AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

UNIVERSITÉ DE LORRAINE

École doctorale Informatique Automatique Électronique Électrotechnique Mathématiques

Méthodologie de conception d'architectures reconfigurables dynamiquement, application au transcodage vidéo

THÈSE

présentée et soutenue publiquement le 02 Décembre 2013

pour l'obtention du

Doctorat de l'Université de Lorraine

(Mention systèmes électroniques)

par

Eric DABELLANI

Composition du jury

Rapporteurs :	M. Bertrand GRANADO, Professeur, LIP6, Paris
	M. François-Xavier COUDOUX, Professeur, IEMN, Valenciennes
Examinateurs :	M. Dominique DALLET, Professeur, IMS, Bordeaux
	M. Serge WEBER, Professeur, IJL, Nancy - Directeur de thèse
	M. Yves BERVILLER, Maître de conférences , IJL, Nancy - Co-directeur de thèse
	M. Hassan RABAH, Professeur, IJL, Nancy

Institut Jean Lamour (UMR7198) - Equipe MAE 406
Département 4 "Nanomatériaux, Électronique et Vivant"
Université de Lorraine - 54506 Vandoeuvre-Lès-Nancy

Remerciements

J'adresse mes sincères remerciements à Monsieur Bertrand GRANADO, Professeur à l'Université Pierre et Marie Curie de Paris, ainsi qu'à Monsieur François-Xavier COUDOUX, Professeur à l'Université de Valenciennes, pour l'attention qu'ils ont accordés à la lecture de ce mémoire de thèse, et pour avoir accepté d'en être les rapporteurs. Je tiens également à remercier Monsieur Dominique DALLET, Professeur à l'Université de Bordeaux, pour avoir accepté d'examiner ce travail et de participer au jury.

Je remercie mes directeurs de thèse, Monsieur Serge WEBER et Monsieur Yves BERVILLER pour les précieux conseils qu'ils ont pu m'apporter durant l'encadrement de mes travaux. Mais également pour leurs recommendations lors de mes premières années d'enseignement. Pour les mêmes raisons, je tiens à remercier Monsieur Hassan RABAH, pour sa disponibilité, et son soutien tout au long de ces années.

Mes remerciements les plus chaleureux vont également à tous les membres et collègues de l'équipe 406 (anciennement LIEN) pour l'aide qu'ils ont pu m'apporter à un moment ou à un autre. En particulier Nicolas MARQUES, collègue et ami depuis nos années de licence. Patrice ROTH, pour tous les chocolats chauds, et les bons moments passés en sa compagnie. Christine DAUDENS et Pierre SCHMITT, pour leurs conseils administratifs. Et bien sûr, Slaviša JOVANOVIĆ pour son écoute et ses conseils avisés.

Pour finir, je remercie mes parents, Clarisse et Pierre, sans qui je ne serais pas arrivé jusque là. Mon frère, Jérémy, qui ne comprend toujours pas pourquoi je ne suis pas capable de réparer sa télévision, malgré mon niveau d'étude en électronique. Et enfin, Lydie, qui a toujours été à mes côtés pour me soutenir, et m'a supporté même dans les instants les plus difficiles.

Eric DABELLANI
Septembre 2013

Table des matières

Table des figures **ix**

Liste des tableaux **xiii**

Introduction générale

1 Contexte 1
2 La reconfiguration dynamique sur FPGA 2
3 Le projet ARDMAHN 3
4 Contribution 4
5 Plan du manuscrit 5

1
Traitement d'un flux vidéo

1.1 Introduction 8
1.2 Généralités 8
1.2.1 Caractéristiques d'une image 9
1.2.1.1 Codage pixel 9
1.2.1.2 Le format YUV 9
1.2.1.3 Partitionnement des pixels dans d'une image 10
1.2.2 Opérations d'adaptation d'une image 11
1.2.2.1 Modification de la résolution 11
1.2.2.2 Modification du frame rate 12
1.3 Encodage vidéo 13
1.3.1 Détection des redondances 13
1.3.1.1 Redondance spatiale 14
1.3.1.2 Redondance temporelle 14
1.3.2 Traitement des résidus d'image 16

1.3.2.1 La transformée 17
1.3.2.2 Quantification 18
1.3.3 Codage des données 19
1.3.3.1 Le codage de Huffman 19
1.4 Transcodage vidéo 20
1.4.1 Les implémentations logicielles 21
1.4.1.1 Architectures sur CPU 22
1.4.1.2 Architectures sur GPU 24
1.4.2 Les implémentations matérielles 25
1.4.2.1 Architectures sur SOC 26
1.4.2.2 Architectures sur FPGA 26
1.5 Conclusion 29

2
Techniques de modélisation de la reconfiguration dynamique

2.1 Introduction 32
2.2 La reconfiguration des FPGA 32
2.2.1 La reconfiguration statique 33
2.2.2 La reconfiguration dynamique 33
2.2.3 La reconfiguration dynamique appliquée au transcodage vidéo 35
2.3 La modélisation 36
2.3.1 Les outils de modélisation 36
2.3.2 Modélisation UML 37
2.3.3 Modélisation SystemC 38
2.3.4 L'ordonnancement des tâches reconfigurables 42
2.4 Conclusion 43

3
Méthode proposée

3.1 Introduction 46
3.2 SystemC 47
3.2.1 Niveaux d'abstraction des processus 47
3.2.2 La reconfiguration dynamique : threads statiques et dynamiques 48
3.3 Construction du modèle 50
3.3.1 Le module reconfigurable 51

3.3.2 Caractérisation des éléments du modèle 53
3.3.2.1 Caractérisation des tâches 53
3.3.2.2 Caractérisation des zones reconfigurables 53
3.3.3 La transaction 54
3.3.4 Communication entre modules 54
3.3.5 Ordonnanceur 56
3.3.6 Simulation et stratégie d'exploration 59
3.4 Modélisation de l'application : Le projet ARDMAHN 59
3.4.1 Présentation 60
3.4.2 Chaine de transcodage 61
3.4.3 Définition des cas d'utilisation 63
3.4.3.1 Les cas d'utilisation mono-flux 63
3.4.3.2 Les cas d'utilisation mutli-flux en entrée 64
3.4.3.3 Les cas d'utilisation multi-flux en sortie 65
3.4.3.4 Les cas d'utilisation multi-flux en entrée et sortie 66
3.5 Conclusion 67

4
Caractérisation des IP

4.1 Introduction 70
4.2 Architecture proposée 71
4.2.1 Modèle de l'encodeur H.264 71
4.2.2 Mise en ordre des données avant traitement 71
4.2.3 Transformée entière 71
4.2.3.1 Matrice de passage pour la DCT 71
4.2.3.2 Matrice de passage pour la Transformée entière 72
4.2.3.3 Architecture utilisée et latence 72
4.2.4 Quantification 74
4.2.4.1 Sélection des paramètres de quantification 74
4.2.4.2 Architecture utilisée et latence 76
4.2.5 Prédiction intra 77
4.2.5.1 Sélection du mode de prédiction 78
4.2.5.2 Architecture utilisée et latence 79
4.2.6 Codage de l'information 80
4.2.7 Boucle de contre réaction 81

4.2.7.1 Améliorations apportées à la boucle de retour 82
4.3 Performances 84
4.3.1 Performances temporelles 84
4.3.2 Performances en surface 85
4.4 Validation des paramètres par implémentation 86
4.4.1 Architecture standalone 87
4.4.1.1 Caractérisation des IP de transcodage (Hors décodeur H.264) 87
4.4.1.2 Caractérisation du décodeur H.264 89
4.4.1.3 Caractérisation du bloc d'adaptation 89
4.4.2 Transcodeur MPEG-2/H.264 91
4.4.3 Utilisation de la reconfiguration dynamique 92
4.5 Conclusion 95

5
Simulation et raffinement du modèle

5.1 Introduction 98
5.2 Synthèse des paramètres de l'application 98
5.2.1 L'encodage et le décodage 98
5.2.2 Les zones reconfigurables 98
5.3 Résultats de simulation 99
5.3.1 Cas d'utilisation mono-flux 100
5.3.1.1 Modèle de l'application 100
5.3.1.2 Résultats 100
5.3.1.3 Modifications possibles 101
5.3.2 Les cas d'utilisation multi-flux en entrée 104
5.3.2.1 Modèle de l'application 104
5.3.2.2 Résultats 104
5.3.2.3 Modifications possibles 106
5.3.3 Les cas d'utilisation multi-flux en sortie 108
5.3.3.1 Modèle de l'application 108
5.3.3.2 Résultats 108
5.3.3.3 Modifications possibles 109
5.3.4 Les cas d'utilisation multi-flux en entrée et sortie 111
5.3.4.1 Modèle de l'application 111
5.3.4.2 Résultats 112

5.3.4.3 Modifications possibles 112
5.4 Synthèse des résultats de simulation 115
5.5 Discussion sur la validité des résultats 116
5.6 Conclusion 118

Conclusion générale et perspectives

Annexes

Annexes **123**
1 Glossaire 124
2 Rapports de synthèse : surface occupée par l'encodeur H.264 126
3 Interfaces du transcodeur H.264/MPEG-2 130
4 Codes Microblaze 131

Publications personnelles

Bibliographie **135**

Table des figures

1 Chaîne de diffusion classique de contenus multimédias 3
2 Organisation de la thèse 5

1.1 L'image originale RGB, et les trois composantes Y, U et V. 9
1.2 Découpage d'une image en macroblocs et blocs 4x4 10
1.3 Composition d'un macrobloc 11
1.4 Image originale, et son équivalent après une opération de downscaling 12
1.5 Principe de fonctionnement d'un encodeur vidéo 13
1.6 Redondances temporelles et spatiales 13
1.7 Image de référence et résidus intra obtenus après traitement 14
1.8 Image de référence et vecteurs de mouvements obtenus après une détection des redondances temporelles 15
1.9 GOP rangé dans l'ordre d'affichage 16
1.10 GOP rangé dans l'ordre de traitement 16
1.11 Coefficients rangés par fréquence après un passage dans l'algorithme DCT . . . 17
1.12 Effet de bloc induite par la compression, à gauche l'image de référence, à droite l'image traitée 19
1.13 arbre généré par le code de Huffman pour la phrase « this is an example of a huffman tree» 20
1.14 Décomposition des étapes de transcodage 21
1.15 Architecture du transcodeur avec son module de conversion [MFA+07] 23
1.16 Architectures CPU et GPU 24
1.17 Puissance de calcul CPU et GPU [Sim] 25
1.18 Architecture interne d'un FPGA 27
1.19 Composition d'un CLB 28
1.20 Architecture du décodeur NOVA [XLGC10] 29

2.1 Méthodes de reconfiguration sur FPGA 33
2.2 Principe de la reconfiguration statique 33

2.3 Principe de la reconfiguration dynamique 34
2.4 Principe de la reconfiguration totale du FPGA, toute la surface du FPGA est modifiée 34
2.5 Principe de la reconfiguration partielle du FPGA, seule la zone reconfigurable est modifiée 35
2.6 Exemple d'instanciation d'un bus macro avec l'outil GASPARD2 [QMD08] . . 37
2.7 Flot de conception SystemC [Pol] 38
2.8 Composition d'un module reconfigurable [AKK08] 40
2.9 Cycle de vie d'un module reconfigurable [AKK08] 40
2.10 Portal utilisé dans ReChannel [RHA08] 41
2.11 Processus d'utilisation de ReChannel [RHA08] 41

3.1 Niveaux d'abstraction utilisés par SystemC 47
3.2 Analogie entre la reconfiguration dynamique et l'usage de la fonction SC_SPAWN 49
3.3 Principe de fonctionnement des wrappers 50
3.4 Principe d'alimentation du modèle 51
3.5 Module reconfigurable 52
3.6 Communication entre les élements du modèle 55
3.7 Exemple de découpage de la mémoire 55
3.8 Exemple d'attribution des priorités lors de la phase design-time 57
3.9 Exemple d'attribution des priorités lors de la phase run-time 58
3.10 Architecture du projet ARDMAHN proposée par le consortium 61
3.11 Canal de transcodage reconfigurable 62
3.12 Canal de transcodage pour un cas d'utilisation mono-flux 63
3.13 Canal de transcodage pour un cas d'utilisation multi-flux en entrée 64
3.14 Canal de transcodage pour un cas d'utilisation multi-flux en sortie 65
3.15 Canal de transcodage pour un cas d'utilisation multi-flux en entrée et sortie . . 66

4.1 Architecture de l'encodeur H.264 70
4.2 Architecture Butterfly utilisée pour la transformée entière 73
4.3 Architecture Butterfly utilisée pour la transformée entière inverse 73
4.4 Latence engendrée par la transformée entière et son inverse 74
4.5 Architecture utilisée pour la quantification et son inverse 76
4.6 Latence engendrée par la quantification et son inverse 77
4.7 Bloc prédit grâce aux pixels adjacents, notés de A à M 77
4.8 Les 9 modes de prédiction utilisés 78
4.9 Architecture utilisée pour la prédiction intra 79
4.10 Architecture utilisée pour chaque mode 80

4.11 Boucle de retour et latences induites 82
4.12 Latence induite par la boucle de contre réaction 83
4.13 Diminution de la latence grâce au traitement alterné de la luminance et de la chrominance 83
4.14 Architecture standalone 87
4.15 Bypass de CAVLC pour la caractérisation du décodeur 89
4.16 Réduction d'image par deux et par quatre 90
4.17 Architecture du transcodeur MPEG-2/H.264 91
4.18 Occupation du FPGA (Vert : Microblazes, rouge : transcodeur, bleu : autres) . . 92
4.19 Architecture du transcodeur reconfigurable 93
4.20 Implémentation de l'architecture reconfigurable. A gauche H.264, à droite MPEG-2 sont implémentés dans la zone reconfigurable 94
4.21 Occupation du FPGA (Vert : Microblazes, violet : ICAP, orange : décodeur, bleu : adaptation, rouge : zone reconfigurable) 95

5.1 Modèle de l'application mono-flux 100
5.2 Evolution des mémoires intermédiaires pour le cas mono-flux avec les paramètres actuels 101
5.3 Evolution des mémoires intermédiaires pour le cas mono-flux avec modification des paramètres des IP 102
5.4 Rappel : boucle de retour H.264 sans modification, et latence induite 103
5.5 Modèle de l'application multi-flux en entrée 104
5.6 Evolution des mémoires intermédiaires pour le cas multi-flux en entrée avec les paramètres actuels 105
5.7 Evolution des mémoires intermédiaires pour le cas multi-flux en entrée avec modification des paramètres des IP 107
5.8 Modèle de l'application multi-flux en sortie 108
5.9 Evolution des mémoires intermédiaires pour le cas multi-flux en sortie avec les paramètres actuels 109
5.10 Evolution des mémoires intermédiaires pour le cas multi-flux en sortie avec modification des paramètres des IP 110
5.11 Modèle de l'application multi-flux en entrée et sortie 112
5.12 Evolution des mémoires intermédiaires pour le cas multi-flux en entrée et sortie avec les paramètres actuels 113
5.13 Evolution des mémoires intermédiaires pour le cas multi-flux en entrée et sortie avec modification des paramètres des IP 114
5.14 Architecture avec dimensionnement des mémoires 117

1 Flot de conception proposé 121

1 Rapport de synthèse post place and root pour l'IP transformée entière 126
2 Rapport de synthèse post place and root pour l'IP prédiction intra 126
3 Rapport de synthèse post place and root pour l'IP quantification 127
4 Rapport de synthèse post place and root pour l'IP transformée inverse 127
5 Rapport de synthèse post place and root pour l'IP quantification inverse 127
6 Rapport de synthèse post place and root pour l'IP CAVLC 128
7 Rapport de synthèse post place and root pour l'IP Exp Golomb 128
8 Rapport de synthèse post place and root pour l'IP parseur 128
9 Rapport de synthèse post place and root pour l'encodeur complet 129
10 Architecture et interfaces du transcodeur H.264/MPEG-2 130

Liste des tableaux

1.1 Résultats visuels obtenus pour trois séquences vidéo [MFA+07] 22

3.1 Scénarios possibles pour un cas d'utilisation mono-flux 64
3.2 Scénarios possibles pour un cas d'utilisation multi-flux en entrée 65
3.3 Scénarios possibles pour un cas d'utilisation multi-flux en sortie 65
3.4 Scénarios possibles pour un cas d'utilisation multi-flux en entrée et sortie . . . 66

4.1 Sélection des coefficients de multiplication (Qm) en fonction de Qp pour une quantification 75
4.2 Sélection des coefficients de multiplication (Qm) en fonction de Qp pour une quantification inverse 75
4.3 Fréquences maximales obtenues pour chaque bloc de traitement 84
4.4 Exemple de débits nécessaires pour certaines résolutions 85
4.5 Ressources utilisées pour l'implémentation de chaque bloc de traitement 86
4.6 Débits en entrée et sortie des blocs de traitement pour une fréquence d'horloge à 125Mhz 88
4.7 Latence et temps de traitement du bloc d'adaptation pour différentes résolutions 90
4.8 Ressources nécessaires à l'implémentation du transcodeur 92
4.9 Ressources nécessaires à l'implémentation du transcodeur reconfigurable . . . 93

5.1 Informations connues sur les différentes tâches du transcodeur pour une fréquence de travail de 125MHz 99
5.2 Calcul de la taille de chaque frame 99
5.3 Performances maximales pouvant être atteintes pour le cas mono-flux 102
5.4 Performances maximales pouvant être atteintes pour le cas multi-flux en entrée 106
5.5 Performances maximales pouvant être atteintes pour le cas multi-flux en sortie . 111
5.6 Performances maximales pouvant être atteintes pour le cas multi-flux en entrée et sortie 115
5.7 Synthèse des performances maximales pour chaque cas d'utilisation 115
5.8 Synthèse des modifications nécessaires pour chaque cas d'utilisation 116

Introduction générale

1 Contexte

Depuis une vingtaine d'années, l'avènement du numérique se fait grandissant, le développement de l'informatique et d'internet en est la cause la plus flagrante. Cette soudaine percée est essentiellement due aux récentes avancées technologiques que nous avons connues dans le monde de l'électronique : développement des méthodes de gravure de circuit sur silicium (Loi de Moore), architectures multiprocesseurs, capacités de stockage accrues, développement des canaux de transmission et augmentation des débits internet (ADSL2+, Fibre optique), etc. Le domaine du multimédia n'est pas en reste, et les standards vidéo ne cessent d'évoluer : augmentation des résolutions d'image, films 3D, nouvelles méthodes de codage des flux, etc. L'adaptation d'un flux vidéo aux caractéristiques des réseaux de distribution et aux besoins des utilisateurs est un processus complexe demandant d'importantes ressources de calcul. Cette adaptation repose sur des techniques de décodage, de modification de données et de paramètres, puis de re-encodage. Ces différentes étapes génèrent des contraintes sur les temps de réponses, ainsi que des problèmes d'implémentation, d'intégration et de validation complexes à maîtriser.

Une possibilité pour résoudre ces problèmes de performances et de timing est de transférer une partie des fonctions d'adaptation du domaine logiciel vers le domaine matériel. Les circuits intégrés de type ASIC (Application Specific Integrated Circuit) sont capables d'atteindre des vitesses de traitement élevées grâce à une implémentation et une exécution spatiale et parallèle, contrairement aux processeurs qui eux s'exécutent séquentiellement. Toutefois, leurs architectures figées une fois implémentées accusent un cruel manque de flexibilité et d'adaptabilité. Les technologies de type FPGA présentent un bon compromis entre ces deux dernières solutions en terme de puissance de calcul. De plus, grâce à la technologie de reconfiguration dynamique, il devient possible de modifier le comportement de leurs circuits en cours d'exécution, et de gagner ainsi en flexibilité et adaptabilité.

2 La reconfiguration dynamique sur FPGA

Inventés en 1985 par la société Xilinx, le premier FPGA commercialisé ne comptait que 1500 portes logiques, sept ans plus tard, son principal concurrent ALTERA lance la famille de FPGA FLEX dont la capacité pouvait dépasser les 10 000 portes logiques. Depuis, les applications devenant de plus en plus gourmandes, la densité des FPGA n'a cessée de grandir, pour arriver aujourd'hui à des capacités dépassant le million de portes logiques. Pour tenter de palier ce besoin croissant de ressources, la technologie de reconfiguration dynamique a été introduite en 1997 par Xilinx avec la commercialisation du premier FPGA clairement adapté à cette pratique. Le XC6200 [XIL96], comptait 85 000 portes logiques et autorisait pour la première fois la reconfiguration partielle.

Jusqu'à cette date, la configuration des FPGA ne pouvait se faire qu'une fois le système à l'arrêt, l'architecture implémentée ne pouvait être modifiée pendant la phase d'exécution. La reconfiguration dynamique comble ce manque en autorisant la modification d'une partie prédéfinie du FPGA, sans perturber ou arrêter le reste du design. En plus du gain en termes de flexibilité, cette technique permet également de réduire la surface occupée sur le FPGA [Kao05] et sa consommation électrique [DT05, PHBB07], puisqu'à n'importe quel moment, il est possible de retirer une tâche non utilisée pour la remplacer par une autre.

Malgré des avantages certains, la reconfiguration dynamique sur FPGA peine à être utilisée par les industriels. Le manque de méthodes et de moyens mis à disposition des développeurs en est la cause principale. Sa mise en oeuvre demande un effort supplémentaire considérable par rapport à une solution statique. Aucun outil officiel ne permet actuellement de prédire le comportement d'une architecture reconfigurable, il est impossible de connaître à l'avance la faisabilité et les ressources nécessaires à sa mise en oeuvre. La validation doit se faire directement sur carte au cours du développement de l'application, et ce n'est qu'au moment de la phase d'implémentation et de tests, que l'on découvre si l'architecture développée respecte bien les contraintes de temps imposées par l'application.

L'ordonnancement de la reconfiguration est un autre point bloquant, le fait de pouvoir modifier le comportement du FPGA à tout moment de l'exécution ouvre des possibilités de scénarios quasi illimités. Chacun d'entre eux impose ses propres besoins et contraintes, l'architecture doit être dimensionnée en fonction de celui-ci. Or, sans simulation en amont, il est impossible de déterminer au préalable le scénario le plus adapté, et les ressources nécessaires à son déroulement.

Pour ces raisons, la technique de reconfiguration dynamique sur FPGA reste très peu utilisée par les industriels. Néanmoins, ce thème de recherche reste majeur dans la communauté scientifique, et beaucoup de projets dans ce domaine sont proposés chaque année. C'est notamment le cas du projet ARDMAHN [ARD] (Architecture Reconfigurable Dynamiquement pour

l'Auto-adaptation en Home Networking), qui propose d'utiliser la reconfiguration dynamique sur FPGA dans une application multimédia.

