 205-217, 1995.
- [12] A. Darte, "On the complexity of loop fusion," *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*. pp. 149-157, 1999.
- [13] K. Kennedy, "Fast greedy weighted fusion," *International Journal of Parallel Programming*, vol. 29, no. 5, pp. 463-91, 2001/10/, 2001.
- [14] A. Fraboulet, K. Kodary, and A. Mignotte, "Loop fusion for memory space optimization," *System Synthesis, 2001. Proceedings. The 14th International Symposium on*. pp. 95-100, 2001.
- [15] P. Marchal, F. Catthoor, and J. I. Gomez, "Optimizing the memory bandwidth with loop fusion," *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*. pp. 188-193, 2004.

- [16] M. Kandemir, I. Kadayif, A. Choudhary *et al.*, "Optimizing inter-processor data locality," *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*. pp. 127-135, 2002.
- [17] M. Kandemir, "Data space oriented tiling," *Programming Languages and Systems. 11th European Symposium on Programming, ESOP 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002. Proceedings (Lecture Notes in Computer Science Vol.2305)*. pp. 178-93, 2002.
- [18] F. Li, and M. Kandemir, "Locality-conscious workload assignment for array-based computations in MPSOC architectures," *Design Automation Conference, 2005. Proceedings. 42nd*. pp. 95-100, 2005.
- [19] V. Krishnan, and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *Computers, IEEE Transactions on*, vol. 48, no. 9, pp. 866-880, 1999, 1999.
- [20] T. Van Achteren, G. Deconinck, F. Catthoor *et al.*, "Data reuse exploration techniques for loop-dominated applications," *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*. pp. 428-435, 2002.
- [21] I. Ilya, B. Erik, M. Miguel *et al.*, "DRDU: A data reuse analysis technique for efficient scratch-pad memory management," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 2, pp. 15, 2007.
- [22] C. Ghez, M. Miranda, A. Vandecappelle *et al.*, "Systematic high-level address code transformations for piece-wise linear indexing: illustration on a medical imaging algorithm," *Signal Processing Systems, 2000. SiPS 2000. 2000 IEEE Workshop on*. pp. 603-612, 2000.
- [23] F. Catthoor, K. Danckaert, K. K. Kulkarni *et al.*, *Data Access and Storage Management for Embedded Programmable Processors*, p. 324: Springer, 2002.
- [24] P. Schaumont, B.-C. C. Lai, W. Qin *et al.*, "Cooperative multithreading on embedded multiprocessor architectures enables energy-scalable design," *Design Automation Conference, 2005. Proceedings. 42nd*. pp. 27-30, 2005.
- [25] Y.-K. Chong, and K. Hwang, "Performance analysis of four memory consistency models for," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 6, no. 10, pp. 1085-1099, 1995, 1995.
- [26] G. Dimitroulakos, M. D. Galanis, and C. E. Goutis, "Performance improvements using coarse-grain reconfigurable logic in embedded SOCs," *Field Programmable Logic and Applications, 2005. International Conference on*. pp. 630-635, 2005.

- [27] B. M. Al-Hashimi, *System-on-Chip: Next Generation Electronics*: IEE, 2006.
- [28] M. J. Forsell, "Step caches - a novel approach to concurrent memory access on shared memory MP-SOCs," *NORCHIP Conference, 2005. 23rd.* pp. 74-77, 2005.
- [29] Y. Bouchebaba, and F. Coelho, "Tiling and memory reuse for sequences of nested loops," *Euro-Par 2002 Parallel Processing. 8th International Euro-Par Conference. Proceedings (Lecture Notes in Computer Science Vol.2400)*. pp. 255-64, 2002.
- [30] Y. Bouchebaba, B. Girodias, G. Nicolescu *et al.*, "MPSoC memory optimization using program transformation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 4, pp. 43, 2007.
- [31] Y. Bouchebaba, B. Lavigueur, B. Girodias *et al.*, "MPSoC memory optimization for digital camera applications: Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on," *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on.* pp. 424-427, 2007.
- [32] B. Girodias, Y. Bouchebaba, G. Nicolescu *et al.*, "Application-Level Memory Optimization for MPSoC," *Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on.* pp. 169-178, 2006.
- [33] H. Kwak, B. Lee, A. R. Hurson *et al.*, "Effects of multithreading on cache performance," *Computers, IEEE Transactions on*, vol. 48, no. 2, pp. 176-184, 1999, 1999.
- [34] R. Atitallah, S. Niar, A. Greiner *et al.*, "Estimating Energy Consumption for an MPSoC Architectural Exploration," *Architecture of Computing Systems - ARCS 2006*, pp. 298-310, 2006.
- [35] "SUIF, <http://suif.stanford.edu>," November 2006.
- [36] "CLooG" <http://www.prism.uvsq.fr/~cedb/bastools/cloog.html>.

CHAPITRE 4. ARTICLE 2: INTEGRATING MEMORY OPTIMIZATION WITH MAPPING ALGORITHMS FOR MULTI-PROCESSORS SYSTEM-ON-CHIP

B. Girodias¹, L. Gheorghe¹, Y. Bouchebaba¹, G. Nicolescu¹,
E.M. Aboulhamid², M. Langevin³, P. Paulin³

¹École Polytechnique de Montréal, ²Université de Montréal, ³STMicroelectronics

Abstract

Multiprocessor systems-on-chip (MPSoC) are defined as one of the main drivers of the industrial semiconductors revolution. MPSoC are gaining popularity in the field of embedded systems. Pursuant to their great ability to parallelize at a very high integration level, they are good candidates for systems and applications such as multimedia. Memory is becoming a key player for significant improvements in these applications (power, performance and area). With the emergence of more embedded multimedia applications in the industry, this issue becomes increasingly vital. The large amount of data manipulated by these applications requires high-capacity calculation and memory. Lately, new programming models have been introduced. These programming models offer higher level programming to answer the increasing needs of MPSoCs. This leads to the need of new optimization and mapping techniques suitable for embedded systems and their programming models. This paper presents novel approaches for combining memory optimization with mapping of data-driven applications while considering anti-dependence conflicts. Two different approaches are studied and integrated with existing mapping algorithms. Some significant improvements are obtained for memory gain, communication load and physical links.

4.1 Introduction

The International Technology Roadmap for Semiconductors (ITRS) defines Multiprocessor Systems-on-Chips (MPSoC) as one of the main drivers of the semiconductor industry revolution by enabling the integration of complex functionality on a single chip. MPSoC are increasing in popularity in today's high performance embedded systems. Given their combination of parallel data processing in a multiprocessor system with the high level of integration of System-on-Chip (SoC), they are great candidates for systems such as network processors and complex multimedia platforms [9]. Thus, in many cases, MPSoC platforms have to provide multimedia enhancements. For flexibility, software solutions are required. This implies the programming of multimedia applications with multi-dimensional streams of signals such as images and videos. In these applications, the majority of the area and power cost is due to the global communication and memory interactions [2,5]. In addition, time-to-market considerations mean that the platform must come with high-level application-to-platform mapping tools that increase developer productivity.

The design of embedded software is an important part of the MPSoC design. It is a concept not completely familiar to chip designers who are used to working solely on design hardware problems. Previously, they would not pay much attention to the application programming. In an MPSoC design, the combination of hardware and software designs can be a solution to solve these problems. For software designers, the hardware is often completely abstract. The software to be embedded in the MPSoC must be extremely reliable and must meet stringent constraints from the hardware such as timing constraints and energy consumption. For hardware designers, mapping an application has become a main concern as it is strongly related to the software domain. The consolidation of these two disciplines is one of the things that make the design of MPSoC both interesting and difficult. Techniques must be developed to facilitate the task of hardware and software designers for the adequate optimization and mapping of applications[1].

More and more sophisticated mapping tools and methodologies exist, but the field still lacks the tools which focus more specifically on memory optimization.

Multimedia applications often consist of multiple nested loops and are data driven applications. Therefore, it is possible to assume potential memory optimizations. Several optimization techniques such as fusion and buffer allocation exist and have proven themselves in single processor architectures. Our previous research presented some of these techniques and their impact on a multiprocessor environment. By analyzing applications and these techniques, significant performance improvements were proposed for these techniques applied to a multiprocessor environment [2]. These enhancements include memory space optimization, reducing the number of cache misses and improving execution time. The initial results obtained reduce memory space by 80%, increase the data cache hit rate by 20% and decrease the execution time by 50% [2].

Another motivation comes from some IMEC's researchers [3] who stated that memory accesses can account for 70% of the global energy consumption. Therefore, memory optimization indirectly reduces the energy consumption.

Driven by these positive results, we propose approaches to combine memory optimization with mapping. The contribution of this paper is the discussion of new application level approaches to guided mapping for memory optimization in MPSoC design. This paper proposes a methodology for combining memory optimization with mapping of streaming applications. The first approach (based on heuristic algorithms) keeps the memory optimization stage separate from the mapping stage and enables their combination in a design flow. The second approach (based on evolutionary algorithms) combines these two stages and integrates them in a unique stage. Since fusion and buffer allocation are software optimization techniques widely used in the multimedia industry, these techniques will be the main focus in this paper.

The remainder of this paper is organized as follows: section 4.2 discusses related work, section 4.3 overviews memory optimization techniques and mapping on MPSoC, section 4.4 describes the design methodology, sections 4.5 and 4.6 introduce the novel approaches, section 4.7 presents some experimentation, section 4.8 gives a small discussion and section 4.9 states the concluding remarks.

4.2 Related Work

In single processor environments (e.g. SoC), there has been extensive research by which several techniques and strategies have been proposed to improve cache data locality [4, 5]. Among them, one can point out scalar replacement [6][3], intra array storage order optimization [7], pre-fetching [8] and locality optimizations for array-based codes [9, 10]. One of IMEC groups [3][8] pioneered code transformation to reduce the energy consumption in data dominated embedded applications.

The work presented in this paper is different from these prior efforts, as it targets MPSoC with emphasis on mapping strategies. Our previous work demonstrated some new techniques and improvements on the different techniques presented above [2] in a MPSoC Symetric MultiProcessor (SMP) environment. However, this paper presents new approaches for combining memory optimization techniques with mapping in a MPSoC streaming environment.

In earlier days, mapping algorithms concentrated solely on optimizing performance in terms of computation power. Forced by new constraints like energy consumption and memory space brought by new architectures like MPSoC, today's mapping algorithms try to optimize new objectives[11]. In [12], Guangyu *et al.* present a mapping approach with the main objective of placing a given data item into a node which is close to the nodes that access it. In [13], Pastricha *et al.* present a mapping approach to design a communication architecture having the least number of busses and reduce memory area cost. Our approach is different as we focus on memory optimization techniques and place potential fusionable

tasks on the same nodes to reduce memory data used by task. Consequently, our approach reduces memory cost area and some communication cost between nodes.

Similar approaches to mapping problems are presented in [14], [15] and [16]. To facilitate the mapping process, new languages are introduced. Brook is an extension of standard ANSI C and is designed to incorporate the idea of parallel computing and arithmetic intensity into a well known language [17]. StreamIt is an independent language designed to facilitate the programming of large-scale streaming applications, as well as their mapping onto various architectures [18].

In [11], Hu et al. introduce the reader to a novel energy-aware scheduling algorithm for a heterogeneous network-on-chip (NoC). This algorithm automatically assigns tasks onto different processing elements (PE) and then schedules their execution under real-time constraints. The authors give a great overview of the difference between a network topology of multiple computers and of multiple cores as in NoC. In [19], Ruggiero et al. present a novel framework for allocation and scheduling task graphs on MPSoCs with communication awareness. The authors combine Integer Linear Programming (ILP) and Constraint Programming (CP) although ILP is only suitable for small problems. To confront this issue, they concentrate on statically scheduled systems. The mapping problem is decomposed into two sub-problems: (1) mapping of the tasks to PE and (2) scheduling of task in time on their execution units. The authors also introduce the concept of stream application. In [20] and [21], a heuristic approach and tool for mapping and scheduling problems using genetic algorithms is presented. Heuristic approaches are widely used and do not guarantee an optimal solution, however they will find an acceptable solution in reasonable time. Thiele L *et al.* [22] present a framework called Distributed Operation Layer (DOL). This framework offers system-level performance analysis and multi-objective algorithm-architecture mapping. The mapping algorithm is based on evolutionary algorithms and tries to optimize the computation time and communication time.

All the above presented contributions concentrate on using mapping for scheduling or power consumption optimization and little on memory optimization. In comparison, this paper focuses on optimizing memory space and size while taking knowledge from these past studies for the regular mapping of standard objectives (load and communication cost) and offers a different approach to memory optimization.

In [23], Fraboulet et al. present an optimal algorithm to reduce the use of temporary arrays by loop fusion in loop dominated applications. Our work is based on principles found in this article. This article presents algorithms to detect and remove conflicting fusion. Our work uses the algorithm to detect conflicts, however presents a completely different approach to avoid conflicts.

4.3 Memory Optimization and Mapping

This section will overview some techniques used in this work. In memory optimization, many techniques exist but two techniques are the focus of the optimization presented further in this paper. These techniques apply to any sequence of nested loops where each loop most depends only on previous nested loops. This section also overviews the concept of mapping.

4.3.1 Memory optimization

4.3.1.1 Loop fusion

Loop fusion [23] is often used in applications with numerous nested loops . This technique replaces multiple nested loops by a single one. It is widely used in compilation optimization since it increases data locality in a program. It enables data already present in the cache to be used immediately.

<pre> L1: do i = 0, N do j = 0, M S1: A(i,j) = F(In); end end L2: do i = 0, N do j = 0, M S2: B(i,j) = A(i,j) + A(i-1,j-1); end end </pre> (a) Initial code	<pre> L1,2: do i = 0, N do j = 0, M S1: A(i,j) = F(In); S2: B(i,j) = A(i,j) + A(i-1,j-1); end end </pre> (b) Loop Fusion
---	---

Figure 4.1 An example of 2 nested loops

Figure 4.1 illustrates the sequence of loop transformations going from the initial code to a loop fusion. In a multiprocessor environment, fusion is mainly applied to increase data locality. However, in a mapping application, fusion may be used to reduce the number of tasks on each processor. This would therefore reduce the memory space and the scheduling overhead of each task on each PE after the mapping. In a streaming application, fusion can also be applied to decrease the granularity of a stream for a better fit on a given target and to improve the load balancing [16].

4.3.1.2 Buffer allocation

Buffer allocation[24] is utilized for applications with temporary arrays which store intermediate computations. Buffer allocation is a technique which reduces the size of temporary arrays. It decreases memory space and reduces the cache miss ratio. The buffer size is defined by the dependencies among statements. The buffer will contain only useful elements (also called live elements). Figure 4.2 illustrates an example of the buffer allocation technique where array A is replaced by the buffer BUF of size 10.

In a multiprocessor environment, buffer allocation is mainly applied to generally decrease memory space. However, in a mapping application, buffer allocation may be used to reduce the transfer of data between tasks and therefore between PEs. For different

architectures and different applications, this may influence the size of the local memory, the type of interconnection and the size of the FIFO in streaming applications.

<pre> do i = 0, 7 do j = 0, 7 S1: A(i,j) = F(ln); S2: B(i,j) = A(i,j) + A(i-1, j-1); end end </pre>	(a) Initial code
<pre> do i = 0, 7 do j = 0, 7 S1: BUF[(8*i+j)%10] = F(ln); S2: B(i,j) = BUF[(8*i+j)%10] + BUF[(8*(i-1)+(j-1))%10]; end end </pre>	(b) Buffer Allocation

Figure 4.2 An example of buffer allocation

4.3.1.3 Anti-dependence

In an application, not all loops can be merged and arrays replaced by buffer allocation. Fraboulet et al. [23] present techniques to identify removable arrays and to detect conflicting removable arrays.

Two nodes can be merged if and only if none of the dependence is reversed in the fuse loop compared to the original code. An edge that carries a dependency which prevents the fusion of its source node and its destination node is called anti-dependence. In order to replace an array by a buffer, one needs to fuse all the nodes sharing this array [23].

Problems arise when several potential removable arrays are located on a cycle of the graph while part of this cycle contains a fusion preventing edge (FPE). Fraboulet et al. [23] state the following proposition:

In order to solve all possible problems that would create an illegal fusion we need to detect all undirected elementary cycles μ of the graph such as $\vec{\mu}$ is composed in the following way:

- all +1 (resp. -1) are fusionable
- at least one -1 (resp. +1) is a FPE

Our work is based on this proposition as well as the principle where all nodes sharing memory need to be fused together.