3 Le projet ARDMAHN

Le projet ARDMAHN s'intègre dans ce contexte multimédia et propose d'étudier les problématiques d'adaptabilité liées à la diffusion et au traitement de flux vidéo.

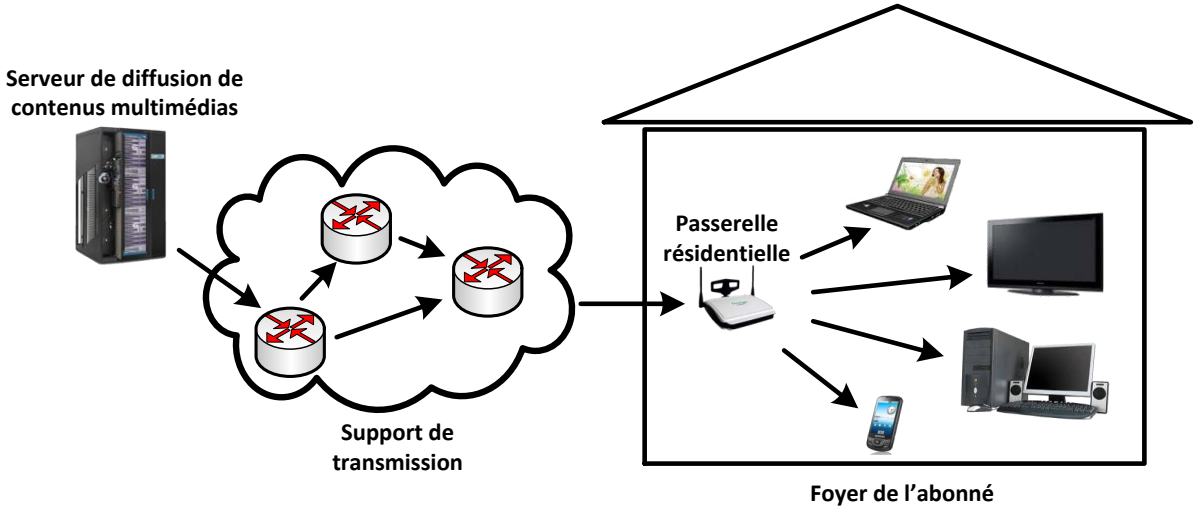


FIGURE 1 – Chaîne de diffusion classique de contenus multimédias

Avant d'arriver jusqu'à l'utilisateur final, un flux vidéo doit passer par tout un réseau de distribution bien coordonné (figure 1). Chez l'utilisateur, les appareils de réception et de distribution des flux, aussi appelés « passerelles résidentielles » ou « set-top box », sont au cœur de ce réseau, ils ont pour rôle de réaliser la jonction entre le support de transmission et le terminal de lecture de l'abonné. Ces deux éléments étant en constante évolution, la set-top box est amenée à être améliorée au même rythme afin de conserver l'homogénéité du réseau de distribution. Malheureusement, les contraintes en terme d'adaptabilité sont telles, qu'il devient de plus en plus compliqué de suivre ces évolutions. De nouvelles approches doivent être trouvées.

Au delà du soucis de flexibilité, qui jusque là pouvait être maîtrisé grâce à des solutions logicielles, c'est un besoin grandissant de puissance de calcul qui se fait ressentir. En effet, les traitements vidéo réalisés sont de plus en plus complexes et demandent toujours plus de ressources aux calculateurs.

La solution que propose d'étudier le projet, est de migrer une partie des fonctions d'adaptation jusqu'à maintenant réalisées de façon logicielle, vers des accélérateurs matériels de type FPGA reconfigurable dynamiquement. Au delà de l'enjeu commercial que représente la réali-

sation d'une telle architecture, le projet contribue pleinement à la recherche, en proposant des méthodologies de vérifications et de simplifications pour la mise en œuvre de la reconfiguration dynamique.

La finalité du projet est de proposer à la vente, une solution de transcodage vidéo auto-adaptable capable de traiter plusieurs flux vidéo parallèlement, tout en prenant en compte les standards de codage MPEG-2 et H.264. Une solution simple à mettre en oeuvre, aurait été d'implémenter autant de chaînes de transcodage qu'il y a de flux vidéo à traiter. Toutefois, par soucis de rendre la passerelle résidentielle abordable et concurentielle financièrement, une telle solution n'est pas envisageable. Pour cette raison, le consortium ARDMAHN a décidé d'étudier et d'utiliser la technique de reconfiguration dynamique sur FPGA, qui permet la mutualisation et la réutilisation des ressources entre reconfigurations successives, et par conséquent de réduire la surface nécessaire et le coût des FPGA.

4 Contribution

L'objectif de ces travaux de recherche est de proposer une méthodologie de modélisation et de simulation d'architectures reconfigurables dynamiquement. Notre outil, écrit dans le langage SystemC, permet de simuler n'importe quelle architecture pour prédire les besoins en ressources de celle-ci, et de valider sa faisabilité. Pour imiter au mieux le comportement de la reconfiguration dynamique, et modifier une tâche au cours de l'exécution de la simulation, nous utilisons une des particularité du langage SystemC : les threads dynamiques.

La recherche de scénarios d'ordonnancement des tâches reconfigurables, adaptés aux contraintes de l'application est une étape complexe mais primordiale. Notre outil en permet l'exploration, et propose une estimation des ressources à allouer à l'architecture, dans le respect des contraintes temps réelles fixées par l'application.

Pour diminuer leurs coûts de développement, les entreprises font de plus en plus souvent appel à des bibliothèques d'IP. Achetées à des sociétés spécialisées, ou récupérées sur d'anciens projets, ces bibliothèques permettent de réduire considérablement le temps de mise sur le marché de nouveaux produits, et leurs coûts de développement. Bien qu'avantageux, il peut arriver que ces IP ne soient pas adaptées ou appropriées au projet en cours. Notre méthodologie passe par une caractérisation et une étude précise des IP à modéliser, afin de valider leurs performances et leur utilisation au sein de l'application.

Pour valider notre méthode sur un cas réel, un bibliothèque d'IP de codage vidéo nous a été fournie par notre partenaire VIOTECH. En complément, d'autres IP ont dues être développées et seront détaillées. Cette application consiste en la réalisation d'un transcodeur H.264/MPEG-

2, rendu auto-adaptable grâce à l'utilisation de la reconfiguration dynamique.

5 Plan du manuscrit

Pour être réaliste, notre modèle est alimenté par les informations provenant de l'application, et prend en compte les contraintes liées à la reconfiguration dynamique. Pour les besoins de notre étude, nous avons donc travaillé en parallèle sur trois points importants (figure 2) :

- La réalisation du modèle de la reconfiguration dynamique en langage SystemC.
- Le développement et la caractérisation d'IP de codage vidéo.
- La conception d'une architecture reconfigurable sur FPGA pour l'implémentation de notre application.

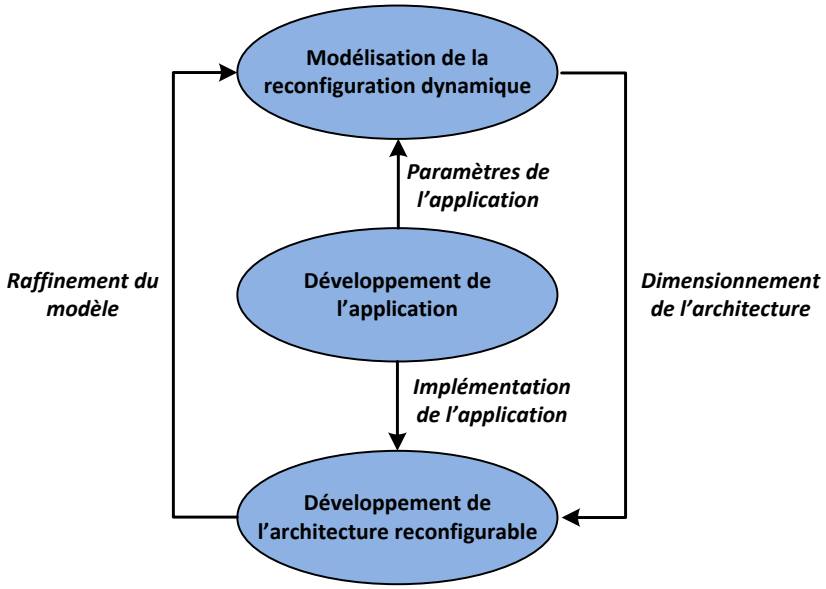


FIGURE 2 – Organisation de la thèse

Les trois parties sont dépendantes : le modèle est alimenté par les paramètres issus de l'application et de l'architecture reconfigurable, et l'architecture est conçue grâce aux IP développées et aux résultats de simulation de notre modèle.

Le manuscrit est structuré en cinq chapitres :

Le premier chapitre présente les notions de transcodage vidéo : nous rappellerons les traitements nécessaires lors du codage et du décodage d'un flux. Nous verrons ensuite des travaux d'implémentations de transcodeur déjà réalisés sur des cibles logicielles et matérielles.

Le second chapitre sera dédié à la reconfiguration sur FPGA. Nous commencerons par rappeler les différents types de reconfiguration que l'on peut rencontrer. Puis nous verrons les

travaux disponibles dans la littérature sur les techniques de modélisation.

Dans le troisième chapitre, nous proposerons notre méthodologie, et reviendrons sur le projet ARDMAHN et l'architecture d'un canal de transcodage. Ce dernier constituera l'application de test pour notre modèle.

Le quatrième chapitre revient sur l'architecture du canal de transcodage plus en détails. Il est question ici de présenter nos travaux de développement de l'application, et de caractériser chaque IP pour les besoins de notre modèle. Nous verrons également le développement de l'architecture reconfigurable et les performances obtenues.

Le cinquième chapitre présente les résultats de simulation obtenus avec notre modèle, ainsi que les modifications proposées pour améliorer les performances du canal de transcodage.

Enfin, nous terminerons par la conclusion générale de ce manuscrit et ferons un point sur les perspectives ouvertes par ces travaux.

Chapitre 1

Traitement d'un flux vidéo

Sommaire

1.1 Introduction 8

1.2 Généralités 8

1.2.1 Caractéristiques d'une image 9

1.2.2 Opérations d'adaptation d'une image 11

1.3 Encodage vidéo 13

1.3.1 Détection des redondances 13

1.3.2 Traitement des résidus d'image 16

1.3.3 Codage des données 19

1.4 Transcodage vidéo 20

1.4.1 Les implémentations logicielles 21

1.4.2 Les implémentations matérielles 25

1.5 Conclusion 29

1.1 Introduction

Avec le déploiement des derniers terminaux de lecture multimédia (Smartphone, télévision HD, tablette, etc), les caractéristiques des flux vidéo ne cessent de se diversifier : nouvelles résolutions, augmentations du nombre d'images par seconde, nouveaux types de codage. Ce renouvellement constant des standards engendre des problèmes d'adaptabilité sur le réseau de transmission, il peut arriver par exemple, qu'un appareil de lecture ne soit pas compatible avec le flux vidéo reçu. Une étape d'adaptation est alors nécessaire pour modifier le flux, et le rendre compréhensible par le terminal, on parle dans ce cas de "transcodage vidéo".

Dans ce chapitre, nous commencerons par rappeler quelques notions de vidéo, avant de voir plus en détails les processus et traitements nécessaires à l'encodage d'un flux. Pour finir, nous ferons un point sur différents travaux de transcodage, et les méthodes d'implémentations sur architectures logicielles et matérielles.

1.2 Généralités

Depuis l'arrivée dans le commerce des premières télévisions à écran plat, la résolution des flux vidéo ne cesse d'augmenter. Les premiers terminaux grand public de ce type étaient capables d'afficher des résolutions d'image dites 720p correspondant à une définition de 1280x720 pixels, puis quelques années plus tard, des résolutions Full HD ou 1080p correspondant à une résolution de 1920x1080 pixels. Et au moment où je rédige ce mémoire, une nouvelle résolution s'apprête à voir le jour dans le commerce, celle-ci, baptisée 4K atteint 4096x2160 pixels, soit quatre fois plus de pixels que pour la Full HD.

Cette montée en résolution implique une expansion significative de la taille des fichiers vidéo. A titre indicatif, une image Full HD représente 2 073 600 pixels, chacun codé sur 24 bits. A raison de 60 images par seconde, la transmission d'un tel flux vidéo nécessiterait un canal de transmission possédant une bande passante de l'ordre de 3 Gbits/s. Or, les technologies actuelles ne permettent pas de transmettre des données jusqu'à l'utilisateur à cette vitesse. Quelques chiffres :

- Modem : 56 Kbits/s
- ADSL : 384 Kbits/s
- ADSL2+ : 25 Mbits/s
- VDSL : 100 Mbits/s
- FTTH : 1 Gbits/s

Comme on peut le constater, même un utilisateur disposant d'une connexion fibre optique (FTTH) sans aucune perturbation ni atténuation, ne pourrait recevoir un flux Full HD brut. Il est alors nécessaire de compresser ce flux avant de le diffuser sur un canal de transmission. D'après une étude réalisée fin 2012 [Dou], le débit moyen constaté en France chez les usagés d'internet est de 5,6 Mbit/s. Il faudrait alors compresser le flux Full HD avec un facteur 533 pour permettre une transmission correcte de celui-ci.

1.2.1 Caractéristiques d'une image

1.2.1.1 Codage pixel

Une image numérique, peut être représentée comme une matrice à deux dimensions constituée par une succession de pixels possédant chacun une coordonnée X et une coordonnée Y. Sur un écran plat, chaque pixel représente un point lumineux. Et est caractérisé par trois informations déterminant le niveau d'intensité des trois couleurs primaires : rouge, vert et le bleu (RGB). La superposition de ces trois composantes, chacune codée généralement sur 8 bits (256 nuances de couleurs) permet de proposer plus de seize millions de combinaisons de couleurs différentes par pixel.

1.2.1.2 Le format YUV

Le standard YUV est une autre méthode de représentation dédiée à la transmission vidéo. Tandis que le format RGB code chaque pixel grâce aux trois composantes de base, YUV lui, détermine un pixel par son niveau de gris (Luminance : Y) et deux composantes couleurs (Chrominance : U et V). L'œil humain étant plus sensible à la luminosité d'une image plutôt qu'à sa couleur, il est possible de réduire la taille des chrominances. Les composantes U et V demandent quatre fois moins d'informations que la composante Y pour être codées.(figure 1.1)

FIGURE 1.1 – L'image originale RGB, et les trois composantes Y, U et V.

Les trois nouvelles composantes sont directement issues des composantes rouges, vertes et bleues et peuvent être retrouvées grâce aux équations 1.1.

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ U &= -0.147R - 0.289G + 0.436B = 0.492(B - Y) \\ V &= 0.615R - 0.515G - 0.100B = 0.877(R - Y) \end{aligned} \tag{1.1}$$

Le format YUV a été mis en place pour faire la transition entre les équipements vidéo affichant la couleur, et ceux plus anciens ne l'affichant pas encore, puisqu'il suffit de ne transmettre que la composante Y pour obtenir l'image noir et blanc. Aujourd'hui encore, ce format est resté le standard dans les systèmes de diffusion télévisuelle.

1.2.1.3 Partitionnement des pixels dans d'une image

Avant de parler de codage et des différents algorithmes de compression vidéo, il est utile de mettre en avant des différentes façons de partitionner les pixels dans une image (figure 1.2). Comme nous le verrons par la suite, les algorithmes de codage classiques ne travaillent pas directement sur un élément aussi petit que le pixel, mais sur des groupes de pixels, les macroblocs (MB).

FIGURE 1.2 – Découpage d'une image en macroblocs et blocs 4x4

Chaque macrobloc est représenté par 256 pixels (16x16) pour la luminance, ainsi que 64 pixels (8x8) pour chaque chrominance (figure 1.3). En fonction du codec vidéo utilisé, les macroblocs peuvent ensuite être divisés en éléments plus petits.

Ainsi, pour le codec MPEG-2, dont les algorithmes de traitement travaillent sur des blocs 8x8, les macroblocs doivent être divisés en quatre blocs de luminance et deux blocs de chromi-

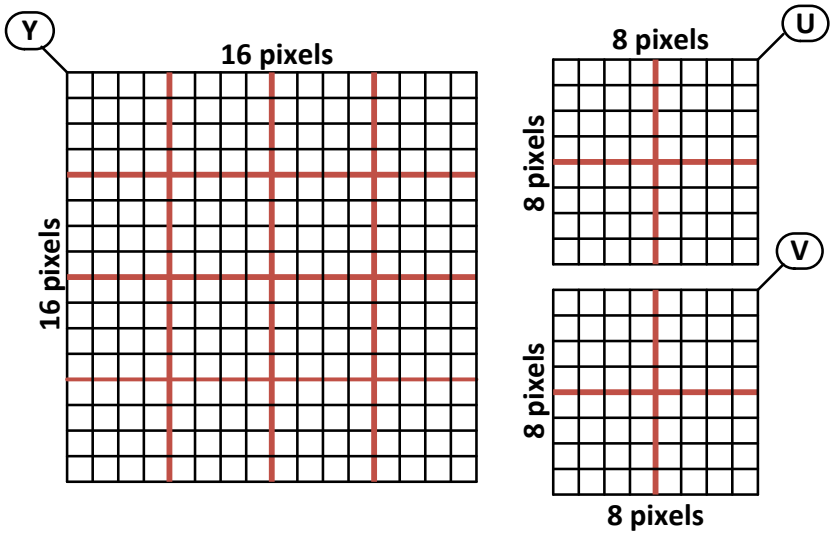


FIGURE 1.3 – Composition d'un macrobloc

nance. Alors que pour le codec H.264, le macrobloc doit être découpé en seize blocs 4x4 pour la luminance et en huit blocs pour la chrominance.

1.2.2 Opérations d'adaptation d'une image

Dans certain cas, le flux vidéo et l'équipement de lecture ne possèdent pas les mêmes caractéristiques, il est alors nécessaire de réaliser une adaptation des paramètres vidéo. Ces modifications sont effectuées sur le flux brut (non codé, ni compressé) au format YUV.

1.2.2.1 Modification de la résolution

L'adaptation la plus fréquente consiste à modifier la résolution de l'image (figure 1.4). On parle d'upscaling pour un agrandissement, et de downscaling pour une réduction.

La procédure d'upscaling consiste à générer de nouveaux pixels grâce aux valeurs adjacentes. Le nombre de pixels ajoutés dépend bien sûr du ratio entre la résolution de l'image d'origine, et la résolution du terminal de lecture : plus ce ratio sera élevé, et plus le nombre de pixels à ajouter sera grand. Le plus souvent, les pixels générés sont directement une duplication des pixels précédents, le fait d'avoir deux pixels de même valeur l'un à coté de l'autre revient au final à créer un seul pixel deux fois plus gros. Si cette opération ne pose pas de problème pour un faible upscaling, elle risque de dégrader fortement la qualité visuelle pour le téléspectateur lors d'un agrandissement plus élevé.

Pour le downscaling, il s'agit de réaliser l'opération inverse, et de fusionner les blocs pour

FIGURE 1.4 – Image originale, et son équivalent après une opération de downscaling

en créer de nouveaux. En général cette opération consiste à réaliser une moyenne arithmétique entre les valeurs des coefficients utiles.

1.2.2.2 Modification du frame rate

Le frame rate, ou FPS (Frames per second) est l'unité de mesure permettant de déterminer le nombre d'images par seconde contenues dans une vidéo. Plus le FPS est élevé, et plus l'animation de la vidéo semblera fluide. Comme la résolution d'image, ce paramètre est étroitement lié au type de terminal de lecture. Au cinéma par exemple, le frame rate est normalisé à 24 images par seconde, alors que pour la télévision, la norme est de 25 images par seconde en Europe, contre 30 au Japon et aux Etats-Unis. Mais avec les derniers appareils de lecture vidéo, il n'est pas rare de trouver des terminaux capables de lire des vidéos à 30, 60 et même 100 images par seconde.

La diminution du frame rate consiste à supprimer certaines images dans le flux vidéo, à condition toutefois de ne par descendre au dessous de 16 images par seconde, correspondant à la cadence minimale avant l'observation de saccades lors de l'affichage.

L'augmentation du frame rate est quant à lui plus complexe, l'ajout d'images dans le flux nécessite une étude approfondie des mouvements visuels d'une image à une autre. Ces mouvements peuvent être vu comme une translation de pixels, également appelés vecteurs de mouvements. Un objet en mouvement vers la droite par exemple, résulte d'une translation d'un ensemble de pixels dans cette direction. La création d'une image est réalisée grâce à la détermination de ces vecteurs de mouvements, et au calcul de vecteurs intermédiaires.

1.3 Encodage vidéo

Comme nous l'avons vu dans le paragraphe précédant, la compression et l'encodage d'un flux vidéo est primordiale avant d'autoriser son émission sur un support de transmission. Pour cela, plusieurs étapes sont nécessaires (figure 1.5) : la détection des redondances (présentes dans une image, ou dans des images voisines), la transformation de celles-ci, leur quantification, et enfin leur codage.

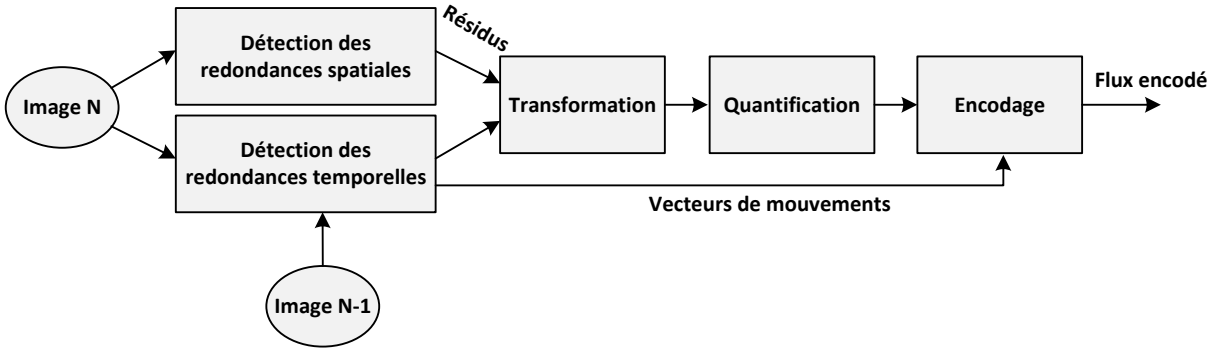


FIGURE 1.5 – Principe de fonctionnement d'un encodeur vidéo

Quelque soit le standard utilisé, la méthode de compression reste identique et suit ce même schéma, seule la complexité et l'efficacité des algorithmes utilisés diffèrent d'un encodeur à un autre. Dans la suite de ce paragraphe, nous verrons plus en détail chacune de ces étapes.