4.3.1.4 Mapping

The mapping problem consists of placing a given application to a particular architecture. This problem is considered an NP-hard problem. This article focuses more on data-driven mapping, since the main interest is memory optimization. The data between nodes (tasks) are analyzed and some transformations are applied to facilitate the mapping.

4.3.1.5 Memory Optimization and Mapping

Memory optimization techniques can always be applied after mapping. However, mapping can influence the capacity of being able to maximize the benefit of these optimizations. For example, in the case of buffer allocation, if all tasks sharing the same array are not located on the same PE after mapping, buffer allocation cannot be applied after the fusion of all tasks.

4.4 Design Methodology

Figure 4.3 illustrates the approaches design flow. The flow on the left (Figure 4.3 a) represents the heuristic approach and the flow on the right represents the evolutionary approach (Figure 4.3 b). These flow target large-scale multi-core platforms including a uniform layered communication network based on a Network-On-Chip backbone

infrastructure and a small number of hardware components for an efficient data transfer (e.g. stream-oriented DMAs or message passing accelerators). It targets data-driven applications on platforms with memory limitation issues.

The flow on the left (Figure 4.3 a) is called a heuristic approach because it uses a heuristic algorithm to transform the task graph. This transformation modifies the task graph to increase the number of possible memory optimizations after the mapping stage. This transformation is then sent to classic mapping algorithms which optimize typical

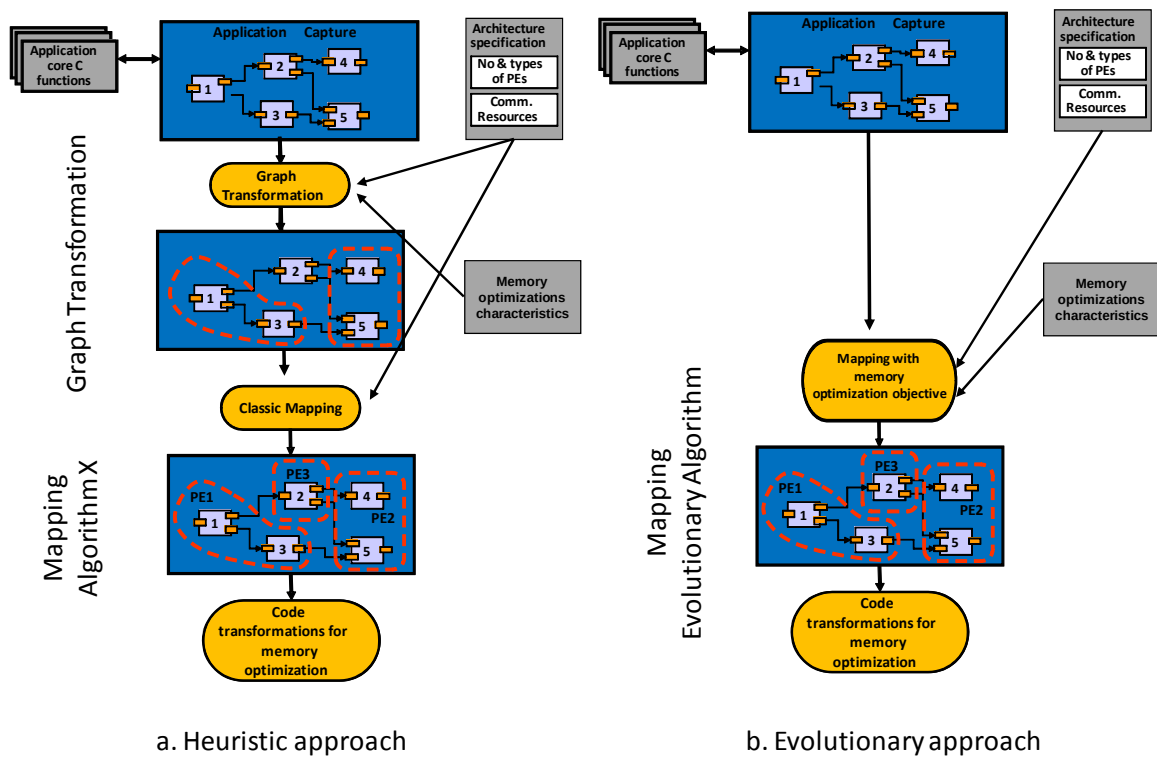


Figure 4.3 Design flow

performance characteristics like communication and load cost. These mapping algorithms can be heuristic or evolutionary algorithms.

The flow on the right (Figure 4.3 b) is called an evolutionary approach because it only uses evolutionary algorithms. In this flow, there is no graph transformation. An evolutionary algorithm is used for the mapping and memory optimization is added as an objective for the mapping.

4.4.1 Application Capture and Architecture Specification

The input of the flow is given by the application capture and the high-level specification of the platform. The application is captured as a task graph annotated with application-specific information (for example, quality-of-service requirements, measured/estimated execution characteristics of the application, data I/O characteristics, etc.). For this approach, load, memory space, memory gain, memory identifier, mapping pre-assignment and communication cost are added to the task graph. Information about the architecture is taken into account. The abstract platform specification includes the main characteristics of the platform running the application. One will find information about the number of PEs, maximum number of channels, maximum memory size, communication cost and communication architecture type.

4.4.2 Application Capture Transformation

This stage is only present in the heuristic approach. The application capture is transformed in order to exploit efficiently the memory optimization advantages. Thus, the main role of this stage is to guide further mapping algorithms by grouping on the same processor the tasks with the required properties for the memory optimization transformations (i.e. sharing the same array), while respecting the load balancing and the communication constraints.

4.4.3 Application Mapping

The graph resulted from the Capture Transformation stage represents the input of the mapping stage for the heuristic approach. For the evolutionary approach, the initial capture is taken.

In our heuristic approach design flow, three Mapping algorithms are used:

1. The first algorithm is based on the energy-aware scheduling algorithm by [11]. The algorithm is composed of two steps: (1) the selection of the task to map and (2) the selection of the next PE to map on. The first step is based on a ready list. This list is composed of tasks in which all its inputs are ready to be processed. The task graph is processed horizontally. The second step consists of an algorithm minimizing an objective function. More precisely, this function concentrates on minimizing the load and the communication cost. This algorithm favors task scheduling.
2. The second algorithm is based on a critical path algorithm [11]. The algorithm is composed of the same two steps as the previous algorithm. This algorithm differs from the previous by the manner in which the next task to be mapped is selected. The next task is selected by depth-first search. The task graph is processed vertically. The second step is similar to the first algorithm. This algorithm favors tasks scheduling with deadline constraints.
3. The third algorithm is based on an evolution algorithm. The algorithm is typically composed of a genetic representation of the solution domain and a fitness function to evaluate the solution domain. The initial solution representing the mapping of the tasks is generated randomly. These solutions are sent to known metaheuristics: NSGA-II, SPEA2, PAES, OMOPSO, AbYSS, MOCcell, PESA-II [25]. The new set of solutions generated is then evaluated by a fitness function taking into account the load and the communication cost. This algorithm computes a set of Pareto-optimal points. Each point represents a

solution of the multi-objective function that is optimal in at least one direction when all the other directions are fixed. This algorithm favors a more driven mapping optimization solution where the user can favor load or communication optimization.

For the evolutionary approach, only the third algorithm is used. This mapping algorithm was extended to take into account memory optimization. More details are found in section 4.6.

4.4.4 Memory Optimization

After each task is assigned to a PE by a mapping algorithm, some memory optimizations like fusion and buffer allocation can be applied locally on each PE (see section 4.3). To name a few, one can use existing tools like PIPS[26], SUIF[27] or Chunky[28] after the graph transformation and mapping to apply these memory optimization techniques.

4.5 Graph Transformations for Memory Optimization

4.5.1 Problem Formulation

The input algorithm's graph is formulated as follows in Definition 1. This algorithm assumes that a partial application analysis has been done and some information about the memory and its dependencies (e.g anti-dependency: write after read) between nodes were added as annotation to the graph as shown in Figure 4.4. This analysis can be done with tools like Dadealus [29].

The proposed algorithm tries to transform the graph by regrouping some vertices while respecting some hardware constraints (e.g. channel limit: number of edges and load limit: weight of vertex). Transformations focus on maximizing memory optimization after mapping.

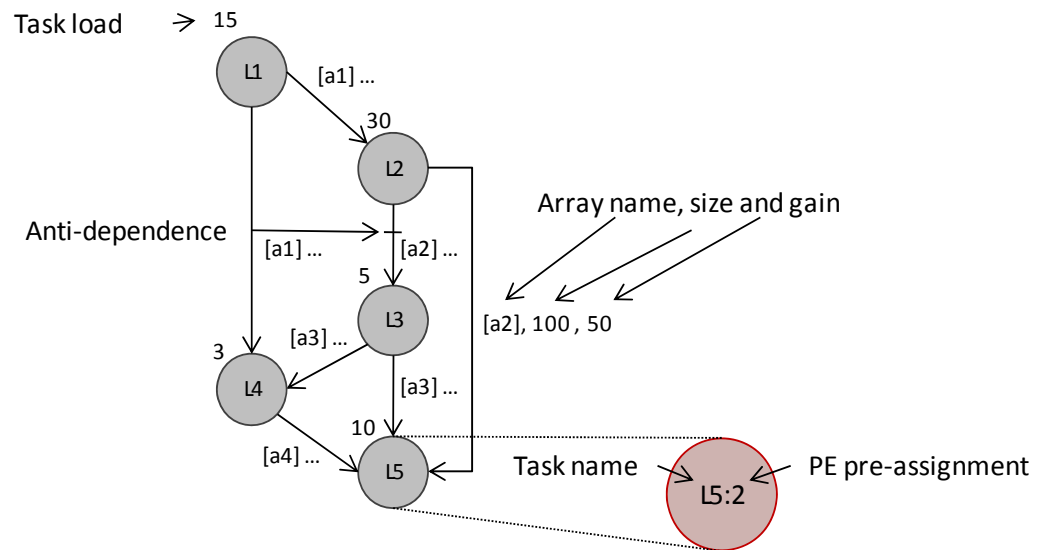


Figure 4.4 Input Graph Annotations

Using a simple example, Figure 4.4 and Figure 4.5 give an overview of the transformations performed to guide the application mapping for memory optimization. Figure 4.4 gives an example of a task graph annotation while Figure 4.5 gives the result after applying the transformation. We can notice that tasks L3, L4 and L5 were grouped (shadowed in Figure 4.5) in order to guide their further mapping on the same processor.

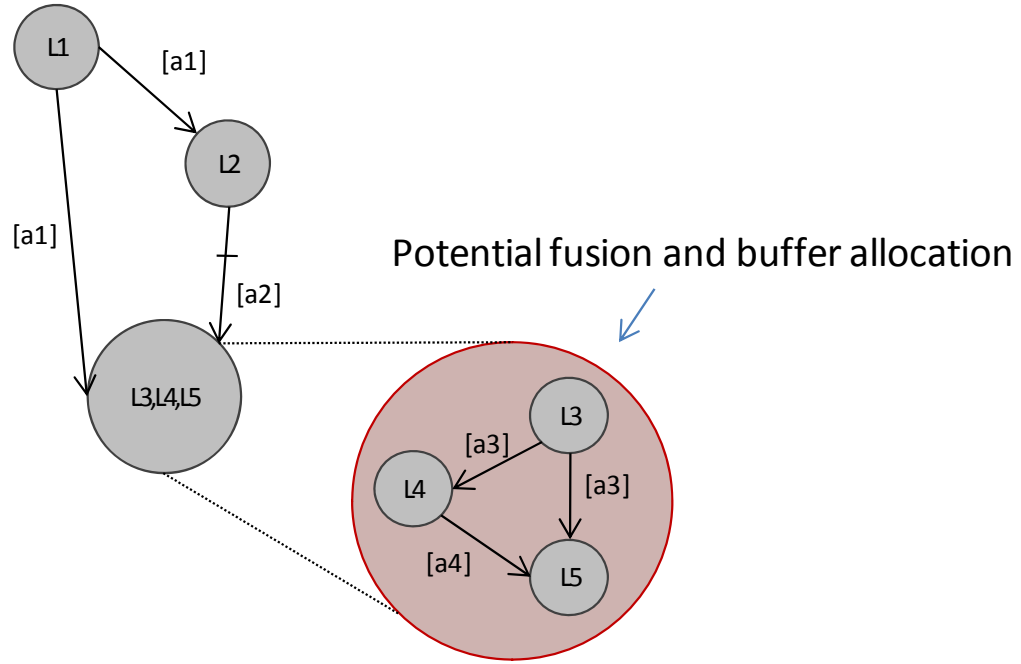


Figure 4.5 Output graph

In the following section, we give the formalism of the proposed transformation.

Definition 1: The input graph is a direct acyclic graph $\mathbf{TG}_{in}(\mathbf{T}, \mathbf{E}, \mathbf{A})$, a task graph with:

T – the non-empty set of vertices (tasks) t_i ,

E – the non-empty set of edges e_i

A - the non-empty set of arrays a_i

Each task $t_i \in T$ has the following annotations:

$load(t_i)$ that is the load of the task t_i , an evaluation of the amount of work,

$channel(t_i)$ that is the number of edges in TG incident with t_i

Part of the task may have the annotation:

$pe(t_k)$ the pre-assignment of the task t_k , to a processor (i.e. for some constraints, the designer gives the directives for mapping the task t_i on a given processor)

Each edge $e_i \in E$ is labeled with the following annotations:

a_i that is the name of an array transferred between the tasks linked by the edge e_i

$size(a_i)$ the memory size that represents the array a_i size between the tasks connected by e_i .

$gain(a_i)$ the memory gain that represents the improvement in memory if the tasks connected by e_i are merged and uses buffer allocation. The gain is the memory size minus the buffer size.

Definition 2: The output graph is a direct acyclic graph $TG_{out}(T, E, A)$, a task graph with:

T – the non-empty set of vertices (tasks) t_i or (super tasks) st_i ,

E – the non-empty set of edges e_i

A - the non-empty set of arrays a_i

A super task (st_i) is a group of tasks (t_i) with the property that these tasks can be fused together.

Each task t_i or $st_i \in T$ has the following annotations:

$load(t_i)$ or $load(st_i)$ that is the load of the task t_i , an evaluation of the amount of work,
 $channel(t_i)$ or $channel(st_i)$ that is the number of edges in TG incident with t_i

Part of the task may have the annotation:

$pe(t_k)$ or $pe(st_k)$ the pre-assignment of the task t_k or super task st_k , to a processor.

Each edge $e_i \in E$ is labeled with the following annotations:

a_i that is the name of an array transferred between the tasks (or super tasks) linked by the edge e_i

$size(a_i)$ the memory size that represents the array a_i size between the tasks (or super tasks) connected by e_i .

$gain(a_i)$ the memory gain that represents the improvement in memory if the tasks (or super tasks) connected by e_i are merged and use buffer allocation. The gain is the memory size minus the buffer size.

Definition 3: The transformation is defined as:

$TG_{out}(T, E, A) = f(TG_{in}(T, E, A), c_1, c_2 \dots c_k)$, where f is explicitly formulated in the next sub-section and $c_i, i \in [1, k]$ are hardware constraints (e.g. load limit, channel limit...).

4.5.2 Transformation Algorithm Formulation (f)

Step 1: The first step of the graph transformation consists of the detection of all potential removable arrays. In order to remove an array from an application, one needs to fuse all the nodes connected by an edge which is labeled by this array in the TG. In order to detect all

the arrays that could be removed by loop fusion, one must perform a transitive closure on the TG. An array cannot be removed by fusion if there exists an edge $e = (u, v)$ labeled by this array where exist a path from u to v in the TG containing a fusion preventing edge (FPE). FPE is an edge that carries a dependency which prevents the fusion of its source node and its destination node.[23].

Definition 4: Let $\mathbf{TG}_{FPE}$ be the set of sub-graph of TG , $TG_{FPE} = (T_{FPE}, E_{FPE}, A_{FPE})$ that is formed by the graphs that contain fusion preventing edge.

$\forall t_i, t_j \in T_{FPE}, \forall e_i \in E_{FPE}$ where $(t_i, t_j) = e_i$, e_i is a fusion preventing edge.

Step 2: The second step of the graph transformation consists of regrouping all tasks sharing the same array together. Hence, all tasks potentially capable of fusing together.