1.3.1 Détection des redondances

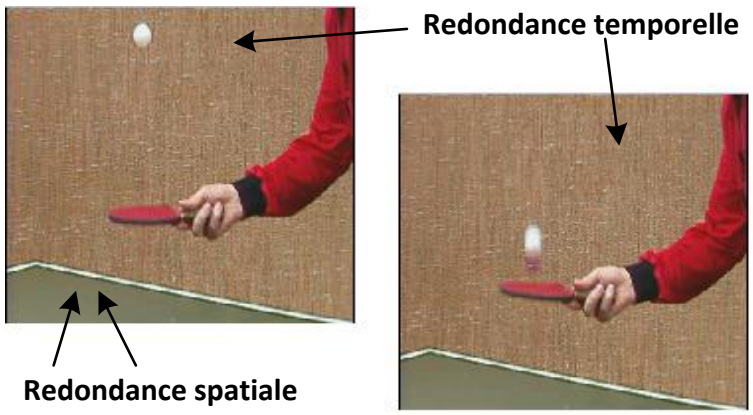


FIGURE 1.6 – Redondances temporelles et spatiales

La détection des redondances dans un flux vidéo constitue la première étape vers sa compression. Au lieu de traiter l'intégralité des données d'une image ou d'un flux vidéo, les pixels ou groupe de pixels possédant les mêmes attributs sont regroupés pour n'être traités qu'une seule fois, réduisant considérablement le nombres d'informations à coder. Les redondances peuvent être regroupées en deux catégories : les redondances spatiales et les redondance temporelles (figure 1.6).

1.3.1.1 Redondance spatiale

Il peut arriver que des groupes de pixels soient identiques dans une image. Dans un ciel bleu par exemple, chaque pixel aura une valeur proche de celle de ses voisins, il s'agit de la redondance spatiale.

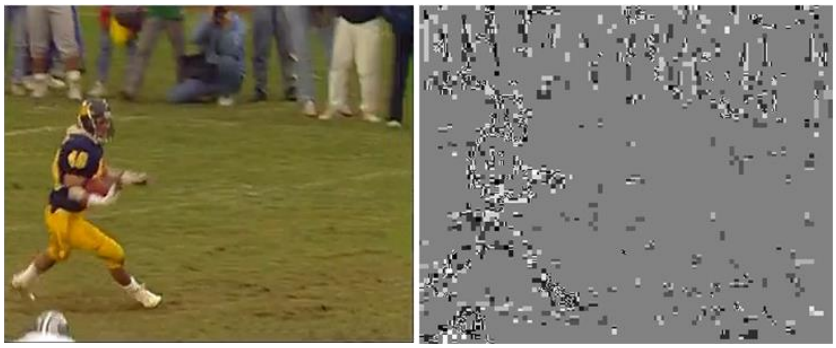


FIGURE 1.7 – Image de référence et résidus intra obtenus après traitement

L'algorithme de prédiction intra est en charge du traitement de ce type de redondance, il soustrait la valeur de chaque pixel avec la valeur des pixels adjacents. Ainsi, ce n'est pas le pixel qui va être codée, mais une différence (aussi appelée résidu intra). Celle-ci peut être proche de zéro en cas de forte ressemblance entre l'image d'origine et l'image prédite (figure 1.7), ce qui est particulièrement intéressant pour l'encodeur et le taux de compression final.

1.3.1.2 Redondance temporelle

Dans le cas de la redondance temporelle, il s'agit de trouver des corrélations entre images successives. Par exemple lors d'un plan fixe, le décor ne varie pas, il est inutile de coder plusieurs fois les mêmes informations.

L'image est en fait divisée en plusieurs blocs de pixels, pour chacun d'entre eux, l'algorithme va rechercher des blocs similaires dans les images voisines. Si une correspondance est

FIGURE 1.8 – Image de référence et vecteurs de mouvements obtenus après une détection des redondances temporelles

trouvée, un vecteur de mouvements reliant les deux blocs est généré. Puis, comme pour le traitement des redondances spatiales, un algorithme de prédiction inter réalise la différence entre ces deux blocs et transmet le résidu inter, ainsi que les vecteurs de mouvements (figure 1.8).

Les corrélations temporelles sont recherchées dans des groupes d'images plus ou moins nombreuses que l'on nomme GOP (Group Of Pictures). Chaque GOP est constitué de trois types d'images :

- Les images I représentent les images de référence : elles servent de source pour la recherche de redondances. Étant donné qu'il s'agit d'images sources (non prédite), ce sont elles qui contiennent le plus d'informations. Chaque GOP ne contient qu'une seule image I qui synchronise le décodeur.
- Les images P sont préditent grâce à l'image de référence I, puis servent elles mêmes de références pour les prochaines images P et B.
- Les images B sont des images bidirectionnelles, elles sont prédites à partir des images I et P précédentes et suivantes. Ce sont les seules images qui ne peuvent pas être utilisées en tant que références.

Sur la figure 1.9, les images B sont prédites à l'aide des images I et P passées, mais également futures alors qu'elles mêmes ne devraient pas être encore prédites. Pour rendre cette opération possible, les encodeurs intègrent un séquenceur qui modifie l'ordre d'envoi des images à l'encodeur, ainsi, les images ne sont pas traitées dans l'ordre dans lequel elles seront affichées (figure 1.10).

Le choix de la taille du GOP est un paramètre important qui va influencer la qualité de l'image et le taux de compression du flux de sortie. Chaque GOP débute par une image de référence I, qui comme nous l'avons vu, est la seule image qui ne sera pas compressée par les

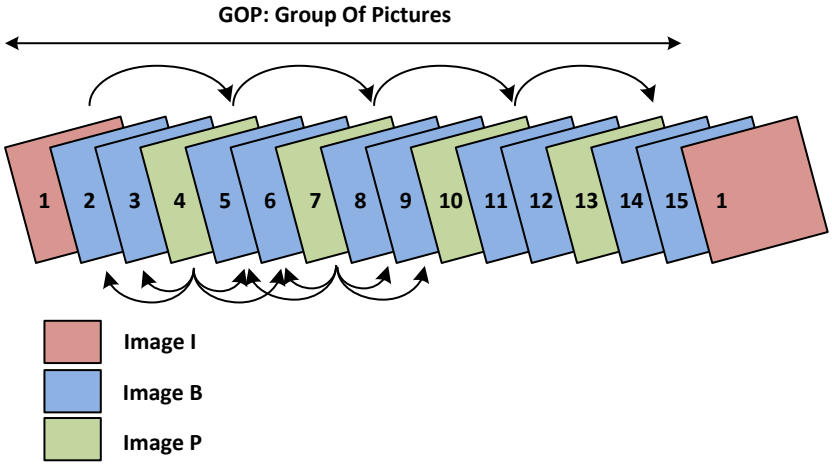


FIGURE 1.9 – GOP rangé dans l'ordre d'affichage

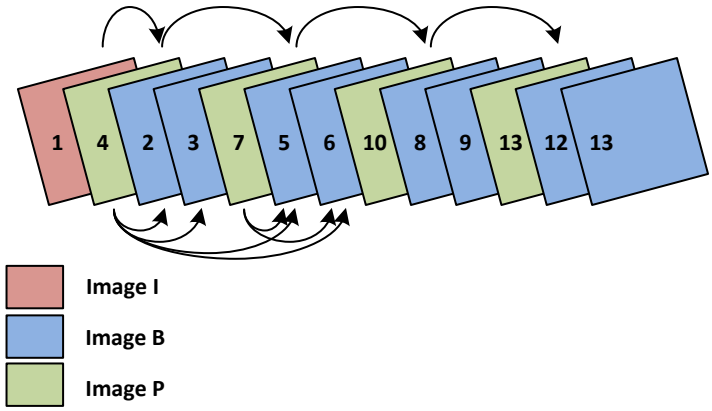


FIGURE 1.10 – GOP rangé dans l'ordre de traitement

algorithmes de détection de redondances. Si une erreur est commise durant le GOP, celle-ci sera perpétuée sur l'ensemble du GOP (principe de la prédiction), mais sera corrigée sur le suivant lorsqu'une nouvelle image de référence sera introduite. Un GOP trop long augmente les probabilités de voir une erreur d'affichage trainer sur plusieurs images consécutives. Alors qu'un GOP plus court augmentera la répétition des images de référence I et influera directement sur la taille du flux de sortie.

1.3.2 Traitement des résidus d'image

Maintenant que les redondances ont été repérées, la seconde étape de l'encodage vidéo consiste à filtrer et réduire les résidus créés par la prédiction. Cette opération est réalisée par les

algorithmes de transformée et de quantification.

1.3.2.1 La transformée

Mathématiquement, la transformée effectue un changement de représentation des données pour les rendre plus manipulables. Une transformation de base, par exemple, consiste à regrouper les coefficients importants entre eux pour faciliter leur traitement par la suite.

Il existe plusieurs algorithmes de transformée liés aux traitements d'images, mais seulement deux d'entres eux retiendront notre attention, il s'agit de la transformée en cosinus discrète (DCT) exploitée dans les standards de compression d'images JPEG et vidéo MPEG, et la transformée entière (TE), qui est une évolution de la DCT utilisée dans la norme H.264.

La transformée en cosinus discrète n'est pas une technique récente, elle a été développée sur la base de la transformée de Fourier discrète et sert depuis les années 80. Son principe est simple : réorganiser les informations contenues dans une matrice de pixels en séparant les basses et les hautes fréquences. On entend par "fréquence", la variation de valeurs des coefficients. Les basses fréquences déterminent les zones de l'image pour lesquelles il y a une grande homogénéité des coefficients, alors que les hautes fréquences correspondent aux changements brusques entre coefficients voisins. (figure 1.11).

FIGURE 1.11 – Coefficients rangés par fréquence après un passage dans l'algorithme DCT

Cette séparation des fréquences est utile pour la suite du traitement des données. On sait que l'œil humain a une faible sensibilité aux hautes fréquences, ces informations peuvent donc être supprimées pour augmenter le taux de compression sans grande dégradation visuelle de l'image. L'algorithme DCT (équation 1.2) est totalement réversible, ce qui permet de retrouver les coefficients initiaux grâce à l'opération inverse IDCT (équation 1.3). Si on considère une image I(m,n) de dimension m x n pixels et d'intensité x(m,n). Et X(u,v), les coefficients obtenus par l'application de la DCT sur l'image entière, où u et v sont les fréquences spatiales de l'image

après le traitement.

$$X_{(u,v)} = \frac{2}{\sqrt{M.N}} C_u.C_v \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{(m,n)} \cos \frac{(2m+1).\pi.u}{2.M} \cos \frac{(2n+1).\pi.v}{2.N} \tag{1.2}$$

$$x_{(m,n)} = \frac{2}{\sqrt{M.N}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} C_u.C_v.X_{(u,v)} \cos \frac{(2m+1).\pi.u}{2.M} \cos \frac{(2n+1).\pi.v}{2.N} \tag{1.3}$$

M et N correspondent aux dimensions des matrices sur lesquelles la DCT est appliquée. L'algorithme n'est pas adapté au traitement de signaux non stationnaires comme c'est le cas dans une image. Il est donc nécessaire de travailler sur des blocs de plus petites tailles, en général 8x8 pixels, minimiser la surface de traitement permet de réduire les risques d'avoir de grandes variations d'amplitude dans les coefficients.

La DCT occupe une place importante dans la compression vidéo, et a évoluée avec la sortie du codec H.264 en 2003. La Transformée entière reprend le même principe utilisé jusqu'alors, à quelques différences :

- Le codec H.264 travail dorénavant sur des blocs 4x4 pixels contrairement à MPEG-2 qui lui ne traite que des blocs 8x8.
- Le nouvel algorithme ne nécessite plus d'opérateur de division comme c'était le cas avec la DCT, le résultat n'est plus sur virgule flottante, mais sur des entiers. Il n'y a donc plus d'erreur due à l'approximation de la virgule lors de la reconstruction du bloc d'origine.
- L'implémentation est bien moins complexe puisque les opérateurs de division et de multiplication ont été supprimés au profit d'additionneurs.

1.3.2.2 Quantification

La quantification permet de supprimer l'ensemble des informations jugées non pertinentes. Son fonctionnement est comparable à un filtre numérique dont la fréquence de coupure serait définit par un coefficient de quantification (QP). Comme il s'agit d'un traitement avec perte de données, plus le seuil appliqué sera élevé et plus le taux de compression le sera également, en contre partie de dégradations visuelles irréversibles (figure 1.12), on parle alors "d'effet de blocs".

Pour minimiser les erreurs induites par la quantification, certains standards utilisent un traitement supplémentaire : le deblocking filter. Son principe est simple, il compare et lisse les valeurs de pixels voisins, l'effet de bloc est alors atténué, et le rendu visuel est significativement amélioré.

FIGURE 1.12 – Effet de bloc induite par la compression, à gauche l'image de référence, à droite l'image traitée

1.3.3 Codage des données

Le codage des données constitue la dernière étape du flot de traitement et de compression de l'encodeur. Il s'agit de mettre les informations sous une autre forme en fonction de leurs statistiques d'apparition. Par exemple, un symbole (dans notre cas, la valeur d'un pixel) ayant une forte présence dans une séquence, et donc une forte probabilité d'apparition, se verra attribuer un code très court. A contrario, un symbole n'apparaissant que peu de fois, aura un code plus long. L'avantage d'utiliser ce type de codage à longueur variable (également appelé codage entropique) est d'obtenir un taux de compression bien plus important qu'avec un codage à longueur fixe. En contre partie, celui-ci est plus sensible aux erreurs de transmission.

1.3.3.1 Le codage de Huffman

Parmi les différents codages entropiques existants, le code de Huffman nous intéresse particulièrement. En plus d'être largement utilisé dans les domaines de la compression et de la théorie de l'information, nous le retrouvons dans les standards MPEG-2 et H.264.

La méthode de Huffman consiste à créer un arbre de codage, dont chaque feuille serait représentée par un symbole (pixel), disposé selon le nombre de fois qu'il est cité dans le signal de référence (une image par exemple). Les feuilles les plus hautes correspondent aux symboles les moins utilisés. Si plusieurs feuilles possèdent le même poids, elles sont additionnées deux par deux pour former un nœud, ce nœud pourra lui même être additionné avec un autre nœud ou feuille de même poids. Et ainsi de suite jusqu'à constituer l'arbre entièrement. Pour finir, chaque feuille et nœud additionné se verra attribuer aléatoirement un identifiant '0' ou '1'. Le code de n'importe quel symbole peut alors être retrouvé en lisant les identifiants de la racine jusqu'à la feuille. Un exemple d'arbre de Huffman est donné dans la figure 1.13.

Il existe plusieurs variantes du code de Huffman. Nous retrouvons par exemple le codage

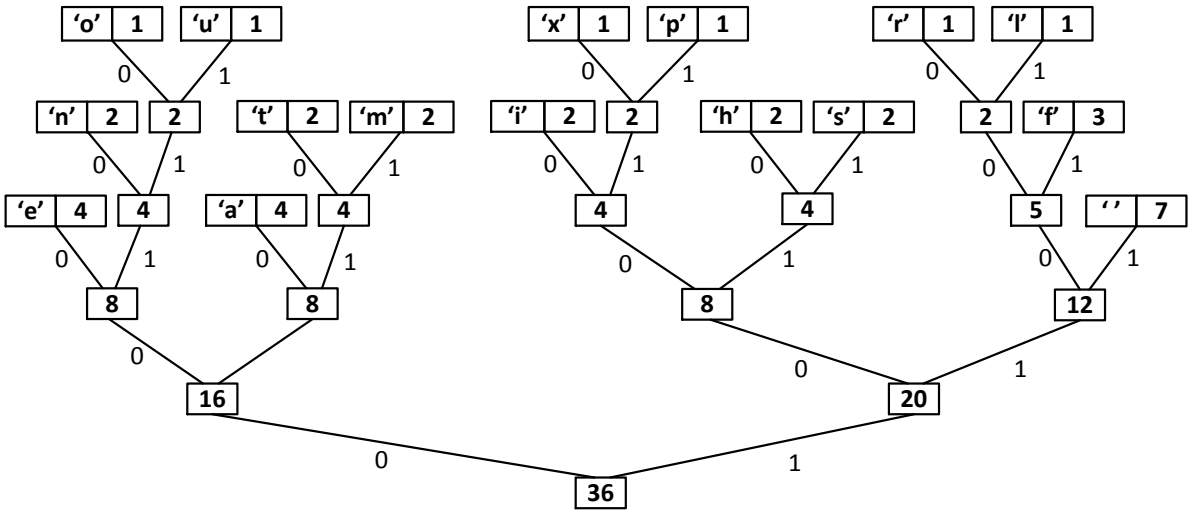


FIGURE 1.13 – arbre généré par le code de Huffman pour la phrase « this is an example of a huffman tree»

VLC (Variable Length Coding) utilisé par MPEG-2 qui fonctionne de la manière décrite précédemment. Une table de codes est construite pour coder chacun des symboles répertoriés dans l'image source, cette table est ensuite transmise au décodeur, qui s'en servira pour retrouver les informations initiales. Comme toutes les dérivées du codage de Huffman, celle-ci est sans perte de données.

Le codec H.264 utilise une variante plus efficace, il s'agit de CAVLC (Context-based Adaptative Variable Length Coding), qui comme son nom l'indique, adapte ses tables de codage en temps réel en fonction des informations reçues, c'est-à-dire que l'arbre de Huffman est en constante modification. Ce procédé est intéressant pour le traitement vidéo Full HD, car il n'est plus nécessaire de stocker une séquence vidéo complète pour déterminer la fréquence d'apparition de chaque symbole. En contre partie, la construction dynamique de la table demande plus de calculs et de temps d'exécution.

1.4 Transcodage vidéo

Le transcodage vidéo sert à adapter les paramètres d'un flux vidéo aux caractéristiques du terminal de lecture. Cette adaptation peut être nécessaire au niveau de la résolution d'image et du frame rate comme nous l'avons déja vu, mais également sur le type de codage utilisé.

Le transcodage est réalisé en trois étapes (figure 1.14) :

– L'analyse du flux entrant et l'utilisation d'un décodeur adapté représente l'étape de déco-

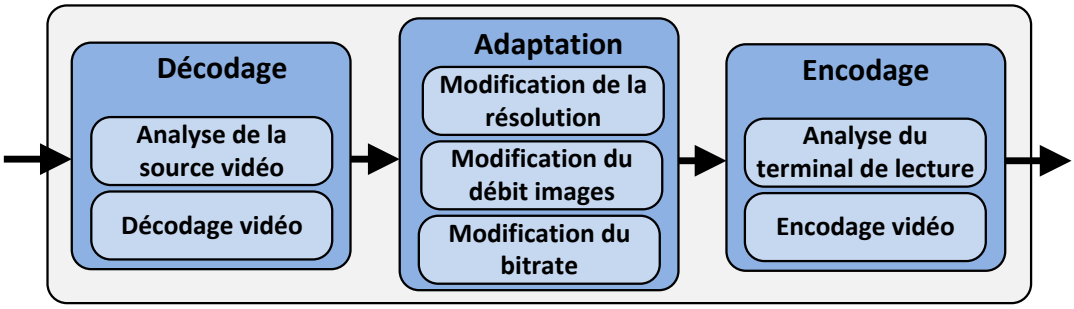


FIGURE 1.14 – Décomposition des étapes de transcodage

dage. Cette étape a pour but de générer un flux brut YUV pouvant être traité par le bloc d'adaptation.

– A l'opposé, l'étape d'encodage analyse le terminal de lecture pour déterminer le codage pris en charge par celui-ci. En fonction de cela, l'encodeur approprié est choisi.
– Entre ces deux parties, l'étape d'adaptation réalise les modifications de résolution et de frame rate pour s'adapter au canal de transmission. En règle générale, il est également possible de modifier le débit binaire (bitrate) du flux vidéo, plusieurs méthodes permettent cette opération, comme par exemple la modification du coefficient de quantification (dégradation non réversible de l'image). Toutefois, cette pratique (modification du bitrate) n'a pas été retenue par le consortium du projet ARDMAHN, elle ne fera donc pas partie de notre étude.

Un transcodeur complet et performant doit pouvoir décoder n'importe quel type de flux en entrée, et être capable de l'encoder quelque soit le terminal de lecture. Ce qui nécessite une disponibilité de tous les codecs existants, au sein de son architecture.

Dans la plupart des applications de transcodage que l'on retrouve dans le commerce, le traitement vidéo est réalisé par des calculateurs logiciels. Toutefois, aux vues de la complexité grandissante des algorithmes, de nombreux travaux sont menés à la recherche de méthodes plus efficaces. Nous étudierons ces travaux dans la suite de ce chapitre, et les séparerons en deux groupes : les implémentations logicielles, et les implémentations matérielles.

1.4.1 Les implémentations logicielles

Dans la partie logicielle, nous allons voir différents types d'implémentations sur processeurs, nous nous intéresserons en particulier aux CPU et GPU qui représentent à eux deux une grande partie des travaux de recherche dans ce domaine.

1.4.1.1 Architectures sur CPU

Un des avantage de travailler sur des architectures logicielles par rapport aux architectures matérielles, est la mise à disposition des codes sources de référence de la plupart des standards vidéo, tels que JM [Ins] ou FFmpeg [FFm]. Il est ainsi plus aisé de les réutiliser pour accélérer le développement d'une plate-forme logicielle, ou tout simplement pour comparer les résultats de sa propre implémentation. En contrepartie, les implémentations purement logicielles sont généralement moins performantes, et requièrent des fréquences de travail très élevées pour réaliser des traitements vidéo temps réel.

Dans [JZy09], il est proposée une méthode pour réduire la complexité du transcodage MPEG-2 vers H.264, tout en augmentant ses performances. Ce travail repose sur l'optimisation de la prédiction inter, et de la partie en charge des vecteurs de mouvements. Alors que MPEG-2 ne travail qu'avec des blocs de 8x8 pixels, H.264, peut aussi bien travailler sur les blocs 8x8 que 4x4. Au lieu de calculer tous les vecteurs de mouvements dans la partie H.264, les auteurs proposent de réutiliser les vecteurs 8x8 déjà obtenus pour MPEG-2 et de les décomposer en blocs plus petits adaptés au standard H.264. Une comparaison des performances est réalisée entre cette nouvelle méthode et le transcodeur de référence JM. Pour une qualité d'image identique, il s'avère que les gains en temps d'exécution sont de l'ordre de 50%.

Une autre étude sur l'amélioration des performances de transcodage est réalisée dans [MFA+07], les auteurs essaient ici de mutualiser les traitements entre un décodeur H.264 et un encodeur MPEG-2 (figure 1.15) en intercalant un module de conversion entre la partie de décodage et d'encodage. Ce bloc repère les informations qui peuvent être utilisées aussi bien par MPEG-2 que par H.264, et au lieu de réaliser deux fois le même traitement, les résultats sont directement transmis du décodeur à l'encodeur. Comme dans le travail précédant, ces informations concernent les vecteurs de mouvements et le traitement inter des données. Le module de conversion permet également de transformer les coefficients issues de la transformée entière de H.264 en coefficients DCT nécessaires à MPEG-2.