Definition 5: Let $\mathbf{TG}_k$ be a sub-graph of TG , $TG_k = (T_k, E_k, \{a_k\})$ with the property:

$$\forall t_i, t_j \in T_k, \exists e_i \in E_k \mid (t_i, t_j) = e_i, \wedge a_i = a_k, TG_k \subseteq TG \quad (1)$$

Definition 6: Let $\mathbf{TG}_{set1}$ be the set of sub-graphs of TG that verify the property (1).

Step 3: This step consists in the elimination of all the sub-graphs where the load capacity exceeds the average load and the channel limit given to us by information on the task graph and the targeted architecture.

Definition 7: Let $\mathbf{load}_{sgk}$ be the load of the sub-graph $TG_k \in TG_{set1}$ where

$$\mathbf{load}_{sgk} = \sum_i \mathbf{load}(t_i), \forall t_i \in T_k. \forall TG_k (T_k, E_k, \{a_k\}) \in TG_{set1} \quad (2)$$

Definition 8: Let $\mathbf{channel}_{sgk}$ be the number of channels (or incident edges) of the sub-graph $TG_k \in TG_{set1}$

$$\mathbf{channel}_{sgk} = |E_k|, \text{ where } E_k = \{e_k \mid e_k = (t_i, t_j) \wedge t_i \in TG_k \wedge t_j \in TG_{set1} - TG_k\} \quad (3)$$

Definition 9: Let $\mathbf{TG}_{elim1}$ be a set of sub-graphs of TG (and a sub-set of TG_{set1}) where the load limit or the channel limit of the sub-graph exceeds the limit values $\mathbf{limit}_{load}$ respectively $\mathbf{limit}_{channel}$.

$$TG_{elim1} = \{TG_k \in TG_{set1} \mid load_{sgk} > limit_{load} \vee channel_{sgk} > limit_{channel}\}, TG_{elim1} \subseteq TG_{set1} \quad (4)$$

In our context, we want to assure the load balancing of the PEs and not exceed the possible number of channels per PE. However, these constraints are flexible and the user can explore different architecture configurations as he wishes. Failing to consider load balancing in memory optimization may result in the mapping of all tasks on one PE for an optimal memory optimization.

Step 4: This step consists of the elimination of all sub-graphs that have tasks with multiple different mapping pre-assignments. In a potential fusionable group, if the tasks have different pre-assignments, they will not be mapped on the same PE. Hence, we will not be able to use buffer allocation.

Definition 10: Let $PE(TG_k)$ be the set of pre-assigned processors (PEs) for the tasks of the sub-graph.

$$TG_k. PE(TG_k) = \{pe(t_k) \mid t_k \in TG_k\} \quad (5)$$

Definition 11: Let TG_{elim2} be a set of sub-graphs of TG (and a sub-set of TG_{set1}) where two or more tasks in the sub-graphs included in this set are pre-assigned to different PEs.

$$TG_{elim2} = \{TG_k = (T_k, E_k, \{a_k\}) \in TG_{set1} \mid \forall k, |PE(TG_k) \cap PE(TG_{set1})| > 1\} \quad TG_{elim2} \subseteq TG_{set1} \quad (6)$$

After the first two iterations, the sets of sub-graphs that are going to be eliminated are the sets TG_{elim1} and TG_{elim2} .

Definition 12: Let be TG_{set2} the set of sub-graphs from TG that is the result of the elimination of exceeding load and channel and incompatible pre-assignments.

$$TG_{set2} = TG_{set1} - TG_{elim1} - TG_{elim2} - TG_{FPE} \quad (7)$$

Step 5: The step consists of the analysis of the set of sub-graphs where the tasks exist in multiple sub-graphs. These tasks are named duplicate tasks. This problem becomes a

combinatorial optimization. For the sub-graphs with duplicated tasks, the algorithm will try to see if it is possible to regroup all the conflicting groups together.

Definition 13: Let $\mathbf{TG}_{\text{duplicata}}$ be the set of sub-graphs of $\mathbf{TG}_{\text{set2}}$

$$\forall TG_i \in TG_{\text{set2}}, \exists TG_j \in TG_{\text{set2}} \mid T_i \cap T_j \neq \Phi \quad (8)$$

Step 6: After the generation of sets of duplicate, the algorithm selects the combination of sub-graphs ($\mathbf{TG}_{\text{set2}}$) that maximizes the memory gain while not exceeding the limit on the load, on the channel and avoiding any fusion conflicts.

Definition 14: Let $\mathbf{TG}_{\text{comb}}$ be the set of potentially fusionable sub-graphs that are combinations of the sub-graphs in $\mathbf{TG}_{\text{duplicata}}$.

$$\mathbf{TG}_{\text{comb}} = \{ TG_i, TG_j, TG_k = TG_i + (TG_i + TG_j) \mid \forall TG_i, TG_j \in \mathbf{TG}_{\text{duplicata}} \wedge TG_i \cap TG_j \neq \Phi \} \quad (9)$$

Definition 15: Let $\mathbf{TG}_{\text{elim3}}$ be the set of sub-graphs of $\mathbf{TG}_{\text{comb1}}$ where the load limit or the channel limit of the sub-graph exceed the limit values $limit_{\text{load}}$ respectively $limit_{\text{channel}}$ or the combinations have assigned different processors within themselves.

Definition 16: Let $\mathbf{TG}_{\text{FPE1}}$ be a sub-graph of $\mathbf{TG}_{\text{comb}}$, $TG_{\text{FPE1}} = (T_{\text{FPE1}}, E_{\text{FPE1}}, A_{\text{FPE1}})$ with the property:

$$\forall t_i, t_j \in T_{\text{FPE1}}, \exists e_i \in E_{\text{FPE1}} \mid (t_i, t_j) = e_i \wedge \text{all} - 1 \ e_i \notin \{E_{\text{FPE1}}\} \text{ where } TG_{\text{FPE1}} \cap TG_{\text{FPE1}} \neq \Phi$$

Step 7: The algorithm eliminates the sub-graphs $\mathbf{TG}_{\text{elim3}}$ of the set $\mathbf{TG}_{\text{comb}}$ that exceed the limit on the load, on the channel or create multiple pre-assignments.

$$\mathbf{TG}_{\text{elim3}} = \{ TG_k \in \mathbf{TG}_{\text{comb1}} \mid \text{load}_{\text{sgk}} > \text{limit}_{\text{load}} \vee \text{channel}_{\text{sgk}} > \text{limit}_{\text{channel}} \vee (\text{PE}(TG_k) \cap \text{PE}(TG_{\text{set1}}) = T_{\text{diff}} \wedge |T_{\text{diff}}| > 1) \} \quad (10)$$

Definition 17: Let $\mathbf{TG}_{\text{comb1}}$ be the set of valid combinations of sub-graphs from that $\mathbf{TG}_{\text{comb}}$:

$$\mathbf{TG}_{\text{comb1}} = \mathbf{TG}_{\text{comb}} - \mathbf{TG}_{\text{FPE1}} - \mathbf{TG}_{\text{elim3}} \quad (11)$$

Step 8: The algorithm eliminates all sub-graphs that create conflicting fusion.

Definition 18: Let $\mathbf{TG}_{FPE2}$ be a sub-graph of TG_{comb1} , $TG_{FPE2} = (T_{FPE2}, E_{FPE2}, A_{FPE2})$ with the property:

$$\forall t_i, t_j \in T_{FPE2}, \exists e \in E_{FPE2} \mid (t_i, t_j) = e_i \wedge (t_j, t_i) = e_j \wedge e_i \neq e_j \wedge e \in \{e_i, e_j\} \text{ where } TG_{FPE2} \cap TG_{FPE} \neq \Phi$$

Definition 19: Let $\mathbf{gain}_{sgk}$ be the gain of the sub-graph $TG_k \in TG_{comb}$ where:

$$\mathbf{gain}_{sgk} = \sum_i \mathbf{gain}(a_i), \forall TG_k \in TG_{comb1}, \forall a_i \in A_k \quad (12)$$

Definition 20: Let $\mathbf{TG}_{comb2}$ be the set of combinations from that TG_{comb1} that maximize the memory gain

$$TG_{comb2} = \{TG_k \in TG_{comb1} \mid \max(\mathbf{gain}_{sgk})\} \quad (13)$$

Step 9: For the same memory gain the algorithm chooses the combination with the minimum load and if still equal with the minimum channel:

$$TG_{comb3} = \{TG_k \in TG_{comb2} \mid \min(\mathbf{load}_{sgk})\} \quad (14)$$

$$TG_{comb4} = \{TG_k \in TG_{comb3} \mid \min(\mathbf{channel}_{sgk})\} \quad (15)$$

Definition 21: Let $\mathbf{TG}_{set3}$ be the final set of sub-graphs from TG that is the result of the algorithm iteration

$$TG_{set3} = TG_{set2} - TG_{duplicata} - \mathbf{TG}_{FPE2} + TG_{comb4} \quad (16)$$

Each sub-graph from the set TG_{set3} will be represented as a new task in the new application task graph (TG_{out}). The new task graph is built with these groups (fusionable tasks) as well as with all the rest of the tasks (non fusionable tasks). The task graph information about communication cost and load of each tasks are updated.

This new graph is then sent to a mapping algorithm which will assign each task to a PE. With the mapping algorithm's results, we propagate the assignment of each group (super task) to the original application task graph.

It is to be noted that, for this article, when tasks are regrouped, the updated load of the group representing the load after a fusion is the worst case scenario where the load after the fusion is equal to the addition of all the load of each task. In future work, this will be improved by the use of an intermediate evaluation with tools like CLoog [28] or statistical analysis. Previous work [2], has demonstrated some significant gain on this aspect.

The complexity of this application is $O(N^2 2^M)$, where M is the maximum number of $\mathbf{TG}_{\text{duplicata}}$ sets of sub-graphs of $\mathbf{TG}_{\text{set2}}$. N is the highest number of tasks $\mathbf{TG}_{\text{duplicata}}$ contains. The more M increases the more N decreases and vice versa.

4.6 Evolution Algorithm for Memory Optimization

The approach presented in the previous section is integrated with only heuristic mapping algorithms. A mapping based on evolutionary algorithm was then introduced. With heuristic mapping algorithms, the approach presented in the previous section would set the limits of the solution's exploration domain by fixing a load threshold on each PE to assure good load balancing. This approach was reasonably limiting the exploration domain of the heuristic mapping algorithms. However, with a genetic algorithm, this heuristic approach over cuts the exploration domain, since genetic algorithms are known to be capable of a vast exploration. Therefore, we decided to complete our work by proposing an approach that allows to incorporate the concepts presented earlier in the genetic mapping algorithm rather than making our approach as an initial phase to the mapping.

From the initial task graph, after applying a transitive closure to detect all removable arrays, we identify all potential fusions in distinct groups. With each group, we generate all legal combinations that don't create any conflicting fusion. Meanwhile, the genetic mapping algorithm generated an initial random population. It evaluates the load and communication for each solution and then evaluates each solution with our memory objective function. Our memory objective function tries to minimize the total memory space of the platform:

$$\text{Mem}_{\text{TOTAL}} = \sum_{i=0}^{i=\text{nbPE}} \text{Mem}_{\text{SPACE}}(\text{PE}_i) - \text{Mem}_{\text{GAIN}}(\text{PE}_i) \quad (17)$$

where $\text{Mem}_{\text{SPACE}}$ is the memory space needed by tasks on each processor and Mem_{GAIN} is the memory gain based on the legal combination found by the mapping defined by the solution.

Each solution found is evaluated to verify if any hardware constraints are violated. In our cases, constraints are limitations on the number of channels and on the memory space on each PE.

For our evolutionary approach, we decided to extend the genetic representation (used in the mapping algorithm) with all the legal combinations possible containing potential fusion. By this, we are able to let the genetic algorithm decide if the memory transformation technique is applied or not. In the previous version, we were assuming that all possible fusions were automatically applied. In the case where the solution gives a choice of different valid combinations of fusion for a certain mapping, the objective function evaluated the combination that gives the greatest memory gain.

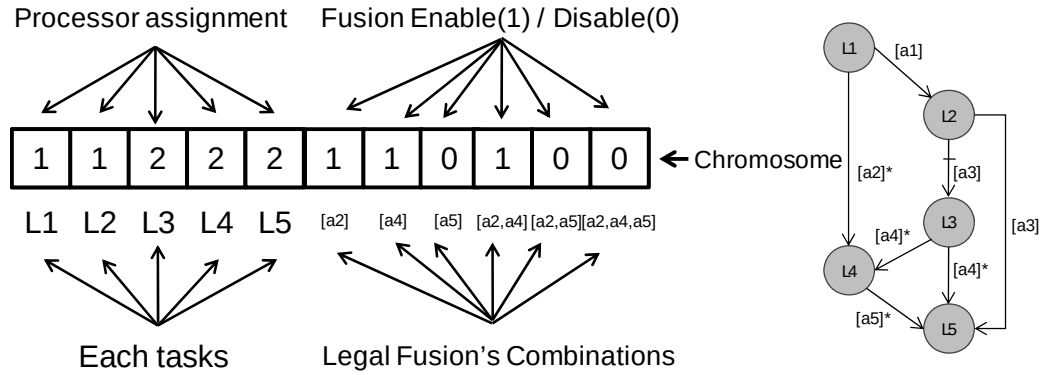


Figure 4.6 Genetic algorithm approach

In Figure 4.6, the tasks L1 and L2 are mapped on processor 1 and tasks L3, L4 and L5 are mapped on processor 2. This solution enables the combination of fusion [a2], [a4] and

[a2 and a4], however, since L1 and L4 are not on the same processor, the only combination left for this mapping is [a4] which is the fusion of L3-L4 and L3-L5.

We can imagine in a near future that the genetic algorithm would, not only be able to enable or disable a transformation, but also decide which order to apply the transformation, while respecting some priority constraints between each transformation. Choosing the useful program transformation as well as the order in which to apply them is a difficult task [30].

4.7 Experimental Results

To evaluate the efficiency of the proposed approaches, we integrated them into the MultiFlex environment [31]. MultiFlex is a platform for MPSoC design space exploration. It offers high-level, standardized platform programming models, a fast application to platform mapping and more regular, predictable architectures.

This article predominantly uses MultiFlex mapping tools. The mapping tool offers more appropriate algorithms using heuristic or evolutionary algorithms to assign blocks to processing units while optimizing certain parameters. It uses the three mapping algorithms presented in section 4.4. From an abstracted tasks graph model with an abstract description of the platform, it tries to assign tasks to multiple PEs while minimizing the communication and respecting load balancing.

All experiments were done on a 4, 8 and 16 PEs architecture. For the NoC architecture, we target the Spidergon topology[32], which is a Chordal ring where each router is connected with its left and right neighbors, as well as the opposite router in the ring. The three mapping algorithms, briefly presented earlier, were used for the mapping phase of the experiments.

Experiments concentrate on four metrics:

1. *Memory gain* - consists of the memory space gain by the fusion and replacement of the temporally array by smaller buffer. It is equal to the size of temporally array minus the size of the buffer used.
2. *Communication* - consists of the amount of data transfer between all the PEs. It is the load of the network connecting all the PEs.
3. *Processing load variance* - consists of the difference between the biggest PE load and the smallest PE load.
4. *Physical link* - consists of the number of links between the PEs.

Multiple task graphs were generated using TGFF [33]. They respect the characteristics of a multimedia application: a large amount of fusible tasks and a heavy data load transfer between tasks. Information on load, memory size and communication cost are generated randomly while considering the characteristics above.

In this section, we present the results of two applications. The first application consists of a generated task graph with 29 tasks (see Figure 4.7) with potential fusion and some anti-dependence. With this application, the heuristic approach was tested. The second application consists of a real-life application (i.e. Demosaicing Figure 4.12). Both approaches were tested with the Demosaicing application.

To be noted that the scalability of the approaches proposed in this article has a direct correlation with the size of the application and the number of processors. Therefore, using an application with a small task number and a platform with 16 processors reduces the possibility of merging tasks since a minimum of one task will be assigned to each processor, for a total of 16 tasks assigned. Once again, the application specifies the architecture selected.