TABLE 1.1 – Résultats visuels obtenus pour trois séquences vidéo [MFA+07]

	Mobile Calendar PSNR (dB)	Stockholm PSNR (dB)	Shields PSNR (dB)
H.264/AVC	41.159	38.986	38.849
MPEG-2	36.571	36.240	34.605
TranscRefer	36.425	36.127	34.559
TranscModif	36.291	36.082	34.499

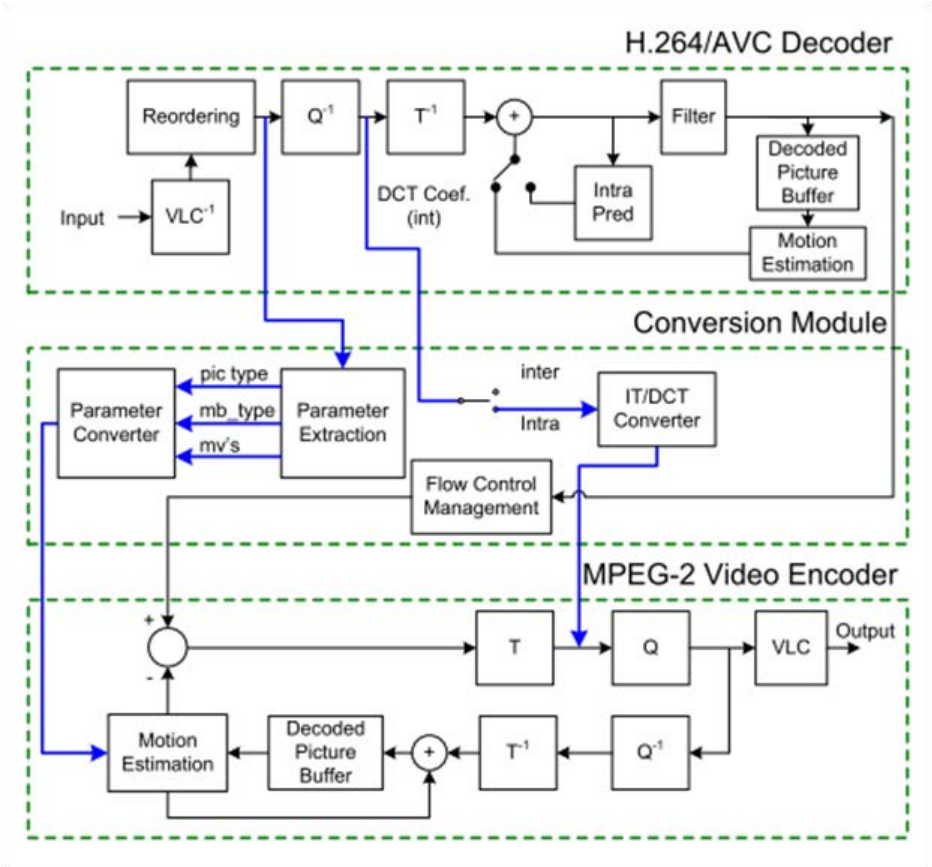


FIGURE 1.15 – Architecture du transcodeur avec son module de conversion [MFA+07]

L'implémentation de ce transcodeur a été réalisée sur un processeur d'ordinateur cadencé à 3Ghz et testée sur des séquences vidéo 720p. Cette solution réduit de 30% les temps d'exécution pour une qualité d'image identique au transcodeur de référence JM, le tableau 1.1 montre les résultats visuels obtenus sur trois séquences vidéos, les deux premières lignes correspondent aux PSNR obtenus avec JM et la dernière ligne aux PSNR avec leur implémentation.

Une dernière méthode [LAH04, MD02, WHZ+01, SLB04, DBKW05] permet d'améliorer les performances de transcodage sur processeur grâce à l'utilisation de proxys. Cette méthode consiste à déporter les opérations de transcodage sur un serveur distant. Les traitements ne sont plus effectués directement chez l'utilisateur, mais sur des processeurs plus puissants disposés entre le fournisseur de contenus et le client, les proxys récupèrent les informations concernant le flux vidéo entrant et le terminal de lecture pour réaliser le transcodage et fournir un flux adapté au client.

Les méthodes d'implémentation sur processeurs restent intéressantes de par leur flexibilité.

Aux vues des performances et des résultats obtenus, ce type de méthode peut être suffisant pour le traitement de vidéo basse résolution (pour l'utilisation mobile par exemple), mais ne permet pas actuellement de tenir les contraintes temps réel nécessaires aux vidéos de plus grandes résolutions. L'augmentation de la densité d'intégration (Loi de Moore), et des fréquences de travail, ont permis aux processeurs de suivre les évolutions du marché. Mais cette tendance atteint ses limites, et depuis quelques années, les travaux d'optimisation des processeurs portent d'avantage sur la multiplication des unités de calcul. (Processeurs 4 cores, 8 cores, etc)

1.4.1.2 Architectures sur GPU

Les GPU (Graphics Processing Units) sont des puces dédiées aux traitements graphiques de nos ordinateurs. L'avantage par rapport aux simples processeurs réside dans la forte parallélisation de leurs unités de traitement, les GPU sont composées d'une multitude (plusieurs centaines, voir milliers) de processeurs indépendants capables d'exécuter simultanément plusieurs calculs (figure 1.16).

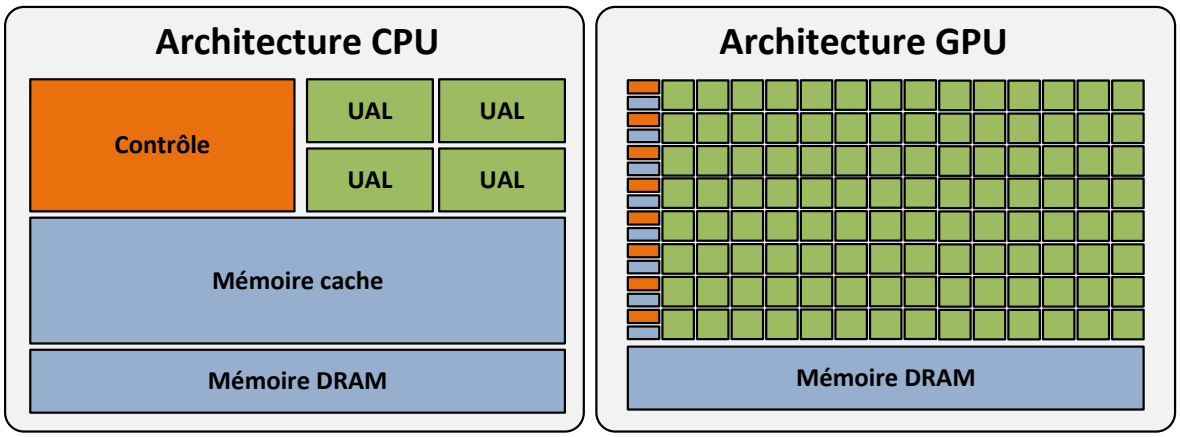


FIGURE 1.16 – Architectures CPU et GPU

L'évolution des GPU a contribué au développement des cartes graphiques au point de rendre leur puissance de calcul très intéressante pour des applications autres que celles dédiées au graphisme. La figure 1.17 montre l'évolution de la puissance de calcul entre CPU et GPU au cours des dernières années.

Dans [CAK+09] et [CFAK10], les auteurs ont étudié la possibilité de déporter certaines fonctions de transcodage sur des architectures GPU. Du fait de la grande parallélisation des modules de traitement, cette solution se prête très bien à l'implémentation de fonctions comprenant beaucoup d'opérations indépendantes, dissociées les unes des autres, comme les algo-

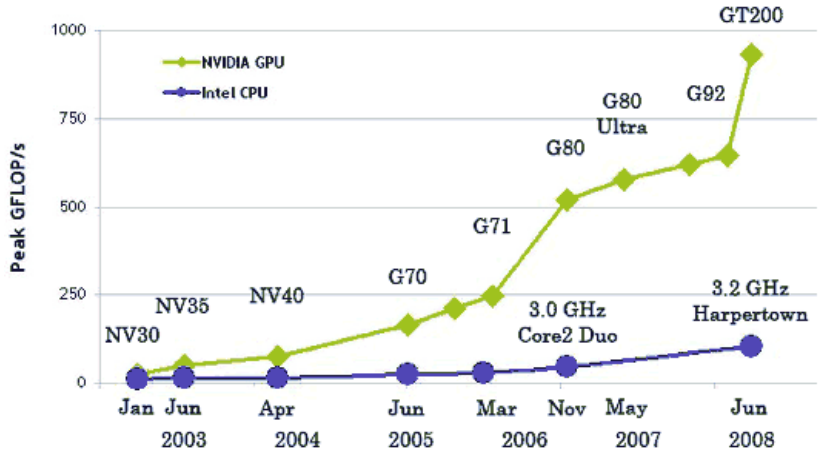


FIGURE 1.17 – Puissance de calcul CPU et GPU [Sim]

rithmes d'estimation du mouvement et de prédiction intra. L'objectif est de fractionner le plus possible ces fonctions pour occuper au maximum le champ de processeurs disponibles, et de réduire au maximum la sérialisation des traitements et les temps de calcul. Les résultats obtenus sur des séquences vidéo en 720p montrent un temps d'exécution de l'ordre de 835ms pour une image, et une décomposition des calculs de la prédiction inter sur 3600 processeurs.

Les deux constructeurs leaders dans le développement de carte graphique, Nvidia et ATI s'intéressent aussi à la question du transcodage et commercialisent déjà leurs applications. Nvidia avec Badaboom [KOW08], une solution logicielle sur GPU pour le transcodage de séquences vidéo haute définition (1080p) à 90 images par seconde. Et ATI avec Avivo Video Converter [WIL09], qui offre des résultats et performances quasi similaires à la solution Nvidia.

Les résultats présentés par ces constructeurs sont plus qu'encourageants, et pourraient apporter un avantage certain pour l'avenir et le transcodage des nouveaux standards très hautes définitions 4K (HEVC).

1.4.2 Les implémentations matérielles

Les solutions matérielles représentent un second axe de recherche en termes d'optimisation des performances de transcodage. Moins flexibles que les solutions logicielles au niveau de la programmation, elles ont néanmoins la capacité de réaliser un grand nombre de calculs en peu de cycles d'horloge. Comme pour la partie précédente, nous avons séparé les implémentations matérielles en deux groupes, les SoC et les FPGA.

1.4.2.1 Architectures sur SOC

Les SoC (System on Chip), sont des circuits intégrés capables embarquer des systèmes complets, avec n'importe quel type d'interface de communication, mais aussi de la mémoire, ou des microprocesseurs dédiés. Plusieurs sociétés dans ce domaine proposent des transcodeurs embarqués à la vente.

C'est notamment le cas du constructeur Thomson [HEN08], qui a développé un SoC pour le transcodage MPEG et H.264. Celui-ci embarque plusieurs DSP dédiés au calcul des coefficients de la transformée entière, des blocs RAM pour temporiser les données entre plusieurs traitements, et deux processeurs ARM pour la régulation des flux.

La société Fujitsu [FUJ09] propose également un SoC de ce type, qui offre des performances similaires au précédant, si ce n'est en terme de consommation. Ce dernier peut réaliser un transcodage temps réel à une fréquence supérieure à 200Mhz sans dépasser 1W de consommation électrique.

1.4.2.2 Architectures sur FPGA

Les FPGA (Field Programmable Gate Array) appartiennent à la famille des circuits logiques programmables, et sont composés de plusieurs types de ressources dont l'utilisation diffère en fonction des besoins. L'architecture interne d'un FPGA change d'un constructeur à un autre, chez Xilinx par exemple, les principales ressources disponibles sont les CLB (Configurable Logic Block), les BRAM (Block Random Access Memory), les DSP (Digital Signal processor) et les IOB (Input-Output Bloc), et sont disposées par groupes et par colonnes (figure 1.18).

– ***Les ressources CLB***

Les blocs logiques configurables servent à configurer différentes fonctions logiques combinatoires et/ou séquentielles. Pour cela, et comme on peut le voir sur la figure 1.19, chaque bloc CLB contient assez de logique pour créer une machine d'états, et des LUT (LookUp Tables) pour la création de fonctions logiques combinatoires. Les données sont synchronisées et acheminées grâce a des bascules D et des multiplexeurs.

L'architecture d'un CLB diffère en fonction du FPGA et de la technologie utilisée. Sur une famille de FPGA Xilinx Virtex 4 par exemple, chaque CLB dispose de deux LUT à quatre entrées (comme représenté sur la figure 1.19), alors que pour les Virtex 5 et 6, chaque CLB dispose de quatre LUT à six entrées, soit deux fois plus de fonctions combinatoires, et la possibilité d'utiliser six variables d'entrée pour chaque fonction au lieu de quatre.

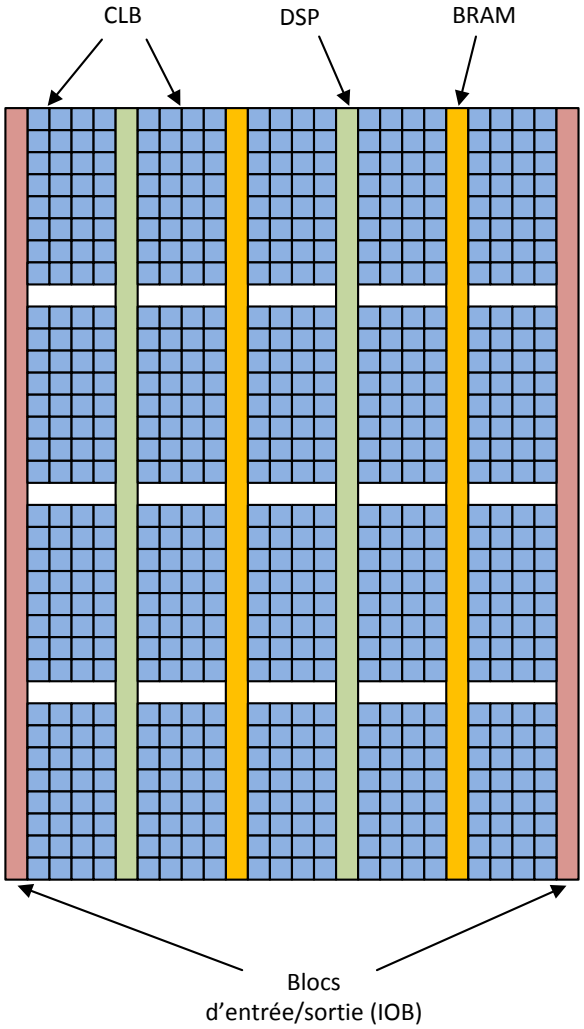


FIGURE 1.18 – Architecture interne d'un FPGA

– ***Les ressources BRAM***

Les blocs BRAM sont des mémoires internes au FPGA, elles permettent de stocker rapidement des données. Leurs temps d'accès étant bien plus court que pour des mémoires externes au FPGA, les BRAM sont particulièrement intéressantes pour le stockage intermédiaires entre plusieurs fonctions. De plus, puisqu'il s'agit de mémoires à doubles ports, il est possible d'utiliser deux horloges différentes pour l'écriture et la lecture dans celle-ci. Comme pour les CLB, les blocs BRAM peuvent différer entre plusieurs familles de FPGA, ainsi on peut trouver des BRAM sur 16 ou 32 Ko.

– ***Les ressources DSP***

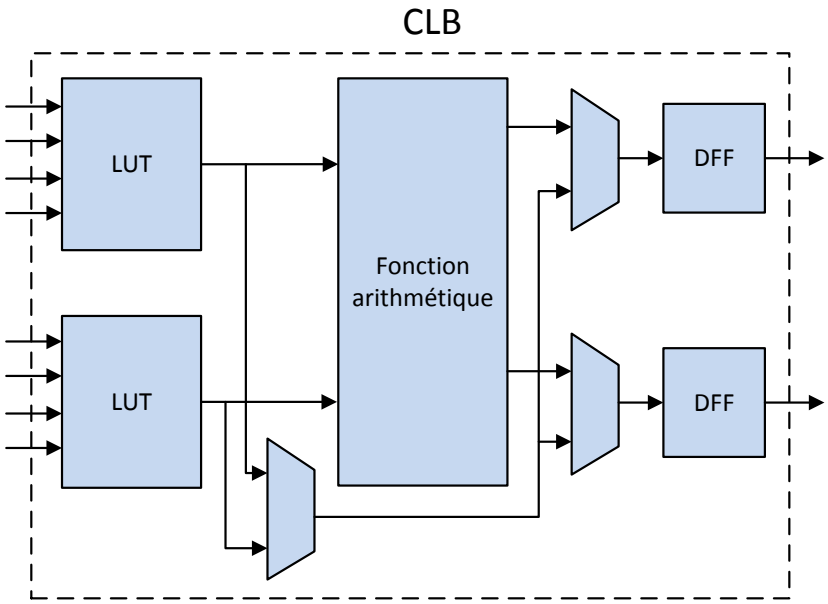


FIGURE 1.19 – Composition d'un CLB

Les blocs DSP réalisent des fonctions à grain plus gros, telles que des traitements de signaux ou d'autres fonctions dédiées comme du multiplexage, des additions ou multiplications, mais également n'importes quelles fonctions combinatoires de base.

– ***Les ressources IOB***

Enfin, les blocs IO sont nécessaires à l'interconnexion des fonctions internes du FPGA avec les éléments externes. Chaque bloc IO est dédié à une broche du composant, et possède toutes ses caractéristiques en mémoire, comme son niveau de tension ou encore son sens de communication. La communication interne quant à elle, est réalisée par des bus macro, placés aux extrémités des modules, ils mettent en œuvre une communication fixe entre les différentes fonctions du FPGA [LBM+06]. Les bus macro sont aussi utilisés pour réaliser la communication entre les modules reconfigurables et la partie statique du design.

Beaucoup de travaux traitent d'implémentation de transcodeur sur FPGA, c'est notamment le cas de [BLN+06] qui propose une architecture capable d'encoder en temps réel une vidéo H.264 de résolution 720p à seulement 50Mhz grâce à la mise en pipeline de leur architecture.

Nova [XLGC10, XC08a, XC08b, XCCP06, XCCP07, XC07] est une autre architecture (figure 1.20) dédiée aux appareils mobile, disponible en open source, elle permet de réaliser un décodage H.264 de résolution QCIF et ce pour une consommation inférieure à 300 μW.

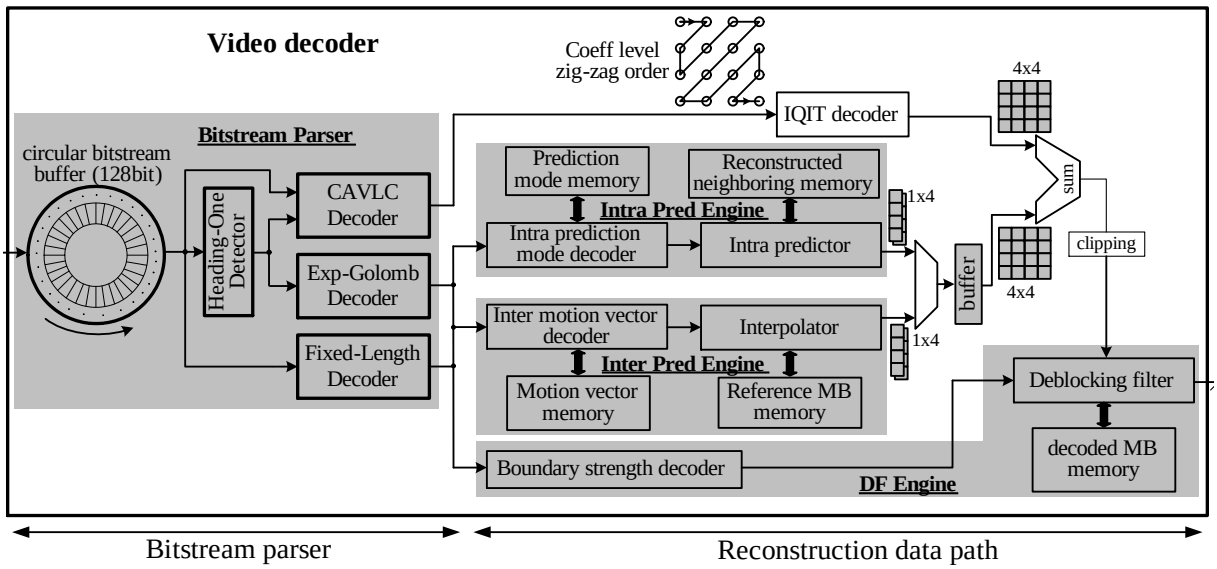


FIGURE 1.20 – Architecture du décodeur NOVA [XLGC10]

1.5 Conclusion

Les solutions logicielles possèdent l'avantage de pouvoir être reprogrammées à n'importe quel moment et offre une flexibilité intéressante. En contrepartie, leurs temps d'exécution ne sont pas toujours adaptés au respect des contraintes temps réel imposées par le traitement vidéo, à moins de réaliser une architecture multiprocesseurs demandant un effort considérable de découpage et de placement des différentes fonctions de traitement.

Les solutions matérielles quant à elles, autorisent grâce à leur architecture interne, une exécution parallèle des tâches, et offrent une puissance de calcul bien plus élevée que les processeurs, pour des fréquences et des consommations électriques moindres. Malheureusement, par leur manque de flexibilité, leur utilisation est, à premières vues, non appropriée au développement d'un transcodeur adaptable. Implémenter l'ensemble des standards vidéo en permanence est peu envisageable au regard des ressources exigées. Malgré tout, cette solution reste envisageable grâce à la méthode de reconfiguration dynamique sur FPGA. Celle-ci permet d'isoler une partie du circuit pour en modifier le fonctionnement sans pour autant arrêter l'exécution du reste du FPGA. Grâce à cette technique, il serait alors possible d'implémenter en temps réel le codec nécessaire au transcodage souhaité.

Avant de faire un choix technologique, il aurait été intéressant d'étudier qualitativement et quantitativement chaque solution par expérimentation. Toutefois, c'est la technique de reconfiguration dynamique que le consortium ARDMAHN a retenue pour l'implémentation du

transcodeur vidéo. Dans la suite du manuscrit, nous nous concentrerons donc, uniquement sur l'étude de celle-ci.