4.7.1 Generated task graph with heuristic approach

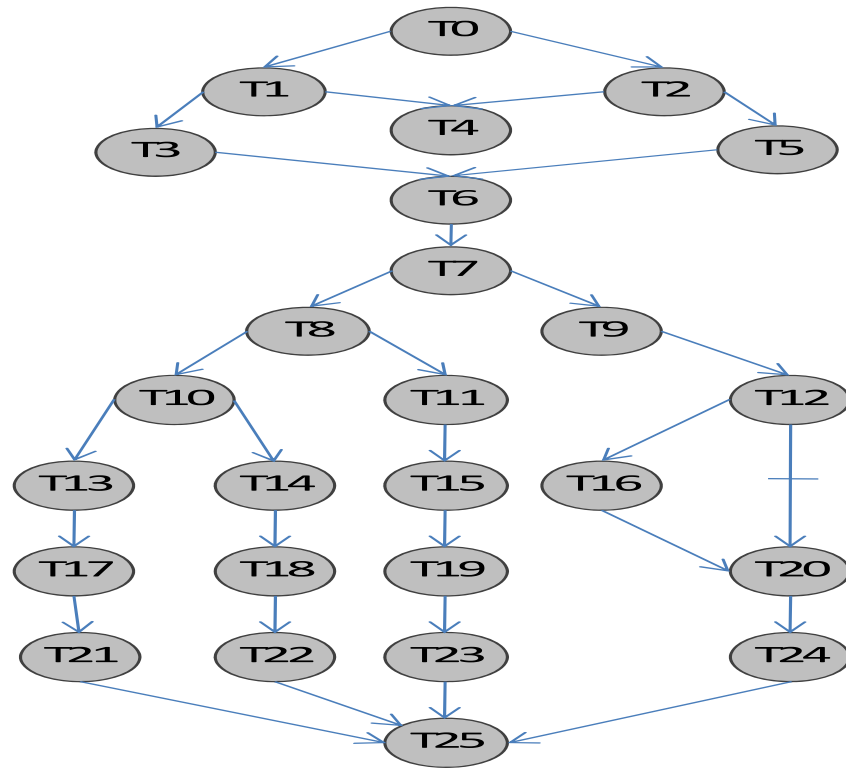


Figure 4.7 Application Task Graph

Figure 4.8 illustrates the memory gain obtained on different multiprocessor architectures (4, 8 and 16 PEs). A mapping algorithm optimizing communication may create some potential memory gain; however, one can see that our approach increases the memory gain. Respectively for a 4 and 8 PEs architecture, this approach increased the memory gain by 63% and 100%. For a 16 PEs architecture, the memory gain was decreased by 5% due to the following reason. Overall, results are influenced by the application complexity and fusionable group size, and therefore if there are too many PEs, it reduces the number of potential fusions since load balancing is always considered. This explains the difference in memory gain between the 8 PEs and 16 PEs cases. However, a smaller number of PEs may

seem to give lower results, since more memory will be distributed to each PE, the ratio of gain will be reduced.

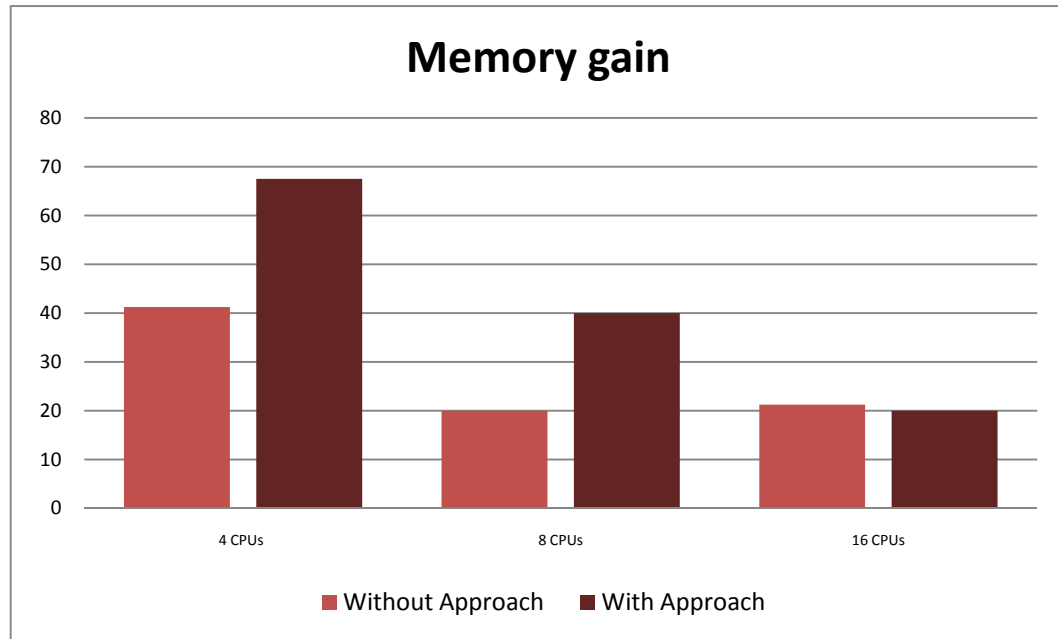


Figure 4.8 Memory Gain

Figure 4.9 illustrates the communication load obtained on different multiprocessors (4, 8 and 16 PEs). While the communication load was not the focus of our optimizing approach, by optimizing the memory, one will indirectly optimize the communication. Respectively for a 4, 8 and 16 PEs architecture, this approach decreases the communication load architectures by 16%, 9% and 0%. If we increase the number of PEs, memory gain decreases, therefore communication is less reduced.

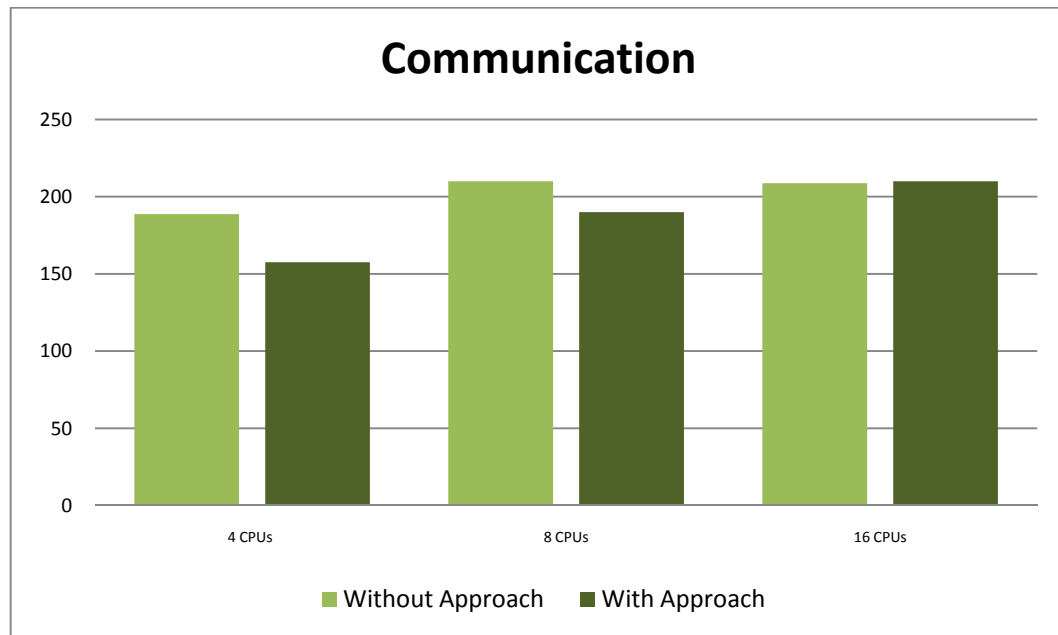


Figure 4.9 Communication

Figure 4.10 illustrates the processing load variance obtained on different multiprocessors architectures (4, 8 and 16 PEs). As discussed previously, this approach currently doesn't consider *load gain* by applying fusion and has considered no gain in load processing even if tasks are to be fused together. When tasks are to be fused together, the results shown under are the addition of each respective task. The results shown below are more conservative; however, regrouping tasks together will increase the load, applying fusion may reduce the load as seen in previous work[2]. Therefore, the load processing variance is increased by 51% and 15% for a 4 and 8 PEs architecture respectively. However, for a 16 PEs architecture the processing load variance is decreased by 17%. Regrouping tasks together will increase the load, however applying fusion may reduce the load as seen in previous work [2]. Future work will consider processing gain due to fusion.

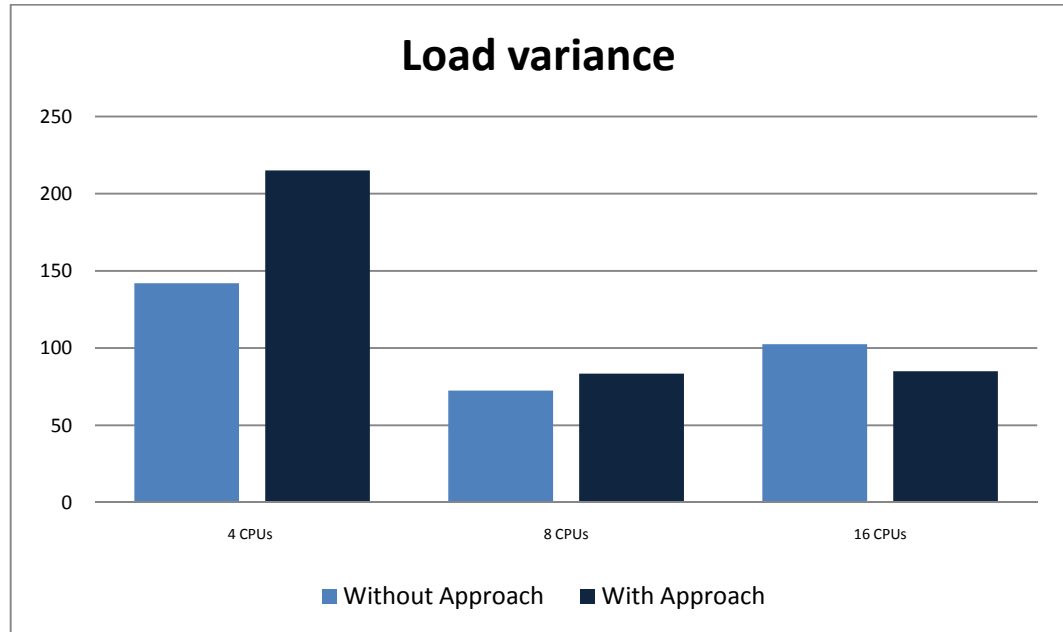


Figure 4.10 Load Variance

Figure 4.11 illustrates the number of physical links obtained on different multiprocessors architectures (4, 8 and 16 PEs). Similar to the communication load, optimizing the memory may reduce the number of links necessary between the nodes. Respectively for a 4, 8 and 16 PEs architecture, this approach decreases the number of links by 15%, 16% and 2%. The same remarks for the communication cost apply to the physical links.

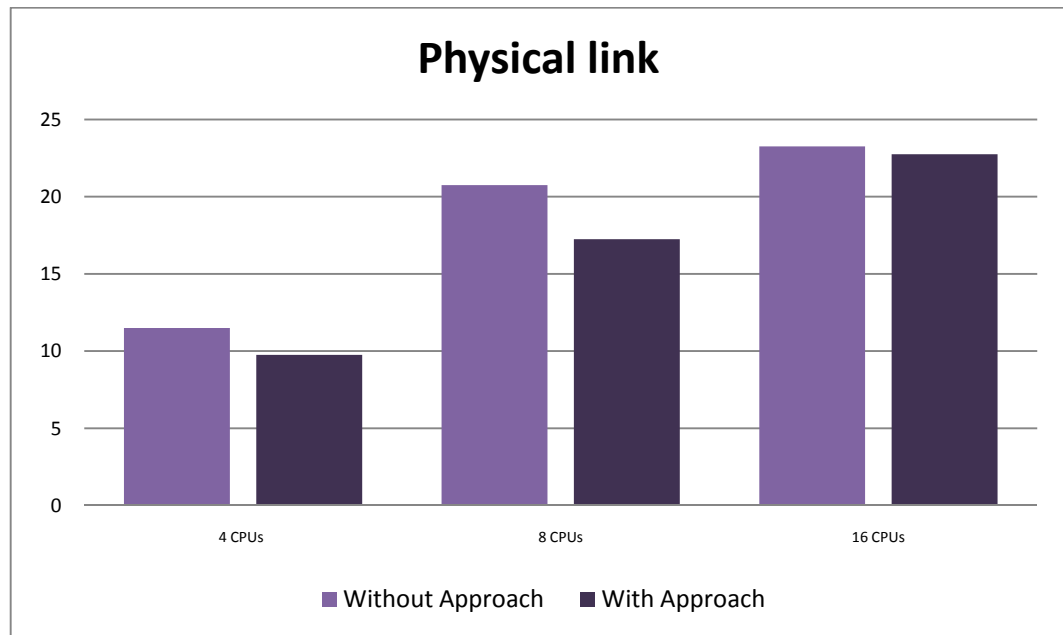


Figure 4.11 Physical Link

4.7.2 Demosaicing with heuristic approach

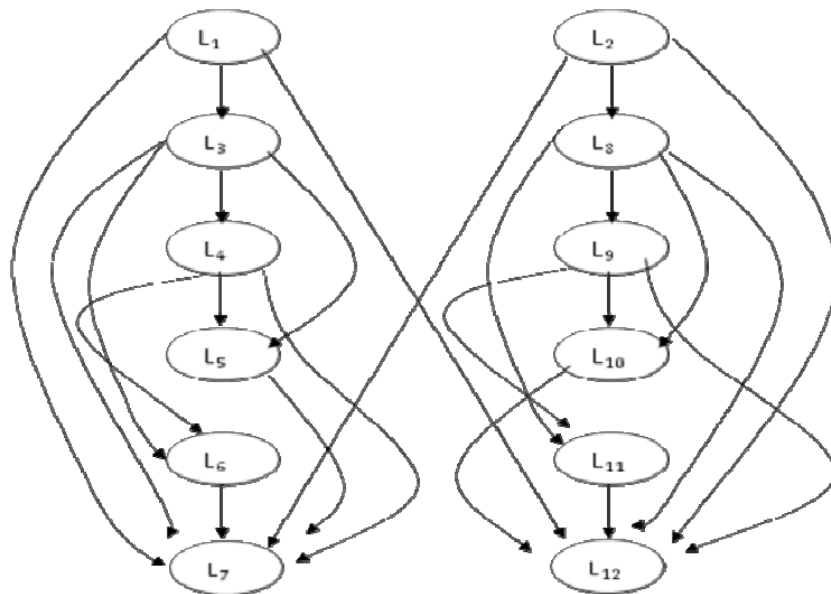


Figure 4.12 Demosaicing Application Task Graph

Figure 4.13 illustrates the memory gain obtained on different multiprocessors architectures (4, 8 and 16 PEs). Respectively for a 4, 8 and 16 PEs architecture, this approach increased the memory gain by 37%, 56% and 27%.

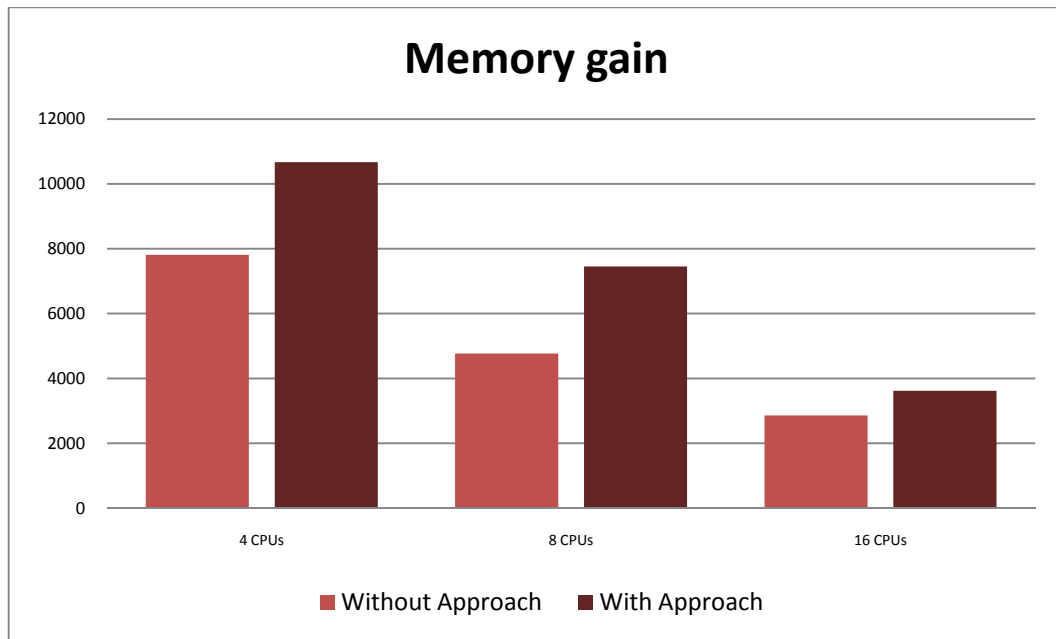


Figure 4.13 Memory Gain (Demosaiicing)

Figure 4.14 illustrates the communication load obtained on different multiprocessors architectures (4, 8 and 16 PEs). Respectively for a 4, 8 and 16 PEs architecture, this approach decreases the communication load by 27%, 8% and 4%.

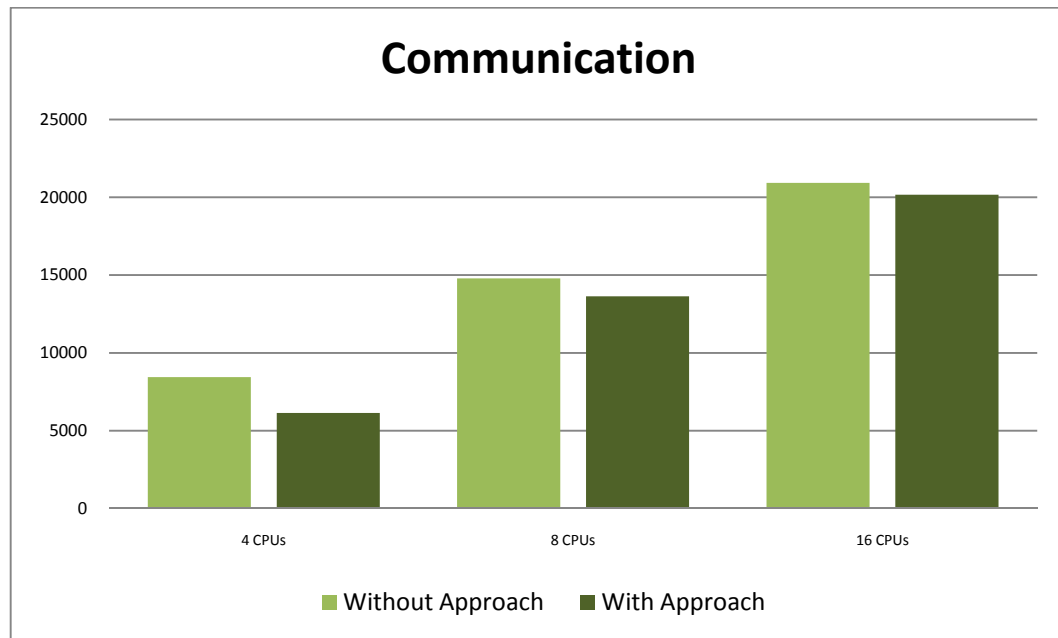


Figure 4.14 Communication (Demosaicing)

Figure 4.15 illustrates the processing load variance obtained on different multiprocessors architectures (4, 8 and 16 PEs). The load processing variance is maintained for the 4 CPUs architecture. It is increased by 20% for a 8 CPUs architecture and it is decreased by 8% for the 16 CPUs architecture. Once again, future work will consider processing gain due to fusion.

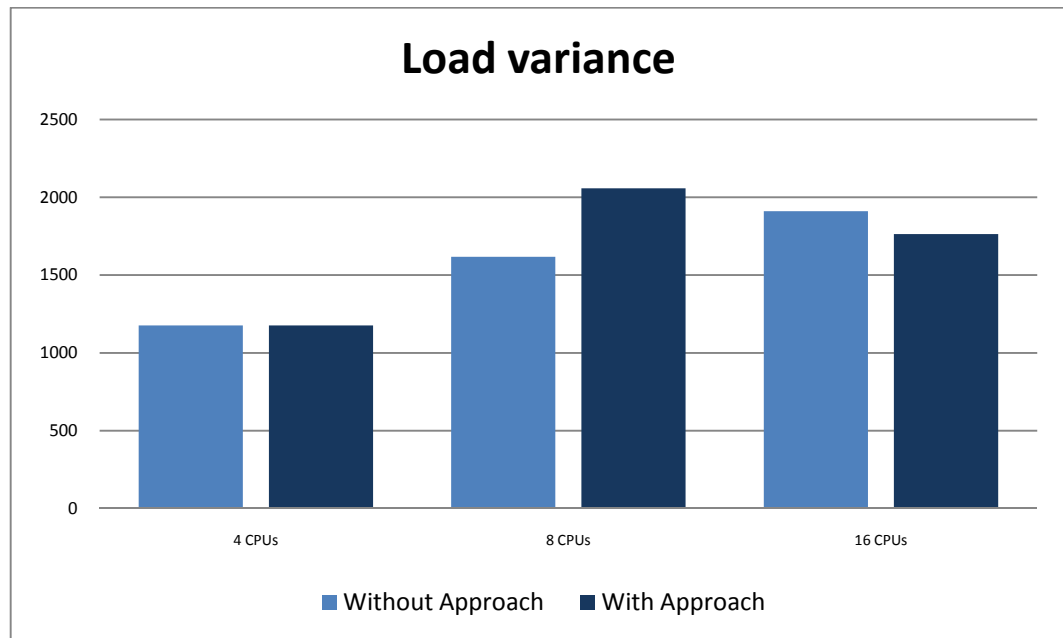


Figure 4.15 Load Variance (Demosaicing)

Figure 4.16 illustrates the number of physical links obtained on different multiprocessor architectures (4, 8 and 16 PEs). Respectively for a 4, 8 and 16 PEs architecture, this approach decreases the number of links by 27%, 22% and 6%.

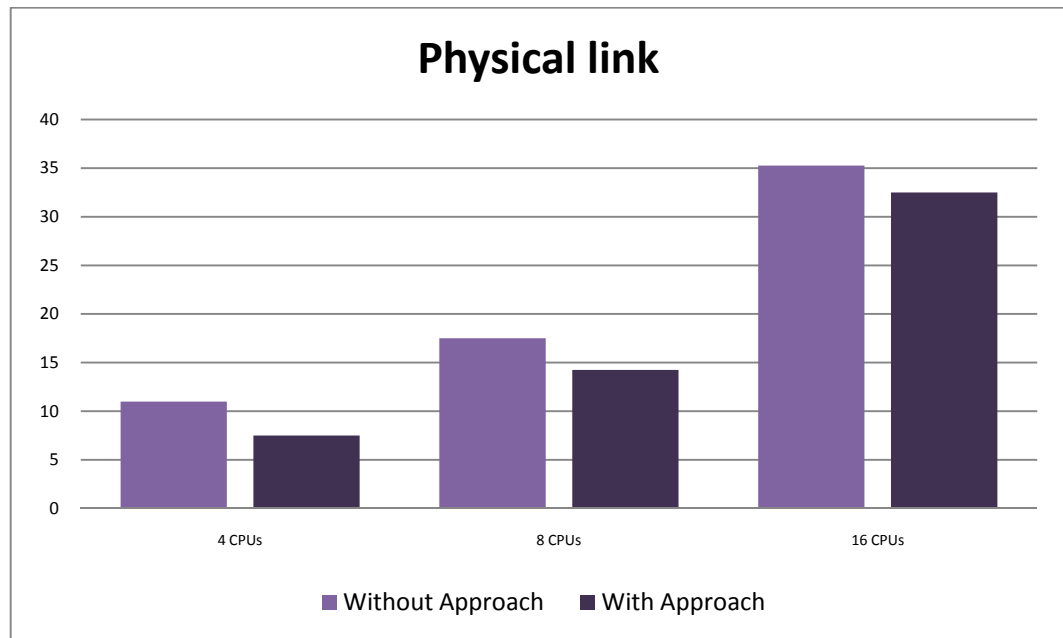


Figure 4.16 Physical Link (Demosaiicing)

Other than parameters like the number of potential fusions and the memory size which can influence greatly the positive results of our approach, one can see that the results obtained are influenced by the architecture type and the size of the application.

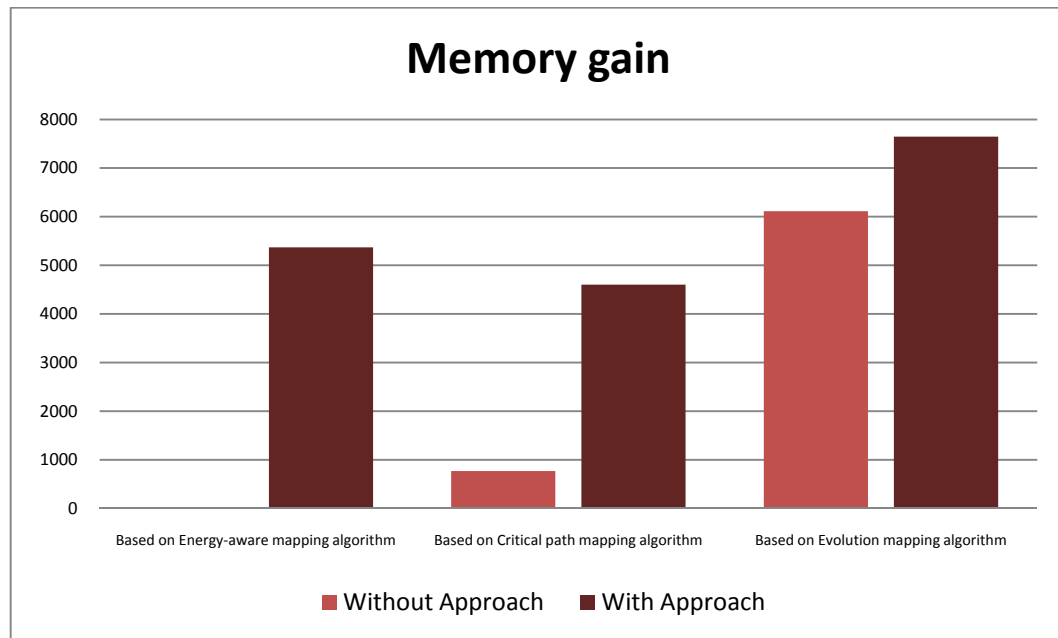


Figure 4.17 Brief comparison of the approach's effect on the different mapping algorithms

Not seen in this section is the influence of the different mapping algorithms and the type of application. The results presented earlier are the average of results obtained with the three different mapping algorithms. Included in this article, Figure 4.17 gives a brief comparison of the approach's effect on the different mapping algorithms discussed in section 4.4.3. Discussion about the influence and difference between each algorithm is beyond the scope of this paper as not only the mapping algorithm may be a factor but also the type of application used. Some experimentation has been done with different application types. However, no negative effect has been demonstrated with our approach. This paper concentrates on multimedia applications involving multi-dimensional streams of signals such as images and videos and therefore we have shown results only for multimedia application with intensive data transfer.

4.7.3 Demosaicing with evolutionary approach

We have evaluated the two versions of our approach with the application Demosaicing in the Multiflex's environment. The first approach is based on heuristic algorithm for memory optimization and genetic algorithm for the mapping. Our second approach is based on genetic algorithm for memory optimization and genetic algorithm as well for the mapping. For our experimentation, we have used the metaheuristics: NSGA-II for the genetic algorithm.

The evaluation consists of 4 different tests. The first(1) test (a.k.a. Pareto curves memory unconscious with graph transformation) is the application sent to our initial approach, then sent to the genetic mapping algorithm. The second(2) test (a.k.a. Pareto curves memory conscious) is the application sent to the modified genetic mapping algorithm with memory as an objective in its fitness function. The third(3) test (a.k.a. Pareto curves memory conscious with fusion) is the application sent to the modified genetic mapping algorithm with memory as an objective in its fitness function and takes into account fusion in the objective function. The fourth(4) test (a.k.a. Pareto curves memory conscious with or without fusion) is the application sent to the modified genetic mapping algorithm with memory as an objective in its fitness function and takes into account fusion but the genetic algorithm decides to enable or disable the memory optimization techniques.

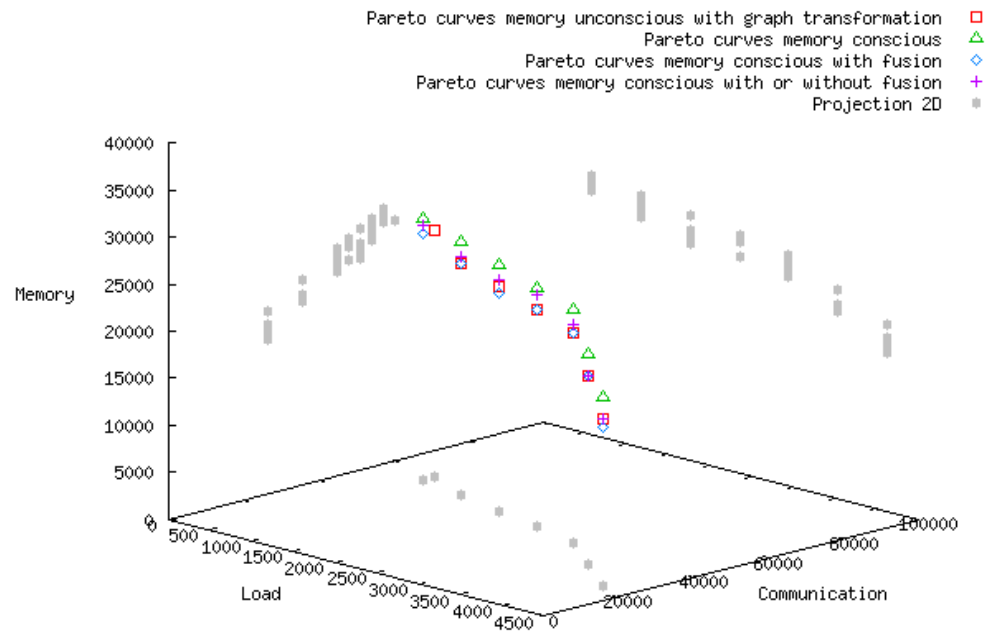


Figure 4.18 Pareto Curves - Three metrics (i.e. Memory, Communication and Load)

Figure 4.18 shows the results obtained by each test configuration. Each point is projected to a corresponding 2d plan (represented in the figure by the solid shape). This figure demonstrates the interaction of the three metrics: (1) memory cost, (2) communication cost and (3) load cost. The metrics memory cost represents the total memory space used by the platform. The metrics communication cost represents the total communication used on the platform. Finally the metrics load cost represents the load variance of the platform. As it is difficult to see results in a 3d pareto curves diagram, we added some 2d capture of each metrics combination (Memory and Communication -Figure 4.19, Memory and Load -Figure 4.20, Communication and Load -Figure 4.21).

Overall, the best results are obtained by the test “Pareto curves memory conscious with fusion”, then comes the test “Pareto curves memory unconscious with graph

transformation”. As mentioned earlier in the discussion, our heuristic approach, by imposing some limits on some metrics, may over cut the exploration domain, therefore cutting some potential optimal solution. The worst results are obtained by the test “Pareto curves memory conscious”. More clear and specific results will be found in the following.

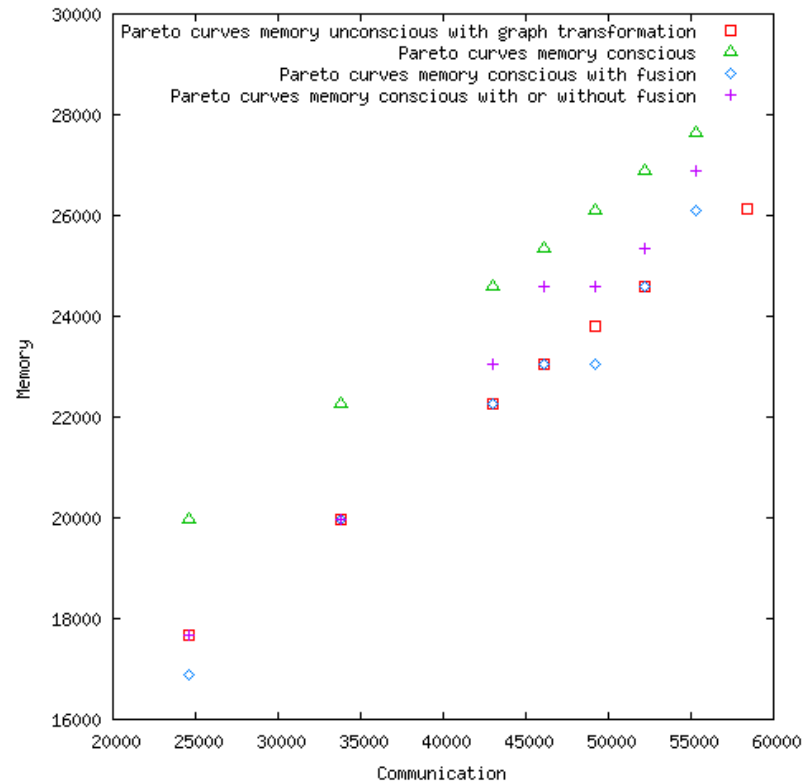


Figure 4.19 Pareto Curves - Two metrics (i.e. Memory and Communication)

Figure 4.19 shows the interaction between memory and communication. One can see that optimizing memory has an impact on communication. For similar communication costs, some tests find better solutions for memory cost. The test “Pareto curves memory conscious with fusion” gives the best results. The test “Pareto curves memory unconscious with graph transformation” gives occasionally the same result than the test “Pareto curves memory conscious with fusion”. The test “Pareto curves memory conscious with or without fusion”.

fusion” results oscillate between the results of two tests “Pareto curves memory conscious” and “Pareto curves memory conscious with fusion”.