Chapitre 2

Techniques de modélisation de la reconfiguration dynamique

Sommaire

2.1 Introduction 32

2.2 La reconfiguration des FPGA 32

2.2.1 La reconfiguration statique 33

2.2.2 La reconfiguration dynamique 33

2.2.3 La reconfiguration dynamique appliquée au transcodage vidéo 35

2.3 La modélisation 36

2.3.1 Les outils de modélisation 36

2.3.2 Modélisation UML 37

2.3.3 Modélisation SystemC 38

2.3.4 L'ordonnancement des tâches reconfigurables 42

2.4 Conclusion 43

2.1 Introduction

Nous l'avons vu dans l'introduction générale, les FPGA ont la capacité de pouvoir être reconfigurés à volonté pour modifier leur fonctionnement. Jusqu'en 1997, avant la sortie du premier FPGA reconfigurable dynamiquement, cette configuration ne pouvait être réalisée qu'une fois le système à l'arrêt, rendant impossible la modification du circuit logique au cours de l'exécution. La technique de reconfiguration dynamique permet de palier ce manque d'adaptabilité. Son rôle consiste à modifier une partie de l'architecture sans interrompre l'exécution du reste du FPGA, on parle alors de circuits intelligents capables de s'auto adapter aux besoins de leur environnement. Cette pratique reste toutefois complexe à mettre en œuvre, il est difficile, voir impossible de définir à l'avance les besoins en ressources d'une telle architecture, ainsi que sa faisabilité pour une application donnée.

La suite de ce chapitre est organisée en deux parties distinctes. Dans la première, nous rappellerons les différentes méthodes de reconfiguration disponibles. Puis, dans la seconde partie, nous verrons les différents travaux réalisés sur la modélisation et la création d'outils dédiés à la simplification de l'utilisation de la reconfiguration dynamique.

2.2 La reconfiguration des FPGA

Avant de débuter ce paragraphe, il est important de préciser que l'architecture d'un FPGA varie en fonction de son constructeur, or, au commencement de la thèse, seul Xilinx proposait vraiment la reconfiguration dynamique et des outils de développement permettant sa mise en place. C'est pourquoi, dans la suite de ce manuscrit, nous nous intéresserons en particulier aux FPGA de ce constructeur. Depuis, Altera a également franchi le pas, et propose ses propres solutions pour cette technique.

Il existe plusieurs méthodes de reconfiguration, comme on peut le voir sur la figure 2.1, nous pouvons les regrouper en deux catégories : la reconfiguration statique, et la reconfiguration dynamique. Cette dernière peut également se décliner en deux variantes, la reconfiguration dynamique partielle ou totale. Dans les paragraphes suivants, nous expliquerons le fonctionnement de ces différentes méthodes et comparerons leurs caractéristiques.

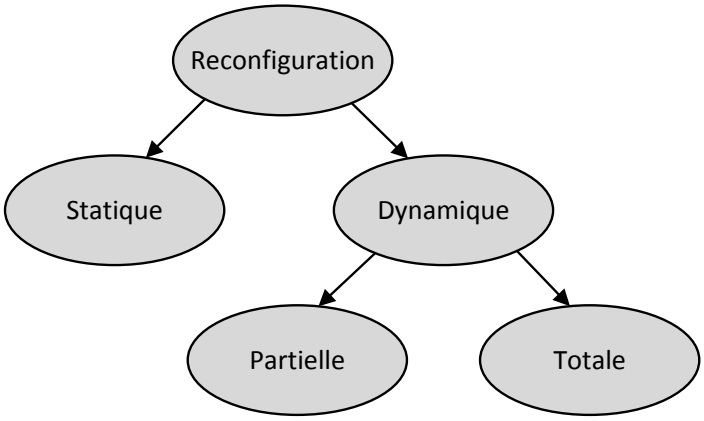


FIGURE 2.1 – Méthodes de reconfiguration sur FPGA

2.2.1 La reconfiguration statique

Comme toutes les méthodes de reconfiguration, la méthode statique permet de modifier la configuration matérielle du FPGA. Elle est dite statique car son action n'intervient pas durant la phase d'exécution du circuit, le FPGA doit être stoppé par l'utilisateur avant qu'une nouvelle configuration puisse être téléchargée. Son fonctionnement est donc figé pendant toute la durée de vie de l'application. Son utilisation n'est pas conseillée pour des systèmes ayant besoin d'adaptabilité et de modifications au cours du temps.

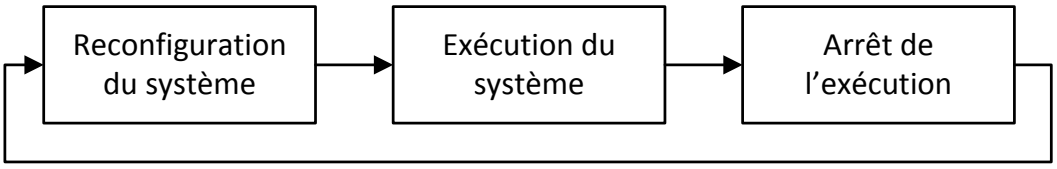


FIGURE 2.2 – Principe de la reconfiguration statique

La figure 2.2 montre le cycle de fonctionnement d'une architecture statique. L'arrêt et la remise en route du système après la reconfiguration sont réalisés par l'utilisateur.

2.2.2 La reconfiguration dynamique

La reconfiguration dynamique octroie au système la possibilité d'être modifié durant la phase d'exécution (figure 2.3). Dans ce cas, la reconfiguration peut être contrôlée soit par l'utilisateur comme dans la méthode précédente, soit par le FPGA via un microprocesseur interne

(Microblaze par exemple). Dans ce dernier cas, on parle de FPGA auto adaptable puisqu'il gère lui-même ses ordres de reconfiguration en fonction des besoins de l'application. En plus de gagner en flexibilité, le FPGA voit l'utilisation de sa surface considérablement optimisée. Plusieurs tâches peuvent être placées à tour de rôle sur une même zone, les ressources sont alors mutualisées et réutilisées d'une configuration à une autre. Cette alternance nécessite tout de même un paramétrage et un ordonnancement bien précis afin de respecter les contraintes de temps imposées par chaque application [MRDW11a, DPW99, HSB06].

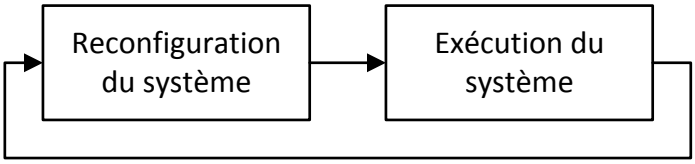


FIGURE 2.3 – Principe de la reconfiguration dynamique

La reconfiguration dynamique peut se décliner en deux variantes : la reconfiguration dynamique totale, et partielle. Dans cette première, l'intégralité de la surface du FPGA est réimplantée à chaque reconfiguration. Entre deux changements consécutifs, il peut arriver que certaines parties de l'architecture soient identiques, mais soient tout de même écrasées par une configuration identique à la précédente. En plus d'être inutile, le temps de reconfiguration est impacté, puisqu'il dépend directement de la taille de la configuration téléchargée.

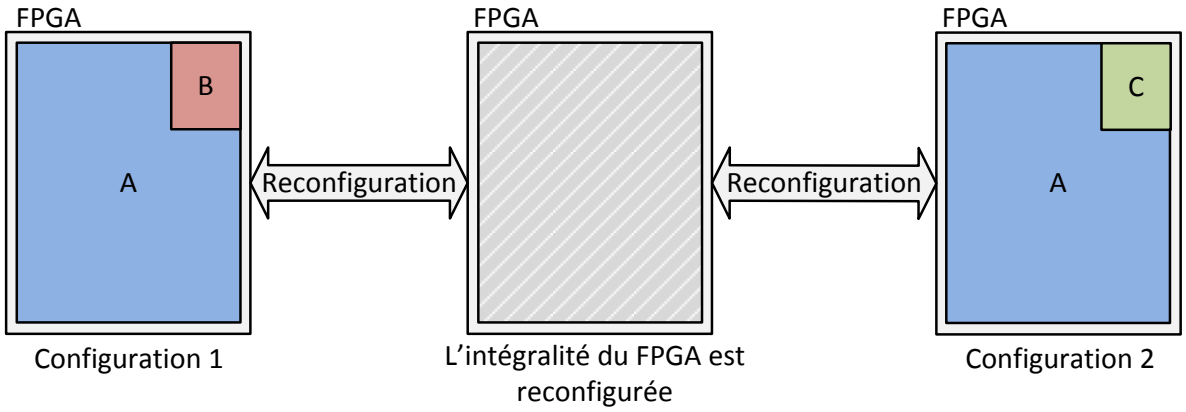


FIGURE 2.4 – Principe de la reconfiguration totale du FPGA, toute la surface du FPGA est modifiée

Prenons l'exemple de la figure 2.4, dans lequel nous avons deux configurations successives.

La tâche A occupe 90% du FPGA et est présente dans les deux cas. Les tâches B et C occupent seulement 10% du FPGA, et partagent leurs ressources d'une configuration à l'autre. Cet exemple illustre bien le problème de la reconfiguration dynamique totale, alors que seulement 10% de la surface du FPGA a besoin d'être modifiée, c'est la totalité du FPGA qui l'est à chaque changement. En plus de l'augmentation inutile du temps de reconfiguration, la tâche A devra suspendre son exécution à chaque fois.

Pour prévenir ce problème, la reconfiguration dynamique partielle peut être utilisée. Cette méthode permet de définir une ou plusieurs zones indépendantes du reste du design, qui pourront être modifiées à n'importe quel moment, sans interrompre l'exécution du reste du FPGA.

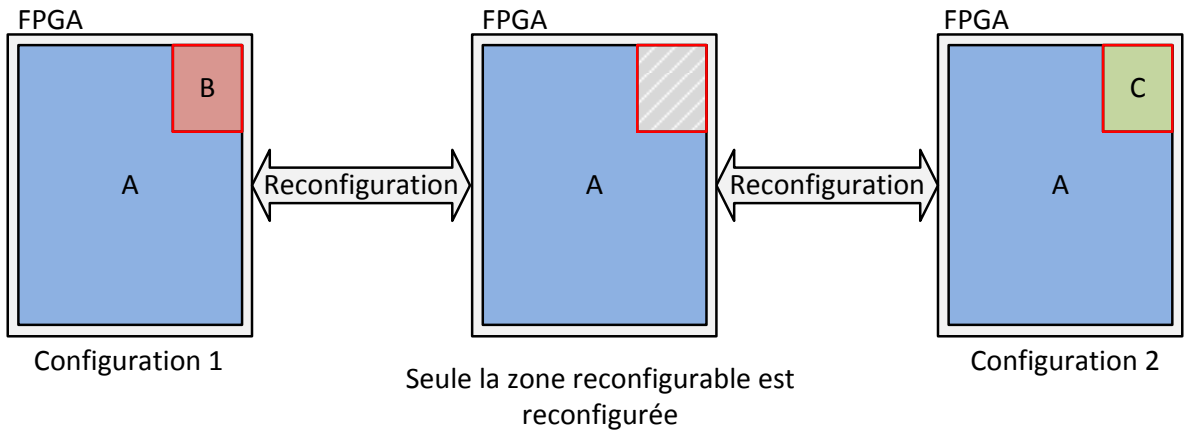


FIGURE 2.5 – Principe de la reconfiguration partielle du FPGA, seule la zone reconfigurable est modifiée

Reprenons l'exemple précédant, dans la figure 2.5. Cette fois, l'utilisateur définit une zone reconfigurable suffisamment grande pour contenir les applications B et C en alternance. Seule cette zone sera modifiée. Le temps de reconfiguration est nettement réduit, et l'application A peut continuer à s'exécuter sans interruption.

2.2.3 La reconfiguration dynamique appliquée au transcodage vidéo

La reconfiguration dynamique de FPGA semble être une solution appropriée pour l'implémentation d'un transcodeur vidéo performant. Les FPGA offrent des performances intéressantes en termes de temps d'exécution et de puissance de calcul, et grâce à l'utilisation de la reconfiguration dynamique, leur flexibilité est proche de celle des solutions logicielles. Pour autant, dans la littérature, les architectures basées sur la reconfiguration dynamique ne portent que rarement

sur l'implémentation d'un transcodeur complet, mais plutôt sur certaines fonctions de celui-ci, comme par exemple [SPC05, DCB+09] pour la prédiction intra, [CLSG06, RB07, GM07, MRDW11a, MRDW11b] pour le codeur entropique CAVLC, ou encore [CWYL06, CWLY09] pour la transformée et la quantification.

Comme la plupart des applications dans le domaine multimédia, le transcodage vidéo fait appel à des contraintes temporelles très fortes, le moindre décalage dans le flux de données temps réel peut impliquer une forte dégradation visuelle pour l'utilisateur. En plus de ces contraintes, l'utilisation de la reconfiguration dynamique engendre de nouvelles latences et retards qu'il faut impérativement prendre en compte.

Avant de développer une telle application, le concepteur doit se poser certaines questions :

- combien de zones reconfigurables dois-je utiliser et comment les découper ?
- Combien de vidéo une chaine de transcodage reconfigurable est elle capable de traiter sans dégrader la vidéo ?
- A quel moment une reconfiguration doit elle avoir lieu ?
- Comment dimensionner les mémoires permettant de stocker les flux vidéo pendant d'une reconfiguration ?

Autant de questions qui nécessitent un lourd travail de recherche au préalable. L'objectif des travaux présentés ici est de répondre à ces questions, et d'aider au dimensionnement de chaque élément de l'architecture. Un modèle d'aide à la conception a été développé durant mon travail de thèse, celui-ci prend en compte différents éléments et variables imposés à la fois par l'architecture du FPGA, mais également par l'application à développer. Dans la suite de ce chapitre, nous verrons les différents travaux déjà réalisés sur la modélisation de la reconfiguration dynamique.

2.3 La modélisation

2.3.1 Les outils de modélisation

La modélisation de la reconfiguration dynamique est un thème récurrent dans la littérature, une grande partie de ces travaux utilisent les langages de modélisation UML et SystemC.

2.3.2 Modélisation UML

UML (Unified Modeling Language) [Grob] est un langage de modélisation graphique initialement utilisé en informatique dans le cadre de la conception orientée objet. Il est né de la fusion de trois autres langages de modélisation (Booch, OMT et OOSE), et a été standardisé par l'OMG (Object Management group) en 1997.

Au fils des années, UML a pu évoluer grâce au développement et à l'intégration de plusieurs extensions, offrant la possibilité aux développeurs de modéliser des systèmes matériels et mixtes. Parmi ces extensions, SysML (System Modeling Language) [Par] par exemple, est un profil UML qui s'adresse particulièrement à la modélisation de systèmes matériels complexes en incluant la possibilité d'introduire des contraintes liées aux lois physiques. Ou encore MARTE (Modeling and Analysis of Real-Time Embedded Systems) [Groa] qui permet d'étendre les possibilités de la modélisation UML en intégrant, entre autres, la notion de temps reel. Plusieurs travaux modélisant la reconfiguration dynamique ont ensuite été menés sur la base de ce qui a été fait avec UML et l'outil MARTE.

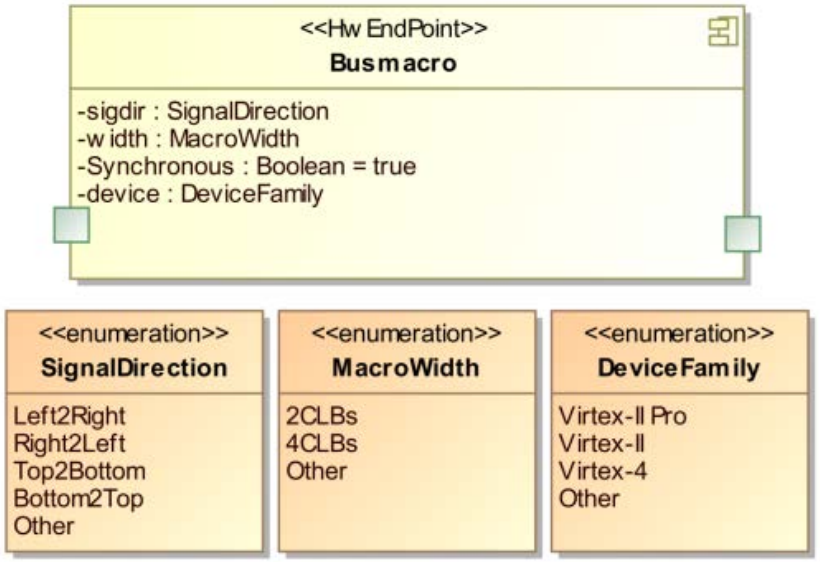


FIGURE 2.6 – Exemple d'instanciation d'un bus macro avec l'outil GASPARD2 [QMD08]

GASPARD2 est un outil basé sur MARTE. Il s'agit en fait d'un environnement de développement visuel pour la modélisation et la conception de SoC. En plus de permettre la modélisation haut niveau du système comme c'était le cas avec MARTE, il permet également la transformation d'un modèle abstrait vers un code dépendant d'un langage spécifique, on parle

alors d'ingénierie dirigée par les modèles (ou MDE pour Model Driven Engineering). La génération du code logiciel et matériel du système peut être faite pour différentes cibles, telles que SequentialC, PThread, OpenCL, OpenMP, Lustre, SystemC, ou encore VHDL afin de pouvoir mener le design flow jusqu'au bout.

Dans [QMD08, QMM+09], les auteurs décrivent les modifications de MARTE apportées par GASPARD2 pour prendre en considération le concept de zone reconfigurable. La figure 2.6 montre les différents paramètres à renseigner lors de l'instanciation d'un bus macro, on y retrouve par exemple, la taille en CLB, la cible FPGA ou encore sa direction.

2.3.3 Modélisation SystemC

Tout comme le langage VHDL ou Verilog, l'outil SystemC [Ini] est considéré comme un langage de description matériel. Il est en fait bien plus que ça, puisqu'il permet de modéliser aussi bien les systèmes logiciels, matériels ou mixtes, et ce, à plusieurs niveaux d'abstraction possibles tels que RTL (Register Transfer Level), ou encore TLM (Transaction Level Modeling), c'est pourquoi il est particulièrement approprié pour la conception de SoC (figure 2.7).

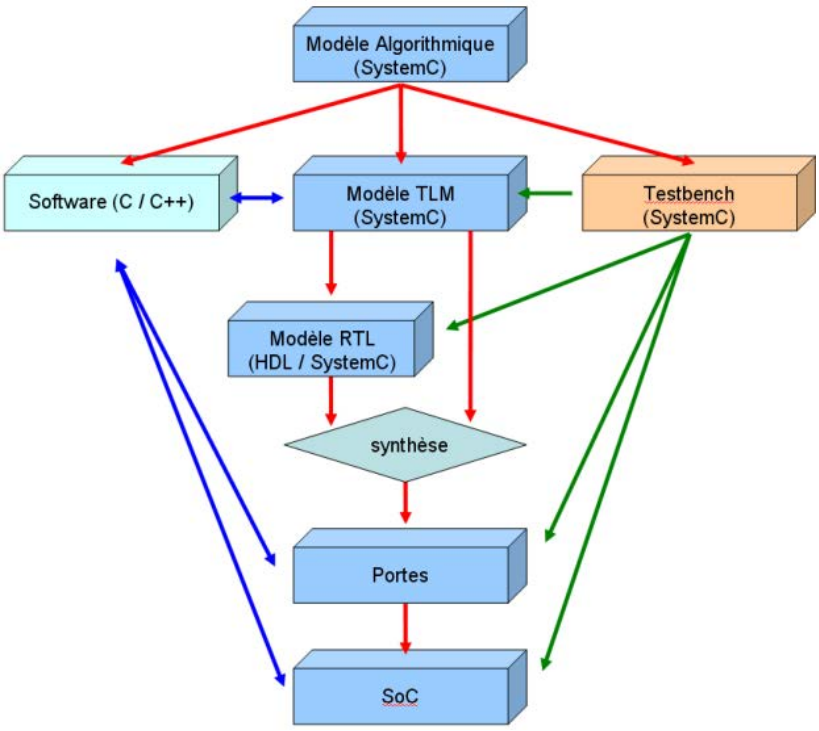


FIGURE 2.7 – Flot de conception SystemC [Pol]

SystemC est une bibliothèque open source incluant un noyau de simulation en langage C++, la simplicité de la syntaxe de ce dernier en fait un outil largement plébiscité par la communauté. Initialement développé en commun par plusieurs entreprises, il a été standardisé en 2005 par l'OSCI sous l'appellation IEEE 1666.

Depuis sa version 2.1, SystemC intègre les threads dynamiques utiles à la modélisation de la reconfiguration dynamique. Contrairement aux threads statiques utilisés jusque là, ceux-ci ont la particularité de pouvoir être créés et détruits durant la phase de simulation. Ce nouveau comportement, bien que trivial permet de faire une première analogie avec le comportement réel de la reconfiguration dynamique sur FPGA. Néanmoins, alors que SystemC autorise le remplacement d'un module par un autre, le changement de connexion de façon dynamique n'est quant à lui pas encore permis. Il est alors impossible de modéliser convenablement la reconfiguration dynamique sans avoir recours à un certain nombre de modifications du code source de SystemC. Plusieurs travaux proposant de corriger ce problème ont déjà été menés.

Dans [BKH+07, BSM10], les auteurs présentent une technique de modélisation et de simulation de la reconfiguration dynamique à l'aide d'instructions haut niveau directement introduites dans le code source de la bibliothèque SystemC. L'idée est de décrire les différents modules en leur définissant une surface d'occupation et un temps de reconfiguration, ces modules sont ensuite mis en exécution ou en veille grâce à des interruptions et évènements. En plus d'avoir l'inconvénient de modifier le code source de SystemC, cette approche ne reflète pas parfaitement le fonctionnement réel de la reconfiguration dynamique, puisque les modules décrits sont en permanence présents, puis mis en veille alternativement.

Les auteurs de [AKK08] quant à eux, proposent une approche utilisant les threads dynamiques. Lors de la création du module dynamique, celui-ci est décomposé en deux processus, un processus utilisateur qui contient la description statique du module et son fonctionnement interne, et un processus de contrôle permettant de gérer l'aspect reconfigurable du module (figure 2.8). Lorsqu'une requête de destruction est envoyée au module, le processus de contrôle transmet une interruption au processus utilisateur, puis attend l'acquittement de celui-ci pour supprimer le processus utilisateur. Une fois cette dernière étape effectuée, seul le processus de contrôle reste présent dans le module, et pourra être réutilisé pour créer un nouveau processus utilisateur lorsqu'une requête de ce genre lui sera envoyée (figure 2.9).