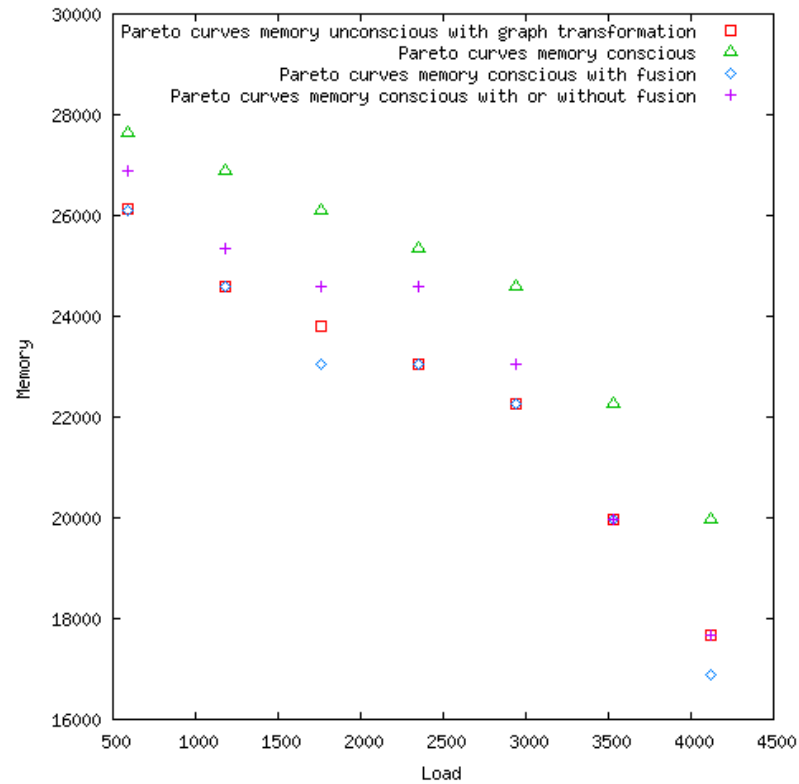


Figure 4.20 Pareto Curves - Two metrics (i.e. Memory and Load)

Figure 4.20 shows the interaction between the memory and the load. To be expected, if the load variance increases, the memory space decreases since more tasks are mapped onto the same processor. The test “Pareto curves memory conscious with fusion” gives the best results overall. In some instances the test “Pareto curves memory unconscious with graph transformation” gave the same result than the test “Pareto curves memory conscious with fusion”. In one case, “Pareto curves memory conscious with or without fusion” gave the same result than “Pareto curves memory conscious with fusion”.

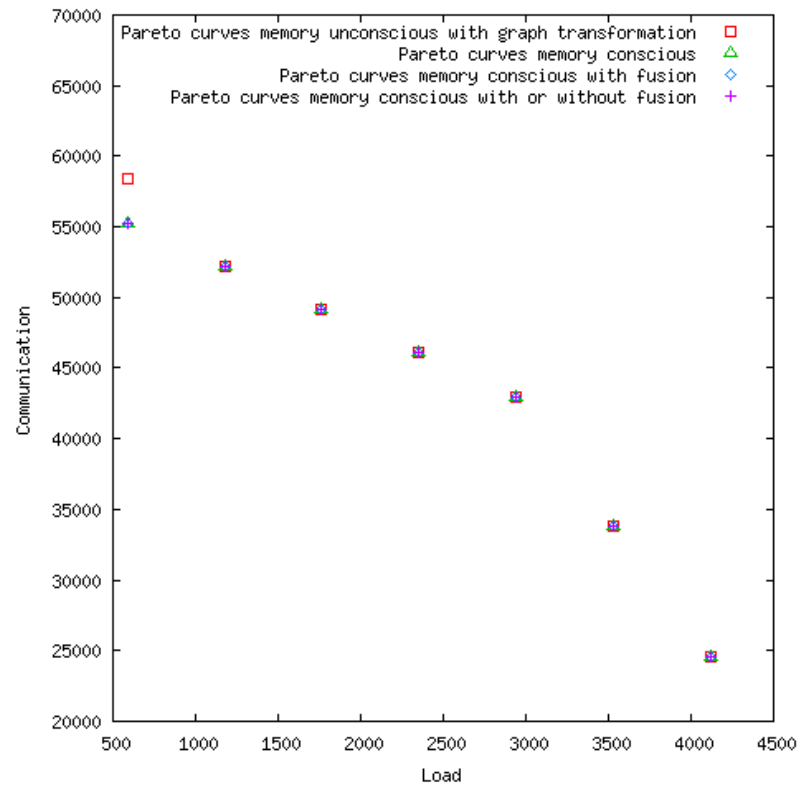


Figure 4.21 Pareto Curves - Two metrics (i.e. Communication and Load)

Figure 4.21 shows the interaction between the communication and load. As the load variance increases, the communication decreases. This result was to be expected since if the load variance increases, it means that more tasks are mapped to a processor cutting down the need of communication between processors. However, this graph is relevant to this paper since it demonstrates that all our tests have no impact on the mapping algorithm for the interaction between the communication and load metrics.

All tests were done with the same number of iterations of the genetic algorithm solution. Surprisingly, we have not obtained expected results with the “Pareto curves memory conscious with or without fusion”. This version explores a great exploration domain and we would have expected to see some difference on the other metrics like the load and the

communication cost. Therefore, more work needs to be done on our genetic algorithm approach to better understand the tendencies towards a memory optimized solution.

4.8 Discussion

This paper presented two different approaches to integrate memory optimization with mapping.

The first approach is completely independent from the mapping algorithm. This approach is considered a heterogeneous approach since it offers the possibility of interacting with different types of mapping algorithms as heuristic mapping algorithm and evolutionary algorithm.

The second approach is completely dependent of the mapping algorithm. This approach is considered a homogenous approach since it adds memory optimization objective function to the genetic algorithm. This approach is within the mapping algorithm.

It is difficult to make a comparison between the two approaches. None of them claims to guaranty the best optimal solution. The heuristic approach goes towards a single solution in a solution domain limited by the transformation algorithm. This approach transforms the task graph to optimize the memory then it sends it to a mapping algorithm. The memory optimization objectives are fixed and cannot be changed to explore different compromises with other optimization metrics. The evolutionary approach gives a set of potential solution. It is capable to explore compromise between memory optimization and the other optimization metrics. These compromises can be influenced by the type of application and the architecture of the MPSoC.

A designer can choose between the two approaches depending on his needs. The heuristic approach can be integrated to his existing mapping algorithm. The evolutionary approach can be used if the designer does not have a mapping algorithm.

4.9 Conclusion and Future Work

This paper has discussed two approaches to combine memory optimization with mapping. The first approach is a two step approach. The first step consists of transforming a task graph for memory optimization and the second step is a standard mapping algorithm. Experimentations demonstrated that the first approach may result in significant improvement for memory gain, communication load and the number of physical links between PEs. This paper also discussed the topic about a second approach integrated within a genetic mapping algorithm. Some results were presented and gives some promising results, however more work need to be done to capture the maximum potential of genetic algorithms.

Future work will explore the possibility of using genetic algorithms not only to select which transformation to apply but also to choose in which order to apply the selected memory optimization transformations. We will also concentrate using existing tools to demonstrate the full design flow from sequential code to parallelized code ready to be mapped and improving the evaluation of the load after the fusion of nodes.

References

- [1] A. A. Jerraya, and W. Wayne, *Multiprocessor Systems-on-Chips*, Elsevier ed., United States of America: Morgan Kaufmann, 2005.
- [2] B. Girodias, Y. Bouchebaba, G. Nicolescu *et al.*, "Application-Level Memory Optimization for MPSoC," *Rapid System Prototyping*, 2006. *Seventeenth IEEE International Workshop on*. pp. 169-178, 2006.
- [3] F. Catthoor, S. Wuytack, E. D. Greef *et al.*, *Custom Memory Management Methodology -- Exploration of Memory Organisation for Embedded Multimedia System Design*, Boston: Kluwer Academic Publishers, 1998.
- [4] A. Darte, "On the complexity of loop fusion," *Parallel Architectures and Compilation Techniques*, 1999. *Proceedings. 1999 International Conference on*. pp. 149-157, 1999.
- [5] K. Kennedy, "Fast greedy weighted fusion," *International Journal of Parallel Programming*, vol. 29, no. 5, pp. 463-91, 2001/10/, 2001.
- [6] S. Carr, and K. Kennedy, "Scalar replacement in the presence of conditional control flow," *Software - Practice and Experience*, vol. 24, no. 1, pp. 51-77, 1994/01/, 1994.
- [7] E. D. Greef, "Storage Size Reduction for Multimedia Application. Ph.D. Thesis," Katholieke Universiteit, Leuven, 1998.

- [8] K. Olukotun, B. A. Nayfeh, L. Hammond *et al.*, "The case for a single chip multiprocessor," *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. pp. 2-11, 1996.
- [9] M. E. Wolf, and M. S. Lam, "A data locality optimizing algorithm," *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. pp. 30-44, 1991.
- [10] M. Cierniak, and W. Li, "Unifying data and control transformations for distributed shared-memory machines," *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. pp. 205-217, 1995.
- [11] J. Hu, and R. Marculescu, "Communication and task scheduling of application-specific networks-on-chip: Communication and task scheduling of application-specific networks-on-chip: Computers and Digital Techniques, IEE Proceedings," *Computers and Digital Techniques, IEE Proceedings -*, vol. 152, no. 5, pp. 643-651, 2005.
- [12] C. Guangyu, L. Feihui, S. W. Son *et al.*, "Application mapping for chip multiprocessors." pp. 620-625.
- [13] S. Pasricha, and N. Dutt, "COSMECA: Application Specific Co-Synthesis of Memory and Communication Architectures for MPSoC." pp. 1-6.
- [14] L. Shih-wei, D. Zhaohui, W. Gansha *et al.*, "Data and computation transformations for Brook streaming applications on multiprocessors," *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*. p. 12 pp.
- [15] I. G. Michael, T. William, and A. Saman, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, San Jose, California, USA, 2006.
- [16] M. I. Gordon, W. Thies, M. Karczmarek *et al.*, "A stream compiler for communication-exposed architectures," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 291-303, 2002.
- [17] "BrookGPU," <http://graphics.stanford.edu/projects/brookgpu/lang.html>.
- [18] "StreamIt," <http://cag.csail.mit.edu/streamit>.
- [19] M. Ruggiero, A. Guerri, D. Bertozzi *et al.*, "A Fast and Accurate Technique for Mapping Parallel Applications on Stream-Oriented MPSoC Platforms with Communication Awareness," *International Journal of Parallel Programming*, vol. 36, no. 1, pp. 3-36, 2008.
- [20] S. Markus, and W. Thomas, "Mapping and Scheduling by Genetic Algorithms," in *Proceedings of the Third Joint International Conference on Vector and Parallel Processing: Parallel Processing*, 1994.
- [21] L. Tang, and S. Kumar, "A two-step genetic algorithm for mapping task graphs to a network on chip architecture," *Digital System Design, 2003. Proceedings. Euromicro Symposium on*. pp. 180-187.
- [22] L. Thiele, I. Bacivarov, W. Haid *et al.*, "Mapping Applications to Tiled Multiprocessor Embedded Systems." pp. 29-40.
- [23] A. Fraboulet, K. Kodary, and A. Mignotte, "Loop fusion for memory space optimization," *System Synthesis, 2001. Proceedings. The 14th International Symposium on*. pp. 95-100, 2001.
- [24] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing surveys*, vol. 26 (4), pp. 345-420, 1994, 1994.

- [25] A. Konaka, D. W. Coitb, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability Engineering and System Safety*, vol. 91, no. 9, pp. 992-1007 2006.
- [26] E. PIPS. "PIPS (interprocedural parallelizer for scientific programs)," 2005; <http://www.cri.enscm.fr>.
- [27] "SUIF, <http://suif.stanford.edu>," November 2006.
- [28] "CLooG," <http://www.prism.uvsq.fr/~cedb/bastools/cloog.html>.
- [29] D. P. Andy, S. Todor, N. Hristo *et al.*, "Tool Integration and Interoperability Challenges of a System-Level Design Flow: A Case Study," in Proceedings of the 8th international workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, Samos, Greece, 2008.
- [30] C. Bastoul, "Improving Data Locality in Static Control Programs. Thesis," Université Pierre & Marie Curie, 2004.
- [31] P. G. Paulin, C. Pilkington, M. Langevin *et al.*, "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 7, pp. 667-680, 2006, 2006.
- [32] M. Coppola, R. Locatelli, G. Maruccia *et al.*, "Spidergon: a novel on-chip communication network." p. 15.
- [33] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free," *Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on*. pp. 97-101.

CHAPITRE 5. DISCUSSION GÉNÉRALE

Dans ce chapitre, nous présenterons une discussion sur notre travail de thèse et sur l'impact des résultats obtenus.

5.1 Architecture avant-gardiste

Quel que soit le type d'architecture ou de modèle de programmation, l'optimisation de code est une étape importante de la conception au niveau système d'une application. Les performances obtenues par des optimisations architecturales ont leurs limites. Des limites physiques théoriques sont inhérentes à certaines technologies. Développer des architectures spécialisées pour une application précise occasionne de grandes dépenses et allonge considérablement les délais de mise en marché. Les optimisations au niveau du code sont les moins intrusives et assurent des gains de performance non négligeables.

C'est un des avantages des optimisations logiciels de ne pas dépendre du matériel et de la technologie. Dans le cadre de cette thèse, nous avons utilisé, à titre expérimental, des techniques d'optimisation mémoire sur des interconnexions avant-gardistes, opération que l'on peut effectuer, par exemple, avec un réseau optique sur puce [77]. Les réseaux optiques sont d'usage courant dans le domaine des télécommunications, mais leur utilisation comme interconnexion sur puce est encore très récente et controversée. Les interconnexions optiques sur puce permettent d'augmenter la bande passante et de diminuer la latence sans créer de collision à l'intérieur du réseau. Nous avons évalué différentes optimisations mémoire d'une application multimédia sur deux types d'interconnexion : (1) l'interconnexion électrique et (2) l'interconnexion optique.

Pour fin de comparaison, trois différentes architectures ont été utilisées. Deux d'entre elles sont des interconnexions électriques : (1) crossbar et (2) STBus [78] et la troisième est

une interconnexion optique : (3) ONoC [77]. Le Tableau 5.1 présente les caractéristiques de ces différentes interconnexions.

Tableau 5.1 Interconnexion Électrique versus Optique

	Électrique		Optique
	Crossbar	STBUS	ONoC
Architecture			
Interconnexion	Point à Point	Hiérarchique	Indirect
Fréquence	Jusqu'à 233MHz	Jusqu'à 233MHz	Jusqu'à 100MHz ⁹
Nombre d'initiateurs	< 10	>10	>10
Arbitre	Non	Oui	Non
Extensible	Non	Oui	Oui
Largeur du chemin de données	Jusqu'à 1024	Jusqu'à 128	Jusqu'à 32
Méthode de commutation	X	X	Circuit
Routage	Déterministe		
Pas de collision	Oui	Non	Oui

Nos expériences nous ont amenés à comparer différentes technologies d'interconnexion. Notre travail de thèse révèle que les techniques d'optimisation mémoire obtiennent des gains en performance identiques (i.e. réduction de la taille de la mémoire, du nombre d'accès mémoire et du temps d'exécution) quel que soit le type d'interconnexion. Pour des informations complémentaires, voir référence [77].

5.2 Modèle de programmation, placement et optimisation mémoire dans le flot de conception des MPSoCs.

Cette section donne une vision d'ensemble des travaux présentés dans le Chapitre 3 et le Chapitre 4 et en indique les retombées.

⁹ Limite due aux interfaces opto-électriques.

Le Chapitre 3 et le Chapitre 4 décrivent l'impact des techniques d'optimisation mémoire sur les performances d'un système. Ils montrent aussi que le modèle de programmation choisi pour spécifier une application à haut niveau d'abstraction influence la démarche du processus de conception.

Dans le domaine des MPSoCs, le modèle de programmation de type SMP parallélise l'application au niveau des données. L'étape du placement se fait de façon implicite, car les données se répartissent sur chaque unité de calcul. L'optimisation mémoire nécessite quelques adaptations telles que des décalages du domaine d'itération, des phases d'initialisation et des changements d'ordre d'itération. Ces adaptations sont nécessaires si l'on ne veut pas changer le comportement de l'application lors de l'application des techniques d'optimisation.

Dans le domaine des MPSoCs, le modèle de programmation de type streaming parallélise l'application au niveau des tâches. L'étape du placement se fait de façon explicite, car les possibilités de placement sont nombreuses et peuvent influencer les performances du système. L'étape de l'optimisation mémoire ne nécessite aucune adaptation, car l'application des techniques se fait comme dans un environnement monoprocesseur. Nous avons vu que l'étape de placement doit être consciente des techniques d'optimisation mémoire potentiellement applicables sur l'application. Par exemple, pour que des techniques d'optimisation comme la fusion soient appliquées après l'étape de placement, certaines tâches doivent être placées sur la même unité de calcul

Dans les années à venir, les applications envahiront les domaines du calcul informatisé, des communications et de la grande consommation. Cette perspective suppose une grande variété de fonctionnalités de standards pour un vaste marché. De plus en plus, les données sont traitées en flux ce qui explique la popularité croissante des architectures de type streaming. La complexité grandissante des applications favorise l'avènement des architectures avec un modèle de programmation de type hybride. L'architecture globale serait une architecture avec un modèle de programmation de type streaming dont les sous

composantes seraient conçues à partir d'architectures avec un modèle de programmation de type SMP. Le choix entre le modèle de programmation de type SMP et celui de type streaming s'en vient de plus en plus difficile. Nous avons démontré que chaque type de modèle de programmation impose un placement différent de l'application. Le modèle de programmation de type SMP se caractérise par un placement implicite qui est régi par les données. Le modèle de programmation de type streaming implique un placement explicite nécessitant une évaluation des fonctions objectives pour obtenir un placement judicieux. Le domaine des MPSoCs axé sur des modèles de programmation de type hybride va requérir des techniques de placement plus évoluées et plus complexes.

Force est de reconnaître, en définitive, le rôle important du modèle de programmation dans le flot de conception des MPSoCs. L'apparition de nouveaux modèles de programmation entraînera l'ajout d'étapes supplémentaires.

Cette thèse porte sur les modèles de programmation de type SMP et streaming. Le travail accompli, nous a amenés à débiter un nouveau projet de recherche; celui-ci s'intéresse plus spécifiquement au modèle de programmation de type hybride.

Il est établi que les flots de conception existants des MPSoCs emploient généralement un seul modèle de programmation, soit de type SMP, soit de type streaming. La combinaison des deux modèles de programmation SMP et streaming à l'étape de parallélisation du code séquentiel de l'application devrait permettre d'optimiser l'étape de placement des applications multimédia.

Ce nouveau projet se fixe comme objectifs de définir un algorithme d'optimisation de placement utilisant deux modèles de programmation en respectant les métriques de conception et de concevoir une approche d'analyse automatique d'applications pour faciliter leur placement selon l'algorithme d'optimisation.

5.3 Positionnement dans le contexte actuel de l'optimisation mémoire pour les applications de type streaming

Cette thèse propose l'intégration des concepts de transformation de code dans le flot de conception pour l'optimisation mémoire. Cette intégration est faite en guidant l'étape placement pour favoriser l'application de techniques d'optimisation mémoire.

À l'origine, les algorithmes de placement se concentraient uniquement sur l'optimisation des performances en termes de puissance de calcul. Aujourd'hui, on ne peut ignorer certaines contraintes liées à l'arrivée de nouvelles architectures telles que les MPSoCs. Ces architectures sont dévoreuses d'énergie et de mémoire d'où la nécessité d'optimiser au maximum les algorithmes de placement [55].

La plupart des recherches se sont attardées au problème de mémoire et d'énergie à la conception au niveau système, mais avec une vue de concepteur matériel. Leur approche propose de réduire la mémoire et la consommation d'énergie en faisant des modèles complexes de la consommation et la taille du circuit. Notre approche provient d'une vue de concepteur logiciel et permet de réduire la taille du circuit et par ce fait la consommation d'énergie en réduisant les accès mémoire et la taille mémoire requis par l'application. Cette affirmation sur la consommation d'énergie est basée sur les travaux réalisés par un groupe de recherche d'IMEC [32] pionnier dans le travail sur les transformations de programme visant à réduire la consommation d'énergie dans les applications embarquées dominées par les données.

Traditionnellement, le domaine de la compilation et le domaine de la conception au niveau système n'interagissent pas. Cette thèse combine des concepts provenant de ces deux domaines, soit la transformation de code pour l'optimisation mémoire et les techniques de placement. Basé sur nos connaissances nous croyons qu'il n'y a pas d'approche combinant des concepts de ces deux domaines pour l'optimisation mémoire. Pour ces raisons, la principale contribution que nous revendiquons est l'intégration d'un

algorithme et non l'algorithme en lui-même. Ce qui explique que nous n'avons fait aucune comparaison avec une approche existante.

5.4 Automatisation

Un projet exploratoire sur l'automatisation des techniques d'optimisation mémoire a été lancé en coopération avec notre partenaire industriel. Ce projet met en œuvre les techniques d'optimisation mémoire présentées dans notre thèse. Les concepts et techniques sont utilisés pour le développement de leur environnement de conception appelé MultiFlex [20]. L'automatisation des techniques d'optimisation mémoire a été intégrée dans l'outil Pluto.

Multiflex[20] est actuellement utilisé dans le multimédia et, en particulier, dans les microprocesseurs des set-up box, pour l'encodage et le décodage de données. Cet environnement de développement ne se limite pas au secteur multimédia; on le retrouve dans les télécommunications, le biomédical et divers autres secteurs.

CONCLUSION ET RECOMMANDATIONS

Conclusion

Les systèmes multiprocesseurs sur puce seront de plus en plus utilisés, car ils réalisent un bon équilibre entre puissance de calcul, consommation d'énergie et miniaturisation. Conséquence des recherches continues dont ils sont l'objet, les systèmes et applications de type multimédia ne cessent d'exiger de plus en plus de puissance. Le progrès des techniques d'optimisation adaptées aux architectures MPSoCs et la venue de nouvelles méthodologies vont en améliorer les performances et en faciliter l'accès.

La première contribution de cette thèse consiste à donner une vue d'ensemble de l'optimisation mémoire dans la conception système d'architecture MPSoC pour une meilleure compréhension des problématiques et défis de cette architecture.

Come deuxième contribution, nous proposons une adaptation des techniques d'optimisation de mémoire. Les techniques actuelles ne sont pas toujours adaptées aux architectures MPSoCs. Les architectures MPSoCs à modèle de programmation de type SMP sont celles qui nous intéressent plus particulièrement. Dans ce modèle de programmation, la même application est exécutée par toutes les unités de calcul et seules les données sont parallélisées. Une adaptation de ces techniques était nécessaire. L'adaptation des techniques d'optimisation mémoire a permis d'optimiser l'espace mémoire et la grandeur du code de l'application, et de diminuer le temps d'exécution. Une étude de l'effet d'environnement multiprocesseur et multifil sur ces techniques montre que la granularité du parallélisme peut influencer leur exécution. Un raffinement de la granularité maximise le parallélisme de l'application, mais peut rendre l'exécution de techniques d'optimisation mémoire difficile, voire même impossible. Nos travaux de recherche ont été faits en collaboration avec STMicroelectronics. Les applications Détection d'images lacunaires et Dématriçage ont été utilisées lors de la phase d'évaluation

des apports de cette contribution. Les techniques décrites permettent d'améliorer le taux de succès de l'antémémoire de 20% et de diminuer le temps d'exécution de 50% pour des applications telles que décrites dans cette thèse (section 2.2.2).

La troisième contribution est la description de deux approches pour intégrer les techniques d'optimisation mémoire avec l'étape du placement de l'application. Ici, ce sont les architectures MPSoCs à modèle de programmation de type streaming qui nous intéressent. Dans ce modèle de programmation, les techniques d'optimisation n'ont pas besoin d'être adaptées comme dans le modèle de programmation de type SMP, car elles s'exercent sur chaque unité de calcul séparément. Par contre, l'étape du placement de l'application influence grandement leur exécution. La première approche se fait en deux étapes. La première étape est une transformation de graphe de tâches et la deuxième, l'exécution d'un algorithme de placement. La seconde approche se fait en une étape. Un algorithme d'évolution pour le placement est modifié pour intégrer les concepts d'optimisation mémoire. Ces deux approches améliorent l'optimisation mémoire, la communication et le temps d'exécution des tâches. STMicroelectronics a apporté sa collaboration à cette autre partie de nos travaux de recherche. Des applications générées aléatoirement et l'application Dématriçage ont été utilisées pour valider les approches proposées par cette contribution. Les résultats dépendent en grande partie de l'application. Nos expérimentations soldent par une diminution de la taille de la mémoire de 36% et par une diminution du coût de communication de 8% pour des applications telles que décrites dans cette thèse (section 2.2.2).

Recommandations

Dans l'adaptation des techniques d'optimisation, l'analyse et l'exécution des techniques sont faites manuellement par le développeur. L'automatisation de l'analyse et de l'exécution des techniques ne sera obtenue que par des études plus poussées.

L'automatisation facilitera l'optimisation lors de la conception d'application pour les architectures MPSoCs.

Plus le niveau d'abstraction d'un langage est élevé, plus il est difficile de faire une analyse automatique du code et de la sémantique en particulier. Les outils disponibles pour l'analyse des codes se limitent à des langages non orientés objet comme C et ne supportent qu'un sous-ensemble de la sémantique du langage. Cette analyse est difficile d'où la difficulté d'automatiser toutes les techniques de transformation d'optimisation mémoire. La complexité augmente lorsqu'il est question d'un environnement multiprocesseur. L'outil analysant le code doit s'ajuster au type, parfois complexe, de l'architecture.

Les difficultés pour automatiser les transformations d'optimisation mémoire ne se limitent pas seulement aux contraintes mentionnées au paragraphe précédent. On ne pourra automatiser une transformation d'optimisation mémoire, sans d'abord comprendre le dynamisme caractérisant chaque transformation d'optimisation mémoire. Par exemple, certaines transformations peuvent empêcher l'application d'une transformation. Des recherches plus approfondies sont donc nécessaires pour obtenir un outil pouvant automatiser les transformations à appliquer et dans quel ordre et quel contexte elles doivent l'être.

Dans l'intégration des techniques d'optimisation à l'étape de placement, on utilise un algorithme d'évolution. Cet algorithme a été modifié pour indiquer quelle doit être la fusion à appliquer sur le code. Il est important de poursuivre l'étude des algorithmes d'évolution et d'apprendre à mieux les utiliser. Ce type d'algorithme devrait permettre d'automatiser le choix des transformations à exécuter et l'ordre de leur exécution. Une première recherche portant sur le choix des solutions initiales des colonies pourrait donner des résultats intéressants.

Notre thèse exploite deux modèles de programmation différents: le modèle de programmation de type SMP (voir Chapitre 3) et le modèle de programmation de type

streaming (voir Chapitre 4). Le modèle de programmation de type SMP s'applique aux systèmes ayant des ressources de traitement symétriques avec des mémoires partagées. Le modèle de programmation de type streaming permet d'encapsuler efficacement l'application en composante logicielle bien définie tout en ayant une sémantique de communication axée sur le flux de données. Dans le domaine de conception de multiprocesseur système sur puce, ce sont les deux modèles de programmation les plus utilisés. Il en existe plusieurs autres. L'apparition de modèles de programmation hybride, par exemple un modèle de programmation intégrant un modèle de programmation de type SMP et un modèle de programmation de type streaming, est d'actualité. Cette nouvelle tendance s'explique par le fait que les avantages et inconvénients de chaque modèle se contrebalancent. Un de nos projets en cours se fixe comme objectif de définir un algorithme d'optimisation de placement utilisant un modèle de programmation de type hybride. Cette approche consiste à analyser de façon automatique des applications pour permettre leur placement selon cet algorithme d'optimisation. Les concepts développés par ce projet concerneront le secteur biomédical qui utilise des applications de traitement d'images en trois dimensions pendant des chirurgies mineures peu invasives.

Notre thèse porte sur les MPSoCs n'ayant qu'une seule application. De plus en plus, on utilisera les MPSoCs comme support simultané de plusieurs applications. Une simple application dispose de la quasi-totalité des ressources. Un placement statique s'avère suffisant. Avec plusieurs applications, les ressources peuvent être partiellement disponibles. Il faut d'autres recherches scientifiques pour savoir s'il est possible de faire du placement dynamique qui puisse se modifier selon la disponibilité des ressources et du nombre d'applications s'exécutant sur le MPSoC.

Finalement, une application peut être grandement optimisée au niveau logiciel. Notre thèse traite surtout des transformations d'optimisation mémoire et peu du parallélisme de données. Ce secteur de la compilation n'est pas pour autant négligé; il suscite d'importantes recherches sur les méthodes permettant d'automatiser la parallélisation par la vectorisation

de code. La vectorisation permet de transformer une application à implémentation scalaire en une application vectorisée. Sur un vecteur, une instruction effectue plusieurs opérations simultanées tandis que sur un scalaire, elle n'en fait qu'une à la fois.

L'optimisation a un impact très important sur les performances du système, la consommation de puissance et le coût de production. Elle doit donc être l'objet de continuelles recherches et d'études toujours plus poussées.

LISTE DE RÉFÉRENCES

- [1] P. G. Paulin, C. Pilkington, et E. Bensoudane, "StepNP: a system-level exploration platform for network processors," *Design & Test of Computers, IEEE*, vol. 19, no. 6, pp. 17-26, 2002.
- [2] "International Technology Roadmap for Semiconductors (ITRS) , <http://www.itrs.net>," no. November 2006. [En ligne]. Disponible.
- [3] A. Jerraya, H. Tenhunen, et W. Wolf, "Guest Editors' Introduction: Multiprocessor Systems-on-Chips," *Computer*, vol. 38, no. 7, pp. 36-40, 2005.
- [4] W. Wolf, "The future of multiprocessor systems-on-chips," in *Design Automation Conference 2004*, 2004, pp. 681-685.
- [5] A. A. Jerraya et W. Wayne, *Multiprocessor Systems-on-Chips*, Elsevier^e éd., United States of America: Morgan Kaufmann, 2005.
- [6] J. Hennessy, D. Goldberg, et D. Patterson, *Computer Architecture: A Quantitative Approach*: Morgan Kaufmann Publishers.
- [7] P. G. Paulin, "The Inexorable Progression of Parallelism in Systems-on-a-Chip," *System Design Frontier*, vol. 3, no. 2, 2006.
- [8] A. Fraboulet, "Optimisation mémoire et de la consommation des systèmes multimédia embarqués," Institut National des Sciences Appliquées (INSA), Lyon, 2001.
- [9] J. Qaddour et R. A. C. Barbour, "Evolution to 4G wireless: problems, solutions, and challenges," in *Book of Abstracts. ACS/IEEE International Conference on Computer Systems and Applications, 3-6 Jan. 2005*, 2004, pp. 78.
- [10] T. Austin, D. Blaauw, S. Mahlke, T. Mudge, C. Chakrabarti, et W. Wolf, "Mobile supercomputers," *Computer*, vol. 37, no. 5, pp. 81-83, 2004.
- [11] S. Signell, D. Rodriguez de Llera Gonzalez, et M. Ismail, "Radio design for future wireless soc platforms- an overview," in *Norchip Conference, 2004. Proceedings*, 2004, pp. 277-280.
- [12] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, et H. De Man, "Global communication and memory optimizing transformations for low" in *VLSI Signal Processing, VII, 1994.*, [Workshop on], 1994, pp. 178-187.