En plus de ce travail sur les threads dynamiques, les auteurs introduisent une classe supplémentaire dans le code source de SystemC, nommée dc_port, elle permet d'interfacer des

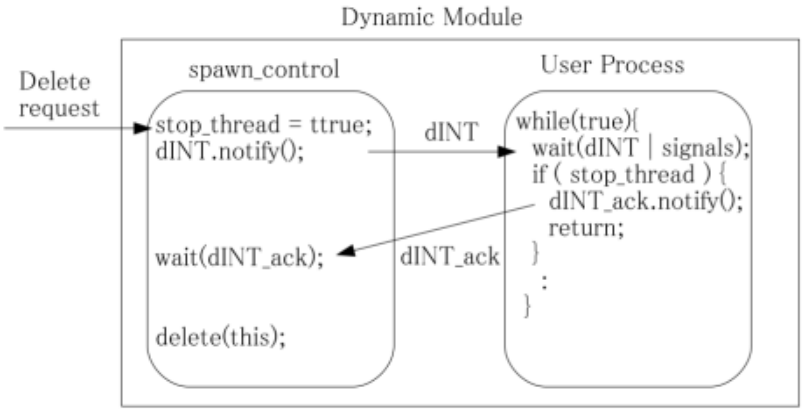


FIGURE 2.8 – Composition d'un module reconfigurable [AKK08]

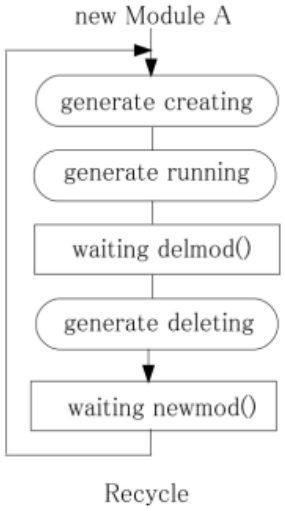


FIGURE 2.9 – Cycle de vie d'un module reconfigurable [AKK08]

modules entre eux durant la phase de simulation, comblant ainsi les manques du langage C++.

Une autre méthode de modélisation [RHA08, RF08] consiste cette fois ci à créer une bibliothèque venant annexer SystemC. Baptisée ReChannel, elle introduit la notion de portal, qui permet de connecter la partie statique du design, à la partie reconfigurable. Chaque portal peut être connecté à un ou plusieurs modules, mais n'autorise qu'aux modules actifs de communiquer, sachant que pour chaque portal, seul un module peut être activé à la fois. Les modules connectés aux portals utilisent l'ordonnanceur SystemC, ils sont donc activés grâce à des évènements répertoriés dans leur liste de sensibilités. Les évènements provenant de la partie statique du design passent d'abord par le portal qui s'occupe de les rediriger vers les modules actifs

(figure 2.10).

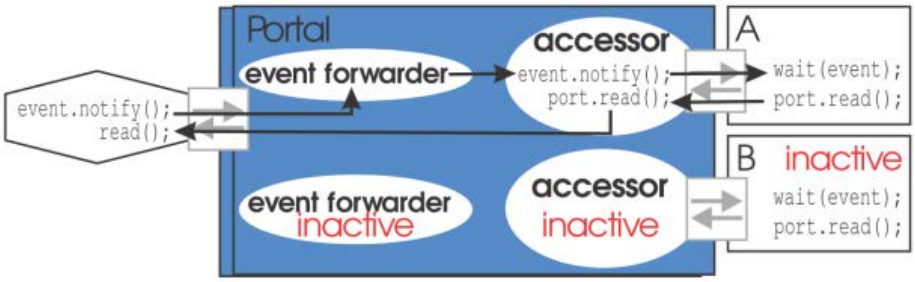


FIGURE 2.10 – Portal utilisé dans ReChannel [RHA08]

Pour permettre de contrôler la reconfiguration, la bibliothèque ReChannel met à disposition un arbitre, qui, en communiquant avec les portals, réalise l'instantiation, le chargement, l'activation, ou la désactivation des modules, tout en respectant les contraintes de temps de la reconfiguration et la taille des modules (figure 2.11).

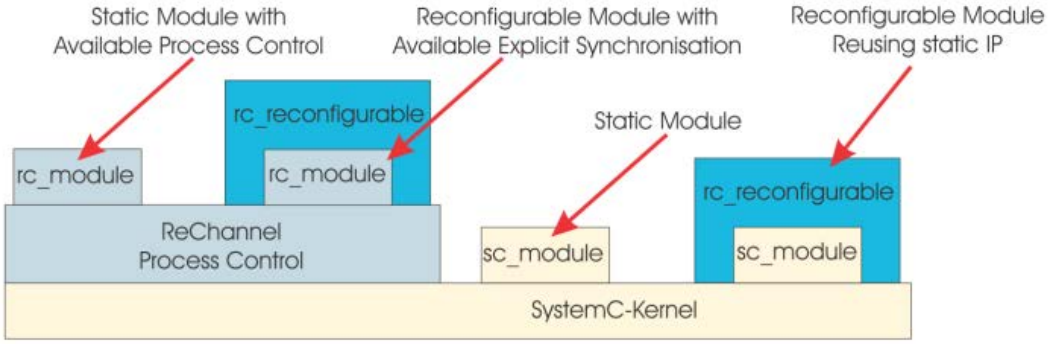


FIGURE 2.11 – Processus d'utilisation de ReChannel [RHA08]

OSSS+R [SNH+09] est une méthodologie importante dans la conception haut niveau de systèmes reconfigurables. Elle est basée sur OSSS [GO02], une extension du langage SystemC qui ajoute les concepts de modélisation, de simulation, ainsi que la synthèse orientée objet. L'hypothèse de base de OSSS+R est que la reconfiguration peut être interprétée comme un échange entre plusieurs objets partageant un type commun dans un conteneur de ce même type aussi. En raison du changement fondamental vers une programmation objet, la réutilisation des IP déjà existantes en SystemC est impossible.

Une autre solution permettant de se défaire des problèmes de communications entre modules dynamiques, est de réaliser une modélisation à un niveau transactionnel TLM (Transaction Le-

vel Modeling). Alors qu'au niveau RTL, la communication se fait à l'échelle du bit logique et au cycle d'horloge près (On parle alors de transaction de type CABA, pour Cycle Accurate/Bit Accurate), en TLM, les communications sont totalement indépendantes de la fonctionnalité du module. Les transactions ne s'effectuent plus que par des appels à des fonctions et à des canaux hiérarchiques, ce sont alors ces canaux qui s'occupent des échanges informationnels de bas niveaux, améliorant ainsi les temps de simulation. En outre, la modélisation TLM apporte également un gain de temps lors de la conception de système de par son niveau d'abstraction élevé. L'utilisation de canaux hiérarchiques pouvant s'apparenter à des modules à parts entières, ceux-ci peuvent donc être manipulés et reconfigurés comme tels. Néanmoins, et malgré ses nombreux avantages, le niveau d'abstraction octroyé par TLM nous amène à penser que son utilisation dans notre étude ne permettrait pas un degré de similitude assez proche du comportement réel de la reconfiguration dynamique.

RecoSim [DML+11b] est un simulateur d'architectures reconfigurables, basé sur SystemC et la modélisation transactionnelle TLM. Il permet de vérifier des ordonnancements de tâches dans le respect des contraintes temporelles de l'application. L'alimentation de ce modèle est réalisée grâce aux informations contenues dans les rapports de synthèse générés par les outils Xilinx. Dans la suite de ces travaux, les mêmes auteurs ont développés un second outil, baptisé FoRTReSS [DMA+13]. Celui-ci est capable de déterminer un ensemble de zones reconfigurables compatibles avec l'application, toujours grâce aux rapports de synthèse. Une première simulation est réalisée avec un nombre de zones reconfigurables prédéfinies. Ce nombre est ensuite incrémenté jusqu'à obtenir une solution viable.

2.3.4 L'ordonnancement des tâches reconfigurables

L'ordonnancement des tâches reconfigurables est une problématique à part entière, mal adapté, il aura un impact certain sur les performances de l'application. Une trop grande fréquence de reconfiguration augmentera significativement la latence du circuit. A contrario, baisser la fréquence de reconfiguration d'une tâche risque de créer des coupures dans le flot de données en aval et créer des encombrements de mémoires en amont. Il est donc primordial de déterminer un ordonnancement en adéquation avec les besoins de l'application.

Les auteurs de [RRMC07] se sont penchés sur les sur-coûts de l'utilisation de la reconfiguration dynamique, et proposent un résumé de quelques solutions explorées dans la littérature. La compression du fichier de configuration (ou bitstream de configuration) est une approche

intéressante, en diminuant le temps de transfert de ce fichier sur la zone reconfigurable, c'est le temps de reconfiguration global qui est directement influencé. Beaucoup de travaux sont réalisés dans ce domaine, et permettent d'atteindre des taux de compression de l'ordre de 80

Une autre méthode consiste à se concentrer sur les tailles des zones reconfigurables implémentées. Utiliser une zone beaucoup trop grande par rapport aux tâches n'a aucun intérêt, et accroît les temps de reconfiguration. Les auteurs de [WP03] proposent un modèle capable d'explorer plusieurs solutions d'ordonnancement, à la recherche de la solution de placement la moins coûteuse possible. Ces ordonnancements s'appuient sur des méthodes préemptives (chaque tâche est configurée pour être exécutée durant un temps déterminé à l'avance) et non-préemptives (Une tâche configurée ne peut être arrêtée que par une interruption).

Enfin, une dernière approche consiste à s'intéresser aux priorités accordées aux tâches reconfigurables. Dans [CRGM11], les auteurs caractérisent chaque tâche en fonction de trois paramètres : le poids, la criticité et la mobilité. Une première priorité leur est d'abord accordée pendant la phase *design-time* en fonction de leur position dans le graphe de tâche, inutile de configurer une tâche en fin de graphe si les précédentes ne le sont pas encore. Une fois la simulation en phase *run-time*, l'ordonnanceur en ligne réattribue les priorités grâce aux trois paramètres précédemment cités. Le poids d'une tache est déterminé en fonction de la latence générée par sa reconfiguration (criticité et mobilité).

2.4 Conclusion

Les travaux sur la modélisation de la reconfiguration dynamique, restent en général à un niveau d'abstraction très élevé. Une grande partie d'entre eux se concentrent sur l'optimisation du dimensionnement des zones reconfigurables, et sur la recherche d'ordonnancement et de placement des tâches. Un ordonnancement ne doit pas seulement réduire les sur-coûts de la reconfiguration, mais doit également permettre à l'application de s'exécuter dans le respect de ses contraintes.

Pour cela, il est nécessaire d'étudier en détails l'application à simuler. Ce qui nécessite de passer par une estimation approfondie des timing à respecter, et de caractériser les flux de données traités et générés par l'application. C'est vers cet objectif que s'orientent nos travaux.

Dans le chapitre suivant, nous reviendrons plus en détails sur le langage SystemC, et la façon avec laquelle nous l'avons utilisé pour notre modélisation. Une première étude de l'application

à simuler sera également donnée.

Chapitre 3

Méthode proposée

Sommaire

3.1 Introduction 46

3.2 SystemC 47

3.2.1 Niveaux d'abstraction des processus 47

3.2.2 La reconfiguration dynamique : threads statiques et dynamiques . . . 48

3.3 Construction du modèle 50

3.3.1 Le module reconfigurable 51

3.3.2 Caractérisation des éléments du modèle 53

3.3.3 La transaction 54

3.3.4 Communication entre modules 54

3.3.5 Ordonnanceur 56

3.3.6 Simulation et stratégie d'exploration 59

3.4 Modélisation de l'application : Le projet ARDMAHN 59

3.4.1 Présentation 60

3.4.2 Chaine de transcodage 61

3.4.3 Définition des cas d'utilisation 63

3.5 Conclusion 67

3.1 Introduction

Ces dernières années, nous avons été témoins d'une forte croissance de la mise sur le marché d'appareils hightech : smartphones, télévisions HD, tablettes. Le marché ne cesse de se renouveler pour satisfaire la demande du consommateur. Cette accélération des demandes a un fort impact sur les industriels et leurs stratégies commerciales, la réduction des temps de développement est primordiale pour rester concurrentiel.

En parallèle, la densité d'intégration offerte par les techniques récentes de gravure sur puce ne cesse d'augmenter, et permet d'un point de vue technique, le développement d'applications toujours plus complexes et gourmandes en ressources. Malheureusement, les outils de conception mis à disposition des développeurs ne suivent pas toujours cette tendance. Afin de réduire les coûts et les délais de développement, les entreprises font de plus en plus souvent appel à des bibliothèques d'IP, ou réutilisent des IP déjà développées pour une précédente application. Cette procédure, rentable dans la plupart des cas, peut s'avérer risquée. Il peut arriver que l'IP réutilisée ne soit finalement pas assez performante pour répondre aux besoins de l'application en cours de développement, et fasse perdre un temps précieux.

La vérification et la simulation des IP (et de l'architecture) en amont de la phase de développement est donc un besoin très fort. C'est dans cette direction, et vers ce besoin que nous avons orienté notre approche. Celle-ci permet de valider une architecture statique ou reconfigurable, et de déterminer les performances requises par chaque IP pour le respect des contraintes de l'application. Les méthodes habituelles partent en général d'un modèle afin de faciliter le développement d'une IP, notre approche est toute différente, et propose de modéliser une architecture à partir d'IP déjà existantes.

Dans la suite de ce chapitre, nous allons revenir sur les éléments qui nous ont conduit à travailler avec le langage SystemC. Puis nous présenterons la méthode utilisée afin de modéliser les différents éléments de l'application. Pour finir, nous reviendrons plus en détail sur le projet ARDMAHN qui constitue l'application de test pour notre approche.

3.2 SystemC

3.2.1 Niveaux d'abstraction des processus

La modélisation avec SystemC peut se faire sur plusieurs degrés de précisions et de détails, on peut par exemple, modéliser une architecture au cycle d'horloge ou au bit près. Ou bien réaliser une simple vérification fonctionnelle, avant d'aller plus loin et de raffiner son modèle. Pour SystemC, ces niveaux d'abstraction sont répertoriés en trois groupes (figure 3.1) :

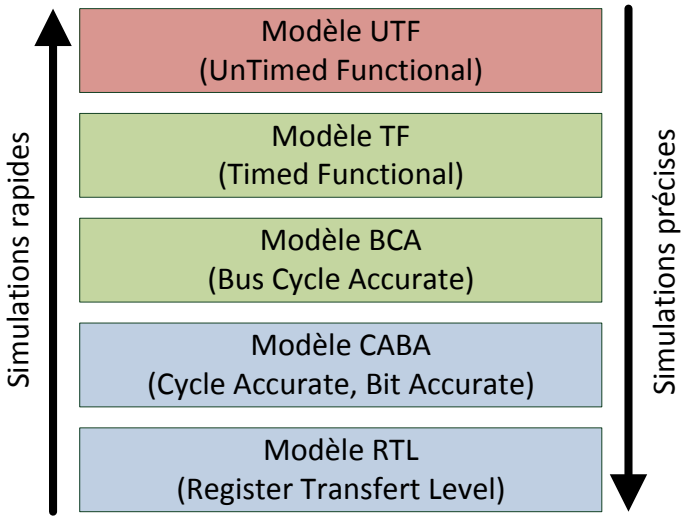


FIGURE 3.1 – Niveaux d'abstraction utilisés par SystemC

– Le modèle UTF (UnTimed Functional) est le niveau de modélisation le plus élevé, il sert à réaliser de simples vérifications fonctionnelles. Ce modèle ne comprend aucune notion de temps et ne permet pas de modéliser les durées d'exécution ou de reconfiguration.
– Le modèle TF (Timed Functional) et BCA (Bus Cycle Accurate) sont deux niveaux d'abstraction inférieurs au modèle UTF, ils intègrent la notion de temps dans les processus, ce qui autorise une analyse plus fine des performances d'une architecture. A ce niveau, les transactions sont correctes au cycle près. Les temps d'exécution, les latences et les temps de propagation de chaque processus peuvent être modélisés, ce modèle est particulièrement bien adaptés à notre étude.
– Le modèle CABA (Cycled Accurate, Bit Accurate) et RTL (Register Transfert Level) sont les niveaux d'abstraction les plus bas, la modélisation se fait au cycle et au bit près, et nécessite une connaissance de chaque registre du système. Ce niveau est synthétisable

et compatible avec les codes VHDL/VERILOG.

Bien sûr, plus le niveau d'abstraction sera élevé et plus les temps de simulations seront courts, à l'inverse, un niveau d'abstraction faible permet d'obtenir des résultats de simulation au plus près des performances réelles. Étant donné que notre étude porte en partie sur le dimensionnement des éléments nécessaires à la mise en place de la reconfiguration dynamique, une grande précision dans les résultats de simulation est de mise, les modèles TF et BCA présentent un bon compromis et offrent une précision de résultats satisfaisante pour notre étude.

3.2.2 La reconfiguration dynamique : threads statiques et dynamiques

Parmi les solutions présentées dans le chapitre précédant, nous avons décidé d'utiliser la méthode des threads dynamiques pour modéliser la reconfiguration dynamique, celle-ci a l'avantage d'être très proche du comportement réel de la reconfiguration, et puisqu'il s'agit d'une fonctionnalité intégrée par les développeurs de SystemC, aucune modification du noyau de celui-ci n'est nécessaire.

Les threads peuvent être apparentés à des fonctions en langage C, mis à part qu'ils sont exécutés de façon concurrente. Alors qu'en langage C, c'est l'utilisateur qui doit faire appel à la fonction, en SystemC, chaque thread est créé avec une liste de sensibilités pour gérer son activation. Trois types de thread existent : les SC_METHOD, les SC_THREAD et les SC_CTHREAD [Bla10]. Ces types fonctionnent typiquement de façon statique, ils sont créés par l'ordonnanceur de simulation pendant la phase d'élaboration, via un appel au constructeur du module.

En complément, depuis la version 2.0 de SystemC, est apparue une nouvelle fonction, les SC_SPAWN. Grâce à elle, il est maintenant possible de créer un processus dynamique au cours de la phase d'élaboration, mais également durant la phase de simulation. L'utilisateur doit au préalable déclarer les différentes tâches reconfigurables sur le thread dynamique, puis c'est un processus complémentaire qui va activer la tâche désirée grâce à la fonction SC_SPAWN. Cette méthode permet ainsi de modifier le comportement d'un thread au cours de son exécution, c'est cette particularité que nous allons utiliser pour modéliser le processus de reconfiguration dynamique.

La figure 3.2 met en évidence l'analogie (simplifiée) entre : l'appel d'une tâche par la fonc-

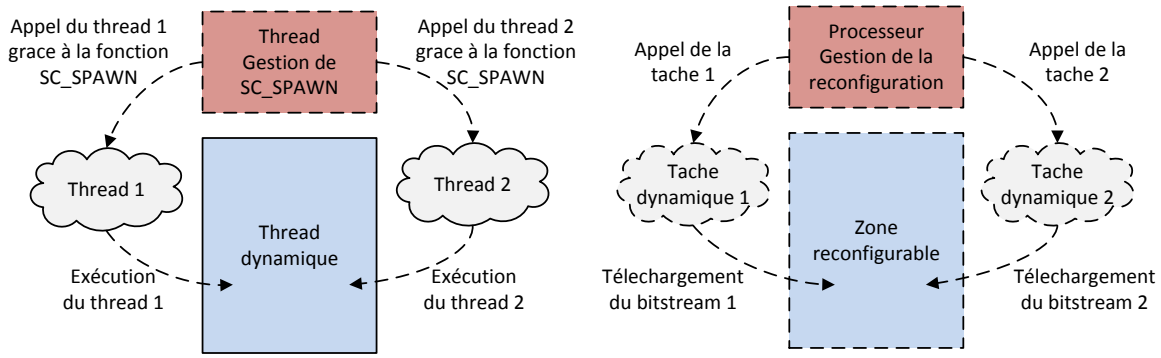


FIGURE 3.2 – Analogie entre la reconfiguration dynamique et l'usage de la fonction SC_SPAWN

tion SC_SPAWN de notre modèle, et la reconfiguration dynamique sur FPGA. Les tâches 1 et 2 représentent les fonctions reconfigurables, modélisées par les threads 1 et 2, le processeur permettant de contrôler la reconfiguration dynamique est représenté par un thread également dédié aux placements des tâches. Enfin, l'appel au téléchargement des bitstream de configuration sur la zone reconfigurable est modélisée par l'utilisation de la fonction SC_SPAWN.

– **Connexions dynamiques**

Une fonction configurée sur un thread dynamique doit avoir accès aux ports d'entrées/sorties de celui-ci pour communiquer avec l'extérieur. Dans le cadre d'un comportement dynamique, ces ports peuvent être communs entre plusieurs fonctions reconfigurables. Or, nous avons vu précédemment que le langage SystemC n'autorisait pas la modification des connexions au cours de la simulation, et que le *portmap* de plusieurs signaux sur un même port est interdit en langage C.

Pour répondre à ce problème, chaque thread dynamique se voit attribuer un wrapper d'entrée, et de sortie. Les fonctions configurées sur le thread dynamique sont alors connectées statiquement et définitivement aux wrappers. C'est ensuite ces derniers qui, grâce à un contrôleur et un multiplexage interne, autorisent ou non la communication vers l'extérieur. La figure 3.3 illustre le cas d'un thread dynamique pouvant accueillir deux fonctions alternativement :

– Dans le cas où aucun thread n'est configuré, les wrappers coupent toute connexion vers l'extérieur.
– Dès qu'une des deux fonction est configurée, les wrappers autorisent et acheminent les

données vers les bons ports.

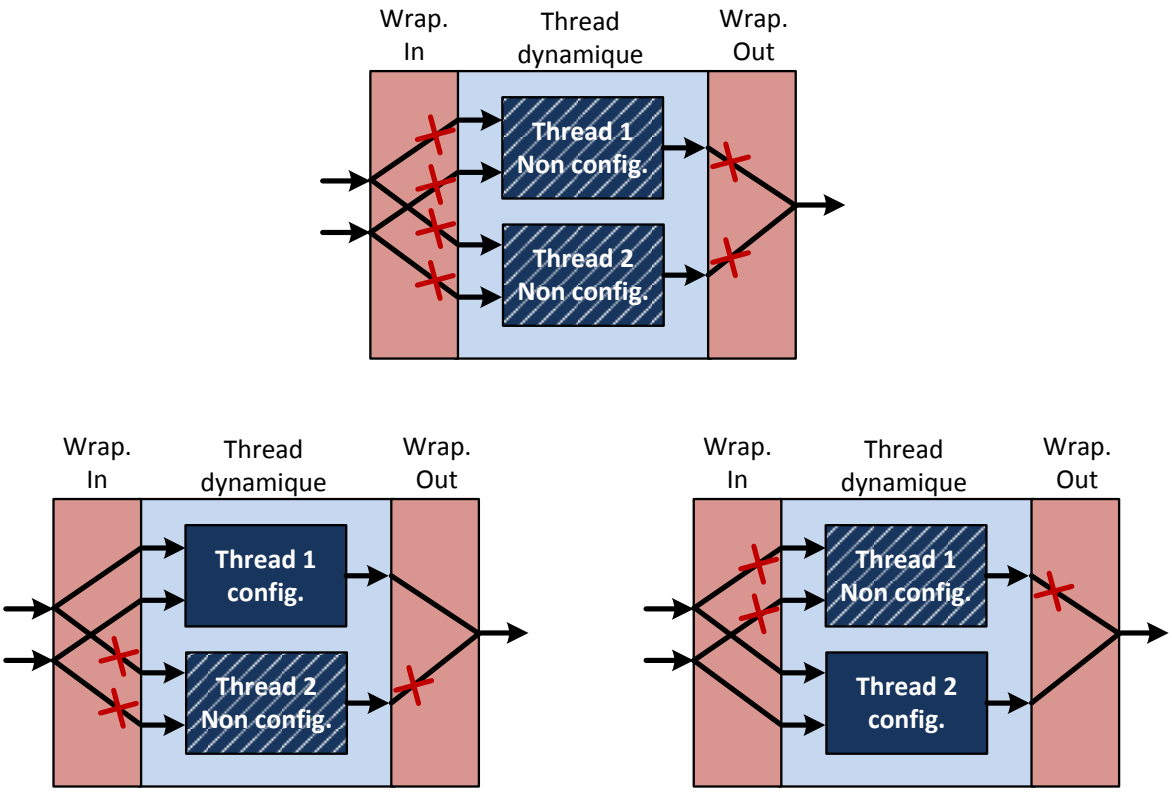


FIGURE 3.3 – Principe de fonctionnement des wrappers

L'utilisation de cette pratique impose la déclaration de l'ensemble des entrées/sorties de chaque fonction sur les wrappers. Cette étape est réalisée lors de la création du thread dynamique et de la déclaration des fonctions reconfigurables sur celui-ci.

3.3 Construction du modèle

L'alimentation du modèle est une étape cruciale pour son élaboration, cette phase est réalisée manuellement par l'utilisateur, et permet de retranscrire l'architecture et l'application que l'on souhaite tester et simuler. Les informations peuvent être rangées en deux groupes :

– Les informations sur l'application, servent à définir les tâches constitutives de l'application. On y retrouve par exemple, les temps d'exécution et les interactions entre les tâches. Ces informations permettent à notre modèle de générer le graphe de tâches de l'application et d'attribuer des priorités à chacune d'entres elles.

– Les informations sur l'architecture, sont utiles pour définir et dimensionner les éléments statiques du modèle, tels que le nombre de mémoires ou la taille des zones reconfigurables.

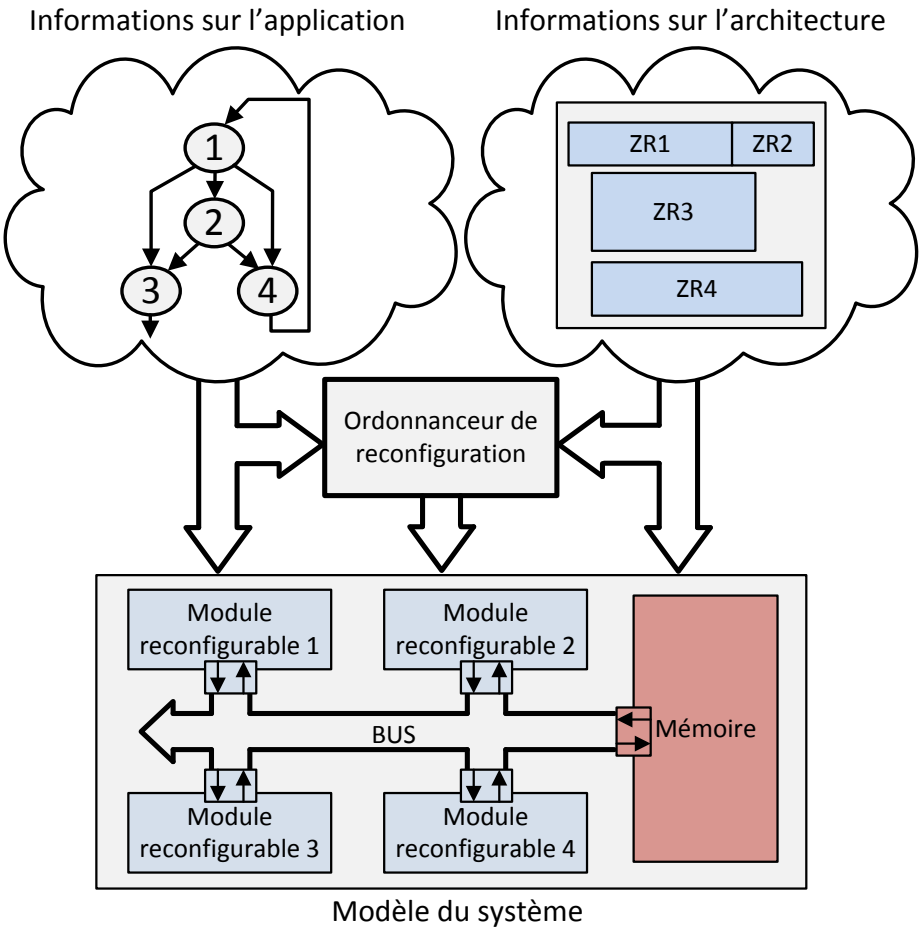


FIGURE 3.4 – Principe d'alimentation du modèle

La figure 3.4 montre le principe d'alimentation de notre modèle, et sera à la base du développement de ce chapitre. Dans la suite, nous reviendrons plus en détails sur les différents paramètres pouvant être entrés par l'utilisateur, ainsi que la façon d'utiliser ces informations pour façonner notre modèle.

3.3.1 Le module reconfigurable

Le module reconfigurable est l'élément de base de notre modèle, il sert à modéliser le comportement des zones reconfigurables. De la même façon que ces dernières, un module peut

supprimer ou instancier une tâche durant la phase de simulation, à condition bien sûr que l'ordonnanceur lui en envoi l'ordre, et que cette tâche ait été au préalable déclarée sur le module. Chaque module reconfigurable est contrôlé par un même ordonnanceur, son rôle est de prendre les décisions relatives au placement et à l'organisation des tâches, nous reviendrons plus en détails sur celui-ci par la suite.

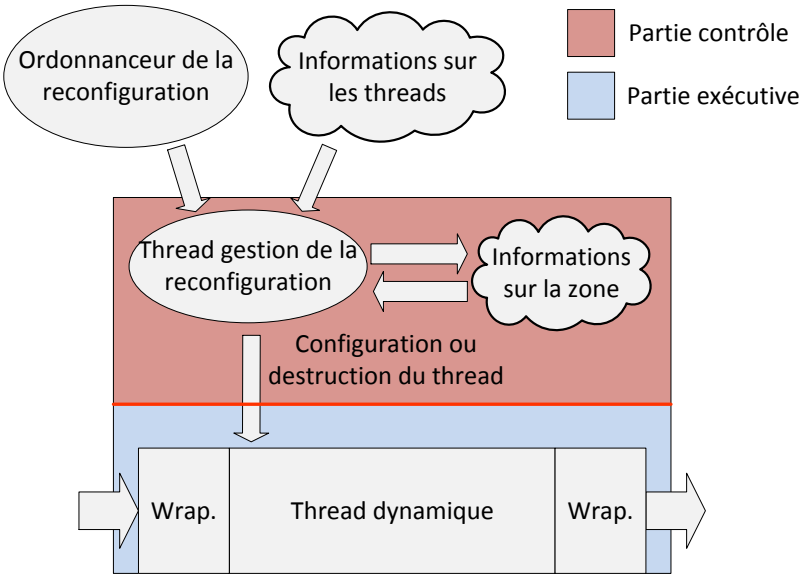


FIGURE 3.5 – Module reconfigurable

Afin de mieux cerner le fonctionnement du module reconfigurable, sur la figure 3.5, nous avons différencié en deux parties leurs mécanismes :

La partie contrôle (en rouge) gère la reconfiguration du module qui lui est associé, elle communique et reçoit ses requêtes directement de l'ordonnanceur externe. En fonction de ces ordres, il a la possibilité de détruire le thread en cours d'exécution sur le module, et de le remplacer par un autre présent dans la liste des threads déclarés. Le temps de reconfiguration simulé est directement proportionnel à la surface occupée par la zone, celui-ci est initialement intégré dans un espace mémoire lors de la création du module, avec d'autres informations telles que la tâche actuellement sur le module, le taux d'occupation de la zone, les temps totaux d'exécution et de reconfiguration depuis le début de la simulation, ainsi que d'autres informations nécessaires à l'utilisateur pour évaluer chaque module.

La partie exécutive (en bleu) modélise le comportement de la zone reconfigurable et l'exé-

cution de la tâche qui lui est attribuée. La modification du thread dynamique est initiée par la partie contrôle grâce à l'utilisation de la fonction SC_SPAWN vue précédemment.

Chaque module est donc composé de deux threads, un thread dynamique qui imite le fonctionnement d'une zone reconfigurable en modifiant son comportement au cours de l'exécution. Et un thread statique qui contrôle la zone, et simule les temps de reconfiguration en fonction de la surface qui lui est attribuée.

3.3.2 Caractérisation des éléments du modèle

Pour rendre notre modèle fidèle à l'architecture et l'application que l'on souhaite simuler, il est nécessaire de caractériser chacun de ses éléments avec le plus d'informations possibles. Celles-ci peuvent être issues directement des résultats obtenus lors des tests de performances réalisés sur les IP développées.

3.3.2.1 Caractérisation des tâches

Nous appelons tâche, les différentes fonctions ou IP constituant notre système, celles-ci peuvent être reconfigurables ou non, matérielle ou logicielle. Chaque tâche est déterminée par :

- Un temps d'exécution, correspondant au temps nécessaire à la tâche pour traiter une donnée.
- Une latence d'entrée, qui détermine le nombre de données nécessaires à la tâche avant de pouvoir lancer une exécution.
- Un débit d'entrée.
- Un débit de sortie.
- Le dimensionnement des ports d'entrée et de sortie. (taille des bus)
- La surface occupée par celle-ci, nous l'exprimons en Slices LUT, Slices Registers et DSP48.

3.3.2.2 Caractérisation des zones reconfigurables

Comme détaillé précédemment, la zone reconfigurable est représentée par le thread dynamique défini dans la partie exécutive de chaque module. Pour chaque thread dynamique créé, il est nécessaire de définir :

– La surface occupée par la zone reconfigurable, toujours exprimée en Slices LUT, Slices Registers et DSP48.
– Le temps de reconfiguration.

Les informations concernant la surface occupée par la zone permettent de calculer le temps de reconfiguration de celle-ci. En effet, ce temps ne dépend pas de la taille de la tâche, mais de la zone, une tâche occupant 50% de la zone et une autre occupant 90% mettront autant de temps à être configurées. Des études [RRMC07, DML11a] montrent que la compression des bitstreams de configuration permet de réduire leurs temps de transfert, et donc leur temps de reconfiguration global. Cette méthode étant encore expérimentale et peu pratiquée, nous ne la prendrons pas en compte dans notre étude.

3.3.3 La transaction

Le terme transaction est utilisé pour définir un échange de données entre plusieurs modules communicants entre eux, ou avec d'autres éléments statiques (Mémoire, périphérique, etc). Chaque transaction comprend une source, une destination et une quantité de données permettant d'acheminer ces dernières correctement.

3.3.4 Communication entre modules

Afin de prévenir la perte de données entre deux configurations, chaque transaction est effectuée à travers une mémoire partitionnée, via un bus de données commun comme illustré sur la figure 3.6.

Le partitionnement de la mémoire est réalisé en fonction du graphe de tâche fourni au modèle. Un premier processus découpe la mémoire en autant de tâches qu'en compte le graphe. Puis, pour chaque partition créée, l'algorithme déduit le nombre de transaction entrante, et génère une nouvelle sous partition pour chacune d'entre elles. Un exemple de partitionnement est illustré figure 3.7.

De cette façon, chaque transaction est mémorisée dans un espace mémoire bien défini. Lorsqu'une tâche configurée est prête à traiter de nouvelles données, une requête est déposée à l'arbitre du bus, celui-ci scrute l'espace mémoire correspondant et lui transmet autant de données que nécessaire pour réaliser une exécution. De la même manière, lorsque des données ont été

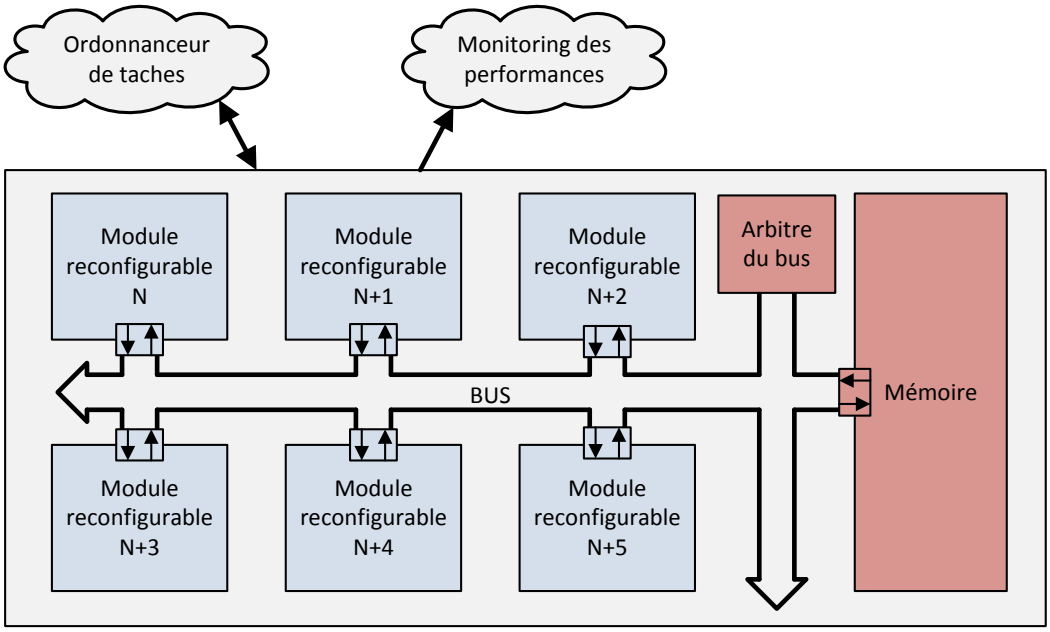


FIGURE 3.6 – Communication entre les élements du modèle

générés, une requête est envoyée à l'arbitre de bus pour lui spécifier l'espace mémoire à utiliser pour enregistrer la transaction.

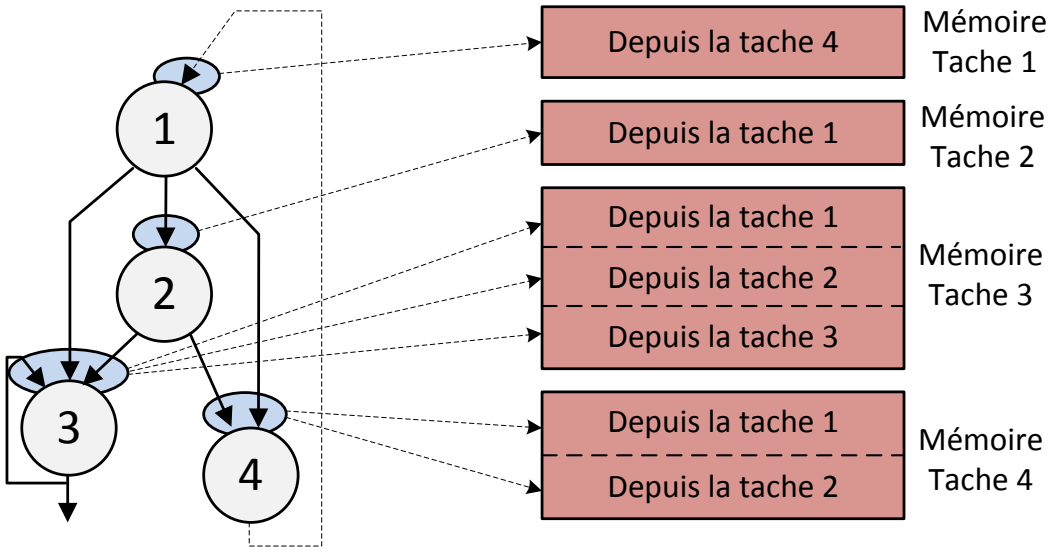


FIGURE 3.7 – Exemple de découpage de la mémoire

Un intérêt de créer des espaces mémoires pour chaque tâche est de rendre ceux-ci totalement indépendants des reconfigurations. Au cours de l'exécution, une tâche sera amenée à être

placée sur plusieurs zones reconfigurables, avec cette méthode, quel que soit l'emplacement de la tâche, celle-ci écrira toujours dans le même espace mémoire.

3.3.5 Ordonnanceur

L'ordonnancement de la reconfiguration dynamique est un problème à part entière, beaucoup de travaux ont déjà été réalisés sur ce thème [RRMC07]. Le choix d'un ordonnanceur est important, c'est lui, qui en cours d'exécution, va décider des instants de reconfiguration, et des tâches à reconfigurer. Du choix de l'algorithme d'ordonnancement, peut dépendre le bon fonctionnement de l'application, ainsi que le respect de ses contraintes.

La chaîne de transcodage vidéo qui est fortement contrainte par les flux vidéo acheminés en temps réel, a besoin d'un ordonnanceur qui prend particulièrement en considération la gestion des données. L'ordonnancement utilisé s'inspire d'un algorithme [CRGM11] s'exécutant en deux phases, la phase 'design-time' et la phase 'run-time' :

La phase 'design-time' intervient lors du paramétrage du modèle, avant l'exécution de la simulation. Cette étape consiste à scanner le graphe de l'application, pour attribuer une priorité de configuration à chaque tâche. La figure 3.8 montre l'attribution des priorités pour un exemple de graphe.

La priorité est calculée en additionnant le temps d'exécution des tâches avec la priorité des suivantes. Lorsqu'une tâche communique avec plusieurs autres tâches à la fois, c'est celle possédant la plus grande priorité qui est prise en compte. Lors du démarrage de l'algorithme, celui-ci recherche d'abord les tâches en fin de graphe (celles qui n'écrivent dans aucune autre tâche), ce sont elles qui auront la priorité la plus faible étant donné qu'aucune autre ne dépend d'elles, une priorité équivalente à leur temps d'exécution leur est alors attribuée.

Dans notre exemple, la tâche 5 a un temps d'exécution de 5, et une priorité de 5. On recherche les tâches communicantes avec elle, ici il s'agit des tâches 3 et 4, leur priorité est calculée en prenant leur temps d'exécution et en l'additionnant avec la priorité de la tâche 5, nous obtenons P(3) = 12 + 5 = 17 et P(4) = 21 + 5 = 26, et ainsi de suite jusqu'à arriver à la dernière tâche du graphe. A noter que pour le calcul de la tâche 2, c'est la priorité de la tâche 4 qui est utilisée puisque P(4) > P(3).

Cette première attribution des priorités permet à l'ordonnanceur de réaliser un premier rem-

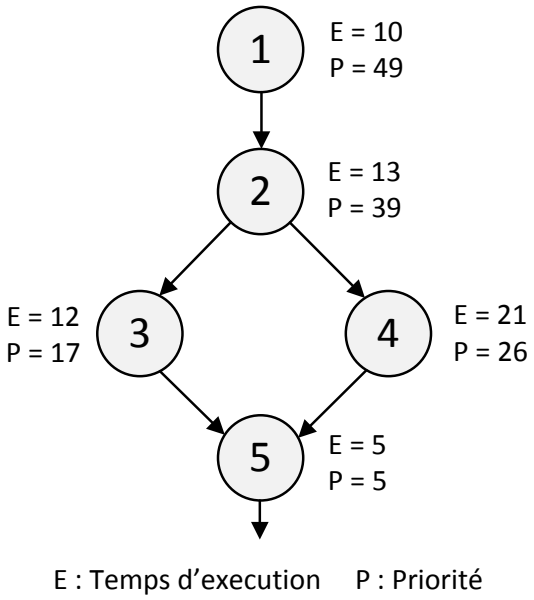


FIGURE 3.8 – Exemple d'attribution des priorités lors de la phase design-time

plissage des zones reconfigurables disponibles. Dans le cas par exemple, où nous ne souhaiterions travailler que sur trois zones reconfigurables, les tâches prioritairement configurées seraient les trois tâches ayant les priorités les plus élevées, c'est-à-dire les tâches 1, 2 et 4, à condition bien sûr que les trois zones reconfigurables soient dimensionnées de sorte à pouvoir accueillir toutes ces tâches en même temps. Dans le cas contraire, les tâches ne pouvant pas être configurées laisseraient leur place aux tâches de priorité directement inférieure, et seraient mise en attente jusqu'à la libération d'une zone adaptée à leur taille. Pour minimiser ce cas de figure, l'ordonnanceur cherche toujours à placer une tâche dans la plus petite zone disponible pouvant l'accueillir.

La phase 'run-time' intervient au cours de l'exécution, l'ordonnancement est modifié en temps réel pour prendre en considération les informations issues de la simulation. L'ordonnanceur a accès aux taux de remplissage des mémoires transactionnelles du modèle, ainsi, à tout moment il sait quelles tâches possèdent assez de données pour pouvoir être configurées et exécutées. A l'opposé, garder une tâche configurée lorsque celle-ci ne possède plus de donnée à traiter n'a pas d'intérêt, sa priorité est donc amoindrie.

La figure 3.9 montre un exemple de situation à résoudre par l'ordonnanceur, les étoiles correspondent au nombre d'exécutions qui peuvent être réalisées grâce aux données stockées

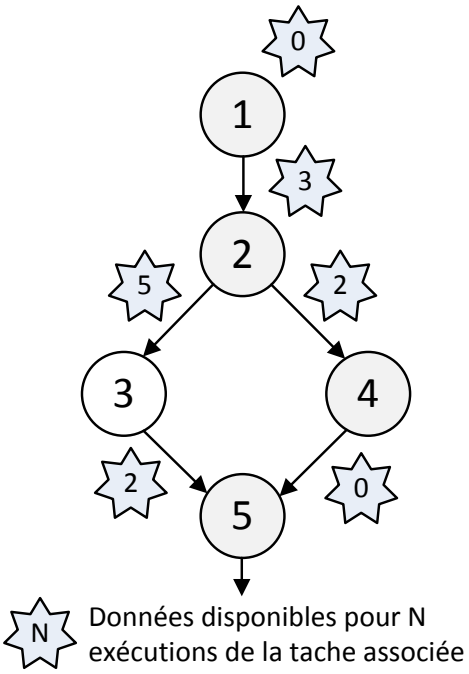


FIGURE 3.9 – Exemple d'attribution des priorités lors de la phase run-time

dans la mémoire transactionnelle. Comme pour la phase 'design time', l'ordonnanceur scrute la totalité du graphe pour affecter des priorités à chaque tâche, mais cette fois, ce sont les données stockées dans la mémoire qui sont prises en compte. Dès qu'une tâche configurée ne possède plus assez de données pour effectuer une exécution, une interruption est envoyée à l'ordonnanceur pour reconfigurer cette zone avec une nouvelle tâche. La priorité est donnée à la tâche ayant le plus de données en entrée. Lorsqu'une tâche reçoit des données issues de plusieurs tâches, c'est la mémoire possédant le moins de données qui est prise en compte. Dans notre exemple (figure 3.9), la tâche 5 possède deux et zéro exécutions, c'est donc le zéro qui est pris en compte donnant ainsi la plus basse priorité de reconfiguration à celle-ci. De la même manière, la tâche 1 ne possède aucune exécution, sa priorité est nulle également. L'ordre d'importance est donc : tâche 3, 2 puis 4. Comme pour le cas précédant, si aucune zone reconfigurable n'est assez grande pour accueillir la tâche prioritaire, c'est la tâche de priorité directement inférieure qui est configurée.

Les reconfigurations sont donc effectuées à chaque fois qu'une tâche a terminé toutes ses exécutions et ne possède plus les données nécessaires pour continuer à travailler. Afin de laisser le contrôle à l'utilisateur, celui-ci a la possibilité de modifier le seuil de reconfiguration pour

chacune des tâches. Au lieu d'attendre qu'une tâche ait consommé la totalité de ses données pour être reconfigurée, un seuil plus haut peut être attribué, ce qui permet de jouer sur le temps d'occupation des tâches sur le circuit, ainsi que sur la fréquence des reconfigurations.

3.3.6 Simulation et stratégie d'exploration

Notre modèle est orienté vers la vérification et la validation d'IP et d'architectures, pour cela, chaque simulation est réalisée de sorte à fournir un maximum d'informations. Les modules reconfigurables déclarés possèdent chacun leur propre système de contrôle, permettant de surveiller et de retranscrire au cours de la simulation, les modifications ou informations pertinentes à l'utilisateur. Il est possible par exemple, de surveiller les débits en entrée et en sortie de chaque blocs de traitement, les capacités mémoires utilisées, ou encore les taux d'occupation de chaque zone reconfigurable, etc.

De cette façon, en cas de défaillance de l'architecture, une analyse des relevés permet de repérer une IP, une zone reconfigurable, ou tout autre élément mal dimensionné qui briderait les performances souhaitées.

Les paramètres de l'élément incriminé peuvent ensuite être ajustés, jusqu'à l'obtention par simulation du comportement attendu. Pour cela, l'utilisateur spécifie au simulateur, l'élément et les paramètres qu'il souhaite libérer (un temps d'exécution, une latence, etc), ainsi que l'étendue et le pas d'exploration qu'il désir parcourir. Un lot de nouvelles simulations est alors rejoué pour chaque variation de paramètre, le nombre et le temps de simulation dépendent bien sûr du pas et de l'étendue de l'exploration. L'analyse des nouveaux résultats par l'utilisateur permet de définir les performances, et l'étendue des modifications à appliquer à l'élément inadapté.

Dans le cas où l'exploration sur un seul élément ne permet pas d'arriver aux résultats escomptés, d'autres investigations peuvent être menées sur deux éléments simultanément, puis trois, etc.

3.4 Modélisation de l'application : Le projet ARDMAHN

Nous avons vu dans cette première partie de chapitre que la modélisation de la reconfiguration dynamique était plus complexe qu'il n'y paraissait. L'ordonnancement des tâches reconfigurables en est une étape importante, mais celle-ci ne peut être faite correctement sans une

compréhension et une étude approfondie de l'architecture utilisée, mais surtout de l'application en elle même. Sans la connaissance des contraintes et des performances de l'application, il est impossible de savoir si les scénarios simulés seront crédibles et réalisables sur FPGA.

Le transcodage vidéo appartient à un domaine régi par de fortes contraintes temporelles, et nécessitant d'importantes ressources de calcul et de mémorisation. Il en fait un sujet d'application parfait pour notre modèle. Dans la suite de ce chapitre, nous reviendrons sur le projet ARDMAHN, et verrons de quelle façon nous allons l'utiliser pour alimenter et valider notre modèle.

3.4.1 Présentation

Depuis plusieurs années les set-top box ont littéralement envahis le marché, en France, la plupart des fournisseurs d'accès à internet proposent en supplément de leurs services, cette fameuse box permettant à l'utilisateur de recevoir la télévision par ADSL. Le projet ARDMAHN a pour but de concevoir une nouvelle génération de set-top box. La finalité recherchée est de proposer une solution capable d'accepter n'importe quel type de flux en entrée et de le retranscrire sur n'importe quel type d'appareil de lecture. De plus, le transcodage pourra être fait sur plusieurs flux simultanément grâce à l'implémentation de toute la partie traitement vidéo sur accélérateur matériel FPGA et à l'utilisation de la technique de reconfiguration dynamique.

La figure 3.10 illustre une première proposition d'architecture par le consortium du projet. Au delà de l'aspect transcodage vidéo, un processeur de type Atom est en charge de toute la partie encapsulation vidéo pour permettre la réception et l'envoi des flux sur le réseau. Ce processeur est connecté à une mémoire DDR via un bus de communication PCI express, dans laquelle les flux en attente de traitement sont stockés. Cette mémoire est constamment alimentée par le processeur Atom, de même que lorsqu'une donnée est prête à être renvoyée sur le réseau, celle-ci est directement prise en charge par le processeur. Pour cette raison, nous nous permettons de nous abstenir de prendre en considération toute la partie logiciel dans notre modélisation, et de nous focaliser sur les zones reconfigurables, la mémoire et les communications internes au FPGA.

Quatre zones reconfigurables sont définies pour la configuration de deux chaines de transcodage complètes simultanément. Dans un premier temps celles-ci sont indépendantes l'une

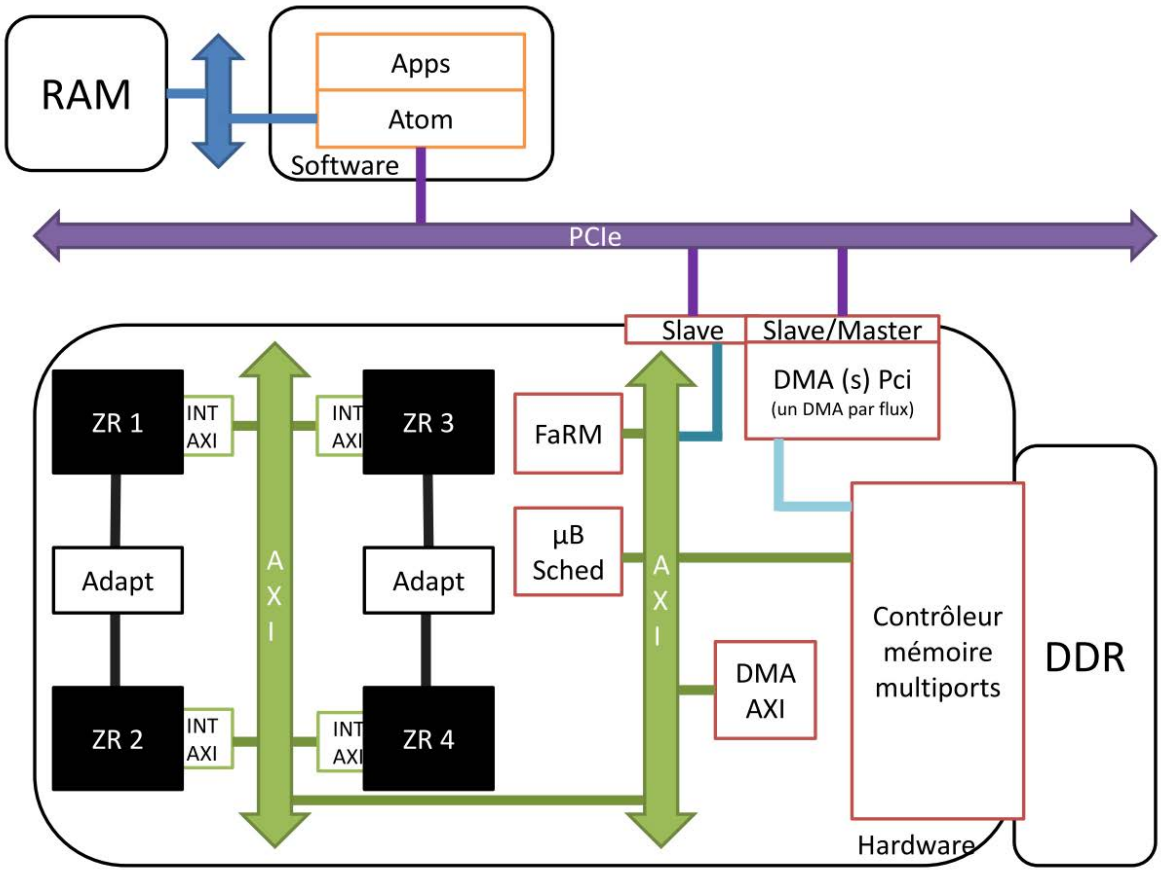


FIGURE 3.10 – Architecture du projet ARDMAHN proposée par le consortium

de l'autre, un flux décodé sur la première chaîne ne peut être adapté et encodé sur la seconde chaine.

3.4.2 Chaine de transcodage

La box devra être capable de traiter des flux codés MPEG-2 et H.264, pour cela, des blocs de traitement adéquats ont été développés (Voir Chapitre suivant). En plus de cette adaptabilité au niveau des codecs, l'intérêt d'un transcodeur réside dans le fait de pouvoir changer les caractéristiques d'un flux vidéo pour l'adapter au terminal de lecture. Par exemple, dans le cas où la box recevrait un flux haute définition H.264 de résolution 1920x1080, pour une lecture sur un appareil mobile (Smartphone par exemple) ne pouvant uniquement recevoir des flux en définition standard MPEG-2 de résolution 352x288, le transcodeur devra tout d'abord décoder le flux HD entrant, modifier la résolution et le frame rate de celui-ci grâce à une étape d'adaptation, et

pour finir, encoder de nouveau le flux dans le codec MPEG-2 adapté au terminal. La chaine de transcodage se décompose donc en trois parties :

– Décodage du flux d'entrée.

– Adaptation du flux aux caractéristiques du terminal de lecture.

– Encodage du flux adapté.

La reconfiguration dynamique est utilisée pour le décodage et l'encodage, pour chaque chaîne, deux zones peuvent recevoir soit des blocs de traitement H.264, soit des blocs de traitement MPEG-2 en fonction des besoins. Quant à la partie adaptation, au vue de son fonctionnement, il a été décidé de la laisser statique et de jouer sur ses paramètres en fonction de l'opération à effectuer.

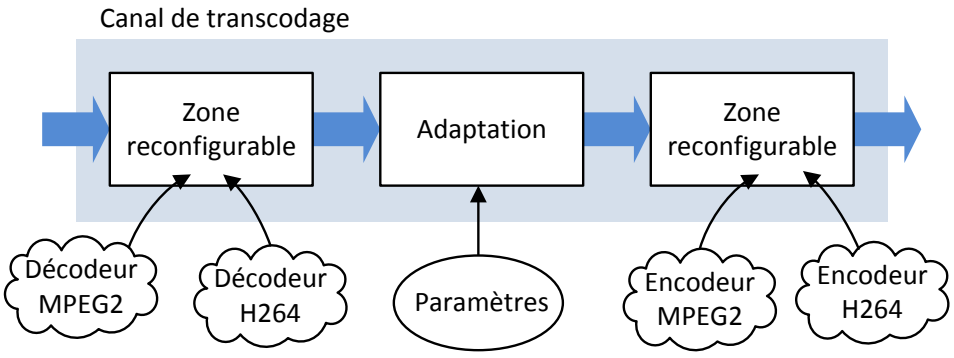


FIGURE 3.11 – Canal de transcodage reconfigurable

Il aurait également été possible d'affiner la granularité de reconfiguration en partitionnant les décodeurs et encodeurs afin de placer leurs fonctions internes (Quantification, transformée, codage entropique, etc) sur des zones reconfigurables plus petites et plus rapides à reconfigurer. Toutefois, cette méthode impose de travailler sur de petits fragments d'image pour éviter l'encombrement de données entre les fonctions de traitement, et d'augmenter la fréquence des reconfigurations. Or le temps d'exécution d'une tache étant infime en comparaison de son temps de reconfiguration (environ 1 pour 1000), il est préférable d'augmenter la granularité et de traiter un maximum de données entre chaque reconfiguration. Le temps de reconfiguration devient alors négligeable par rapport au temps de traitement des tâches.

La figure 3.11 montre un exemple de canal de transcodage composé des deux zones reconfigurables et du bloc d'adaptation.

Chaque canal de transcodage est capable de traiter plusieurs flux vidéo simultanément grâce à la reconfiguration dynamique. Toutefois, plus le nombre de flux augmente, plus les contraintes de traitement temps réel seront difficiles à respecter. Les performances du canal dépendent d'un grand nombre de paramètres tels que les caractéristiques des flux vidéo, les temps de reconfiguration des zones reconfigurables, et bien sûr les performances de nos blocs de traitement vidéo. Nous verrons dans la suite du manuscrit l'intégration de ces paramètres dans notre modèle et des estimations de performances du canal de transmission.

3.4.3 Définition des cas d'utilisation

Les cas d'utilisation représentent l'ensemble des scénarios qui peuvent être rencontrés par la box lors de son utilisation. Etant donné le nombre important de résolutions existantes, nous allons les regrouper en deux catégories : les "flux SD" représentent toutes résolutions autorisées par le standard MPEG-2, c'est-à-dire les résolutions inférieures ou égales à 720x576 pixels. Et "les flux HD", tous les flux supérieurs à la résolution SD jusqu'à la résolution 1920x1080 qui correspond au Full HD, la résolution maximum que peut traiter le standard H.264.

Pour les mêmes raisons, nous appellerons "mono-flux" les cas dans lesquels la box ne reçoit ou n'envoie qu'un seul flux vidéo. Et "multi-flux", le fait de traiter plusieurs flux en parallèle en entrée ou en sortie, dans ce cas, le nombre de flux parallèles doit au moins être égal à deux.

A partir de ces éléments, il est alors possible de répartir les différents cas d'utilisation en quatre catégories que nous allons lister dans la suite de ce paragraphe.

3.4.3.1 Les cas d'utilisation mono-flux

Les cas de type mono-flux sont les plus simples car ils ne nécessitent aucune modification de codec pendant l'exécution. Il s'agit typiquement d'un utilisateur qui souhaite regarder un seul flux vidéo (figure 3.12).

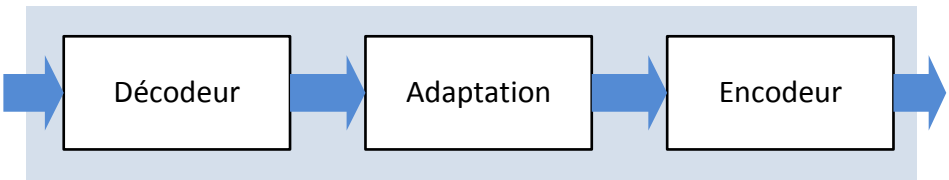


FIGURE 3.12 – Canal de transcodage pour un cas d'utilisation mono-flux

TABLE 3.1 – Scénarios possibles pour un cas d'utilisation mono-flux

Scénarios	Codage d'entrée	Codage de sortie	Reconfiguration du décodeur	Reconfiguration de l'encodeur
1	MPEG-2	MPEG-2	Non	Non
2	MPEG-2	H.264	Non	Non
3	H.264	MPEG-2	Non	Non
4	H.264	H.264	Non	Non

Ici, la reconfiguration dynamique est inutile puisqu'un transcodeur statique suffit (tableau 3.1), à moins bien sûr que l'utilisateur décide de changer le flux vidéo, ou de terminal.

3.4.3.2 Les cas d'utilisation mutli-flux en entrée

Pour ce second cas, plusieurs flux arrivent simultanément en entrée de la box et sont envoyés sur un seul et même terminal (figure 3.13). Cet exemple peut être rencontré lorsque l'utilisateur veut regarder en plus du flux vidéo principal, un second flux au format PiP (Picture in Picture).

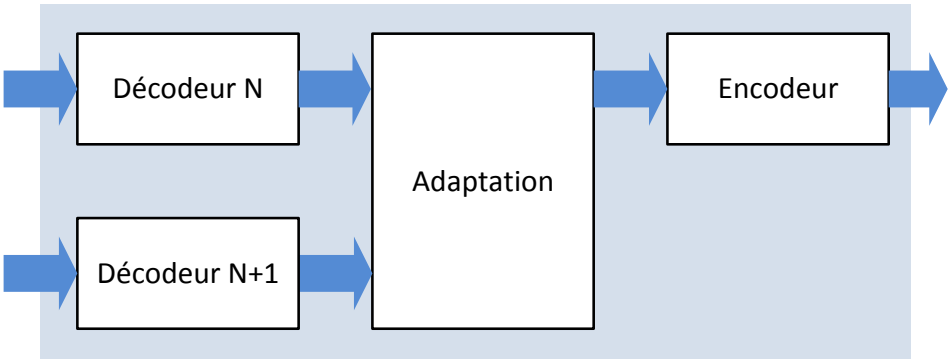


FIGURE 3.13 – Canal de transcodage pour un cas d'utilisation multi-flux en entrée

Lorsque les flux d'entrée sont communs (même codec), la reconfiguration n'est pas nécessaire, un simple multiplexage est effectué pour envoyer alternativement les flux entrants sur le même bloc de décodage configuré (tableau 3.2). Par contre, la reconfiguration dynamique est nécessaire lorsque les sources sont mixtes (H.264 et MPEG-2) et vient s'appliquer en supplément du multiplexage. L'encodeur n'est pas modifié quant à lui et est utilisé pour encoder les deux flux.

TABLE 3.2 – Scénarios possibles pour un cas d'utilisation multi-flux en entrée

Scénarios	Codage d'entrée	Codage de sortie	Reconfiguration du décodeur	Reconfiguration de l'encodeur
1	Commun	MPEG-2	Non	Non
2	Commun	H.264	Non	Non
3	Mixte	MPEG-2	Oui	Non
4	Mixte	H.264	Oui	Non

3.4.3.3 Les cas d'utilisation multi-flux en sortie

Ce cas de figure est l'opposé du précédant, cette fois-ci une seule source vidéo est envoyée sur la box pour être visualisée sur deux terminaux simultanément (figure 3.14).

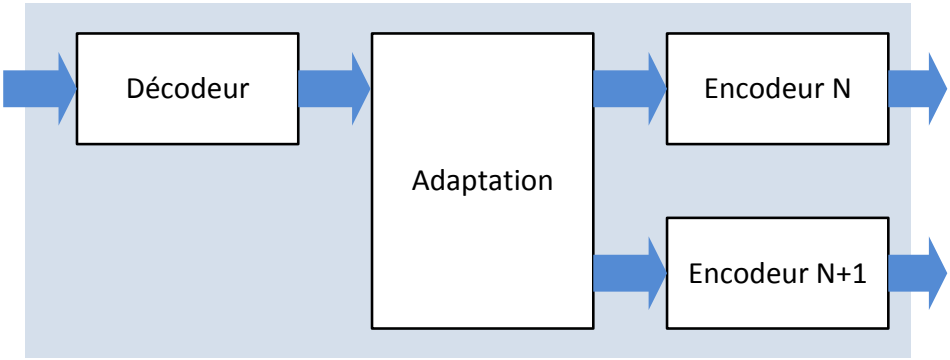


FIGURE 3.14 – Canal de transcodage pour un cas d'utilisation multi-flux en sortie

TABLE 3.3 – Scénarios possibles pour un cas d'utilisation multi-flux en sortie

Scénarios	Codage d'entrée	Codage de sortie	Reconfiguration du décodeur	Reconfiguration de l'encodeur
1	MPEG-2	Commun	Non	Non
2	H.264	Commun	Non	Non
3	MPEG-2	Mixte	Non	Oui
4	H.264	Mixte	Non	Oui

Cette fois c'est l'encodeur qui nécessite une reconfiguration lorsque les terminaux de lecture ne prennent pas en compte les mêmes codecs (tableau 3.3). Les scénario 9 et 10 sont des cas à part, les codecs de sortie étant communs nous pourrions croire que la vidéo doit être encodée

une seule fois puis envoyée à tous les terminaux utilisant le même codec, mais c'est sans prendre en compte le module d'adaptation. Puisque les terminaux peuvent utiliser les mêmes codecs, mais ne pas avoir les mêmes résolutions, il est nécessaire de réaliser un encodage spécifique pour chaque flux adapté.

3.4.3.4 Les cas d'utilisation multi-flux en entrée et sortie

Le dernier cas est une combinaison des deux précédents, il s'agit de décoder plusieurs flux entrant et de les envoyer sur plusieurs encodeurs (figure 3.15). Il correspond à plusieurs utilisateurs qui souhaitent chacun visualiser un flux vidéo sur leur propre terminal. La reconfiguration dynamique intervient alors à la fois sur le décodeur et sur l'encodeur.

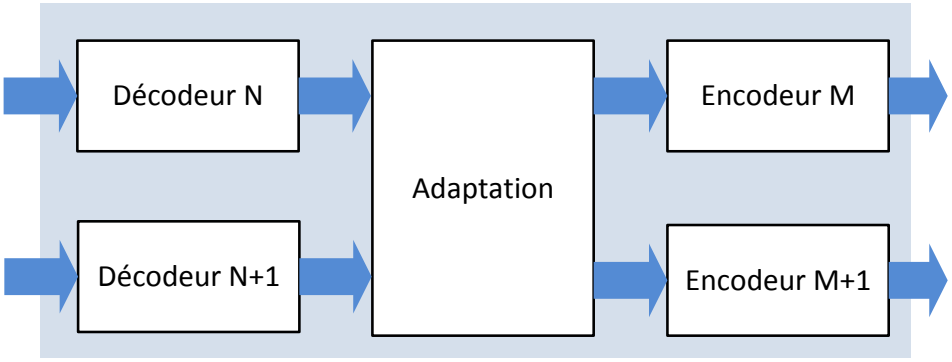


FIGURE 3.15 – Canal de transcodage pour un cas d'utilisation multi-flux en entrée et sortie

TABLE 3.4 – Scénarios possibles pour un cas d'utilisation multi-flux en entrée et sortie

Scénarios	Codage d'entrée	Codage de sortie	Reconfiguration du décodeur	Reconfiguration de l'encodeur
1	Commun	Commun	Non	Non
2	Commun	Mixte	Non	Oui
3	Mixte	Commun	Oui	