- [13] M. E. Wolf et M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991, pp. 30-44.
- [14] M. Haines et W. Bohm, "An evaluation of software multithreading in a conventional," in *Parallel and Distributed Processing, 1993. Proceedings of the Fifth IEEE Symposium on*, 1993, pp. 106-113.
- [15] "SUIF, <http://suif.stanford.edu>" no. November 2006. [En ligne]. Disponible.
- [16] E. PIPS, "PIPS (interprocedural parallelizer for scientific programs)," no. 2005. [En ligne]. Disponible: <http://www.cri.ensmp.fr>.
- [17] U. Bondhugula, A. Hartono, J. Ramanujam, et P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101-113, 2008.
- [18] U. Bondhugula, "PLUTO - An automatic parallelizer and locality optimizer for multicores," 2009. [En ligne]. Disponible: <http://www.cse.ohio-state.edu/~bondhugu/pluto/>.
- [19] P. G. Paulin, "Automatic mapping of parallel applications onto multi-processor platforms: a multimedia application," in *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*, 2004, pp. 2-4.
- [20] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagne, et G. Nicolescu, "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 7, pp. 667-680, 2006.
- [21] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, et G. Nicolescu, "Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management," in *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, 2004, pp. 48-53.
- [22] C. Bastoul, "Improving Data Locality in Static Control Programs," Université Pierre & Marie Curie, 2004.
- [23] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, et O. Temam, "Putting Polyhedral Loop Transformations to Work," in *Languages and Compilers for Parallel Computing*, 2004, pp. 209-225.

- [24] "Daedalus," 2009. [En ligne]. Disponible: <http://artemis.liaacs.nl/Site/Daedalus%20home.html>.
- [25] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, et E. Deprettere, "Daedalus: Toward composable multimedia MP-SoC design," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, 2008, pp. 574-579.
- [26] "Compaan," 2008. [En ligne]. Disponible: <http://www.liaacs.nl/~cserc/compaan/>.
- [27] S. Todor, Z. Claudiu, T. Alexandru, K. Bart, et D. Ed, "System Design Using Kahn Process Networks: The Compaan/Laura Approach," in *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, vol. 1, Ed.^Eds., ed.: IEEE Computer Society, 2004, pp.
- [28] S. Carr et K. Kennedy, "Scalar replacement in the presence of conditional control flow," *Software - Practice and Experience*, vol. 24, no. 1, pp. 51-77, 1994.
- [29] E. D. Greef, "Storage Size Reduction for Multimedia Application. Ph.D. Thesis," Katholieke Universiteit, Leuven, 1998.
- [30] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, et K. Chang, "The case for a single chip multiprocessor", 1996
- [31] M. Cierniak et W. Li, "Unifying data and control transformations for distributed shared-memory machines," in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 1995, pp. 205-217.
- [32] F.Catthoor, S.Wuytack, E. D. Greef, F.Balasa, L.Nachtergaele, et A.Vandecappelle, *Custom Memory Management Methodology -- Exploration of Memory Organisation for Embedded Multimedia System Design*, Boston: Kluwer Academic Publishers, 1998.
- [33] A. Darte, "On the complexity of loop fusion," in *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, 1999, pp. 149-157.
- [34] K. Kennedy, "Fast greedy weighted fusion," *International Journal of Parallel Programming*, vol. 29, no. 5, pp. 463-91, 2001.
- [35] A. Fraboulet, K. Kodary, et A. Mignotte, "Loop fusion for memory space optimization," in *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, 2001, pp. 95-100.

- [36] P. Marchal, F. Catthoor, et J. I. Gomez, "Optimizing the memory bandwidth with loop fusion," in *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, 2004, pp. 188-193.
- [37] M. Kandemir, I. Kadayif, A. Choudhary, et J. A. Zambreno, "Optimizing inter-processor data locality," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, 2002, pp. 127-135.
- [38] I. Kadayif et M. Kandemir, "Data space-oriented tiling for enhancing locality," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 2, pp. 388-414, 2005.
- [39] M. Kandemir, "Data space oriented tiling," in *Programming Languages and Systems. 11th European Symposium on Programming, ESOP 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002. Proceedings, 8-12 April 2002*, 2002, pp. 178-93.
- [40] F. Li et M. Kandemir, "Locality-conscious workload assignment for array-based computations in MPSOC architectures," in *Design Automation Conference, 2005. Proceedings. 42nd*, 2005, pp. 95-100.
- [41] V. Krishnan et J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *Computers, IEEE Transactions on*, vol. 48, no. 9, pp. 866-880, 1999.
- [42] T. Van Achteren, G. Deconinck, F. Catthoor, et R. Lauwereins, "Data reuse exploration techniques for loop-dominated applications," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, 2002, pp. 428-435.
- [43] I. Ilya, B. Erik, M. Miguel, et D. Nikil, "DRDU: A data reuse analysis technique for efficient scratch-pad memory management," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 2, pp. 15, 2007.
- [44] F. Catthoor, K. Danckaert, K. K. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. van Achteren, et T. Omnes, *Data Access and Storage Management for Embedded Programmable Processors*: Springer, 2002.
- [45] M. J. Forsell, "Step caches - a novel approach to concurrent memory access on shared memory MP-SOCs," in *NORCHIP Conference, 2005. 23rd*, 2005, pp. 74-77.
- [46] Y. Bouchebaba et F. Coelho, "Tiling and memory reuse for sequences of nested loops," in *Euro-Par 2002 Parallel Processing. 8th International Euro-Par Conference. Proceedings, 27-30 Aug. 2002*, 2002, pp. 255-64.

- [47] Y. Bouchebaba, B. Girodias, G. Nicolescu, E. M. Aboulhamid, P. Paulin, et B. Lavigueur, "MPSoC memory optimization using program transformation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 4, pp. 43, 2007.
- [48] Y. Bouchebaba, B. Lavigueur, B. Girodias, G. Nicolescu, et P. G. Paulin, "MPSoC memory optimization for digital camera applications: Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, 2007, pp. 424-427.
- [49] B. Girodias, Y. Bouchebaba, G. Nicolescu, E. M. Aboulhamid, P. Paulin, et B. Lavigueur, "Application-Level Memory Optimization for MPSoC," in *Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on*, 2006, pp. 169-178.
- [50] C. Ghez, M. Miranda, A. Vandecappelle, F. A. C. F. Catthoor, et D. A. V. D. Verkest, "Systematic high-level address code transformations for piece-wise linear indexing: illustration on a medical imaging algorithm," in *Signal Processing Systems, 2000. SiPS 2000. 2000 IEEE Workshop on*, 2000, pp. 603-612.
- [51] P. Schaumont, B.-C. C. Lai, W. Qin, et I. Verbauwhede, "Cooperative multithreading on embedded multiprocessor architectures enables energy-scalable design," in *Design Automation Conference, 2005. Proceedings. 42nd*, 2005, pp. 27-30.
- [52] Y.-K. Chong et K. Hwang, "Performance analysis of four memory consistency models for Multithreaded Multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 6, no. 10, pp. 1085-1099, 1995.
- [53] G. Dimitroulakos, M. D. Galanis, et C. E. Goutis, "Performance improvements using coarse-grain reconfigurable logic in embedded SOCs," in *Field Programmable Logic and Applications, 2005. International Conference on*, 2005, pp. 630-635.
- [54] B. M. Al-Hashimi, *System-on-Chip: Next Generation Electronics*: IEE, 2006.
- [55] J. Hu et R. Marculescu, "Communication and task scheduling of application-specific networks-on-chip," *Computers and Digital Techniques, IEE Proceedings*, vol. 152, no. 5, pp. 643-651, 2005.

- [56] C. Guangyu, F. Li, S. W. Son, et M. Kandemir, "Application mapping for chip multiprocessors," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, 2008, pp. 620-625.
- [57] S. Pasricha et N. Dutt, "COSMECA: Application Specific Co-Synthesis of Memory and Communication Architectures for MPSoC," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, 2006, pp. 1-6.
- [58] L. Shih-wei, D. Zhaohui, W. Gansha, et A. G.-Y. L. Guei-Yuan Lueh, "Data and computation transformations for Brook streaming applications on multiprocessors," in *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, 2006, pp. 12 pp.
- [59] M. I. Gordon, W. Thies, et S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, vol., Ed.^Eds., ed. San Jose, California, USA: ACM, 2006, pp.
- [60] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, et S. Amarasinghe, "A stream compiler for communication-exposed architectures," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 291-303, 2002.
- [61] "BrookGPU," 2008. [En ligne]. Disponible: <http://graphics.stanford.edu/projects/brookgpu/lang.html>.
- [62] "StreamIt." [En ligne]. Disponible: <http://cag.csail.mit.edu/streamit>.
- [63] L. Benini et G. De Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70-78, 2002.
- [64] M. Ruggiero, A. Guerri, D. Bertozzi, M. Milano, et L. Benini, "A Fast and Accurate Technique for Mapping Parallel Applications on Stream-Oriented MPSoC Platforms with Communication Awareness," *International Journal of Parallel Programming*, vol. 36, no. 1, pp. 3-36, 2008.
- [65] S. Markus et W. Thomas, "Mapping and Scheduling by Genetic Algorithms," in *Proceedings of the Third Joint International Conference on Vector and Parallel Processing: Parallel Processing*, vol., Ed.^Eds., ed.: Springer-Verlag, 1994, pp.
- [66] L. Tang et S. Kumar, "A two-step genetic algorithm for mapping task graphs to a network on chip architecture," in *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, 2003, pp. 180-187.

- [67] L. Thiele, I. Bacivarov, W. Haid, et H. Kai, "Mapping Applications to Tiled Multiprocessor Embedded Systems," in *Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on*, 2007, pp. 29-40.
- [68] "Hyperthreading, <http://arstechnica.com/articles/paedia/cpu/hyperthreading.ars.>" no. 2006. [En ligne]. Disponible.
- [69] "MathWorks, <http://www.mathworks.com.>" no. 2008. [En ligne]. Disponible.
- [70] A. Behboodian, "Model-Based Design," in *DSP Magazine*, vol., Ed.^Eds., ed., 2006, pp.
- [71] "POSIX." [En ligne]. Disponible: <http://standards.ieee.org/regauth/posix/>.
- [72] J. Gummaraju et M. Rosenblum, "Stream programming on general-purpose processors," in *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, 2005, pp. 12 pp.
- [73] P. Quinton, "The systematic design of systolic arrays," in *Centre National de Recherche Scientifique on Automata networks in computer science: theory and applications*, vol., Ed.^Eds., ed. Paris, France: Princeton University Press, 1987, pp.
- [74] R. M. Karp, R. E. Miller, et S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *J. ACM*, vol. 14, no. 3, pp. 563-590, 1967.
- [75] "Wikipedia, <http://www.wikipedia.org>" no. November 2006. [En ligne]. Disponible: <http://www.wikipedia.org>.
- [76] "Demosaiicing, http://www.imageval.com/public/Products/ISET/ISET_Manual/Demosaiicing.htm." no. 2006. [En ligne]. Disponible: http://www.imageval.com/public/Products/ISET/ISET_Manual/Demosaiicing.htm.
- [77] M. Briere, B. Girodias, Y. Bouchebaba, G. Nicolescu, F. Mieyeville, F. Gaffiot, et I. O' Connor, "System level assessment of an optical NoC in an MPSoC platform," in *Proceedings of the conference on Design, automation and test in Europe*, vol., Ed.^Eds., ed. Nice, France: EDA Consortium, 2007
- [78] A. Bona, V. Zaccaria, et R. Zafalon, "System level power modeling and simulation of high-end industrial network-on-chip," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 3, 2004, pp. 318-323 Vol.3.

ANNEXE 1 – Publications

Journaux

Acceptés et publiés

- [1] BOUCHEBABA, Y., GIRODIAS, B., COELHO, F., NICOLESCU, G., ABOULHAMID, E.M., “*Buffer and register allocation for memory space optimization*”. Springer Journal of VLSI Signal Processing, 2007
- [2] BOUCHEBABA, Y., GIRODIAS, B., GAGNE, V., NICOLESCU, G., ABOULHAMID, E.M., PAULIN, P., LAVIGUEUR B., “*MPSoC memory optimization using program transformation*”. ACM Journal Transactions on Design Automation of Electronic Systems (TODAES), 2007.
- [3] GIRODIAS, B., BOUCHEBABA, Y., NICOLESCU, G., ABOULHAMID, E.M., PAULIN, P., LAVIGUEUR, B., “*Multiprocessor, Multithreading and Memory Optimization for On-Chip Multimedia Applications*”. Springer Journal of VLSI Signal Processing, 2008.

Soumis

- [4] GIRODIAS, B., GHEORGHE, L., BOUCHEBABA, Y., NICOLESCU, G., ABOULHAMID, E.M., PAULIN, P., LANGEVIN, M., “*Integrating Memory Optimization with Mapping Algorithms for Multi-Processors System-on-Chip*”. Submitted to IEEE TECS Journal. (Juin 2009)
- [5] FOURMIQUE, A., GIRODIAS, B., NICOLESCU, G., ABOULHAMID, E.M., “*Wireless Design Platform combining Simulation and Testbed Environments*”. Submitted to IEEE TCAD Journal. (Septembre 2009)

Articles

Acceptés et publiés

- [6] GIRODIAS, B., BOUCHEBABA, Y., NICOLESCU, G., ABOULHAMID, E.M., PAULIN, P., LAVIGUEUR, B., “*Application-Level Memory Optimization for MPSoC*”. Rapid System Prototyping 2006. RSP 2006.
- [7] BOUCHEBABA, Y., LAVIGUEUR, B, GIRODIAS, B., NICOLESCU, G., ABOULHAMID, E.M., PAULIN, P., LAVIGUEUR B., “*MPSoC memory optimization for digital camera applications*”. EUROMICRO CONFERENCE on DIGITAL SYSTEM DESIGN 2007. DSD 2007.
- [8] BRIÈRE M., GIRODIAS B., BOUCHEBABA Y., NICOLESCU G. O'CONNOR I., MIEYEVILLE F., “*System Level Assessment of an Optical NoC in an MPSoC Platform*”. Design Automation and Test in Europe 2007, DATE 2007.
- [9] FOURMIQUE, A., GIRODIAS, B., NICOLESCU, G., ABOULHAMID, E.M., “*Co-Simulation Based Platform for Wireless Protocols Design Explorations*”. Design Automation and Test in Europe 2009, DATE 2009.

Soumis

- [10] GIRODIAS, B., GHEORGHE, L., BOUCHEBABA, Y., NICOLESCU, G., ABOULHAMID, E.M., PAULIN, P., LANGEVIN, M., “*Combining Memory Optimization with Mapping of Multimedia Applications for Multi-Processors System-on-Chip*”. Submitted to ESTIMedia 2009.

Chapitre de livre

Publications 2010

- [11] GIRODIAS, B., BOUCHEBABA, Y., NICOLESCU, G., ABOULHAMID, E.M., PAULIN, P., LAVIGUEUR, B., “*Chapter 7: Compiler Techniques for Application Level Memory Optimization for MPSoC*”. Multi-Core Embedded Systems, CRC Press, 2010
- [12] FOURMIQUE, A., GIRODIAS, B., NICOLESCU, G., ABOULHAMID, E.M., “*Chapter 10: Platform for wireless design protocol exploration*”. Heterogeneous Systems Design: Theory and Practice, Springer, 2010

ANNEXE 2 – Complexité de l'algorithme

L'annexe 2 présente la complexité de l'algorithme ($O(N^2 2^M)$) utilisé dans l'approche pour l'optimisation mémoire dans le flot de conception des MPSoCs à modèle de programmation de type streaming.

La figure de la page précédente illustre l'analyse de la complexité de l'algorithme. Chaque boîte représente une fonction ou sous-fonction. Les boîtes se trouvant au même niveau hiérarchique sont exécutées séquentiellement. Les flèches représentent la décomposition hiérarchique d'une fonction ou d'une sous-fonction.

L'analyse est basée sur ces quatre définitions suivantes:

Définition 1: Un **ensemble de tâches** est défini comme étant le regroupement de tâches qui peuvent être fusionnées.

Définition 2: Soit ***n*** est le nombre d'**ensembles de tâches** ayant des tâches en commun. Le nombre de ***n*** ne peut pas être supérieur à ***t*** (i.e. ***n* ≤ *t***).

Définition 3: Soit ***m*** est le nombre de tâches dans chaque ensemble de tâches. Le nombre de ***m*** ne peut pas être supérieur à ***t*** (i.e. ***m* ≤ *t***).

Définition 4: Soit ***t*** est le nombre de tâches dans un graphe de tâches d'une application donnée. Le nombre ***t*** est toujours égal à la sommation de ***m*** et ***n*** moins un (i.e. ***m* + *n* - 1 = *t***).

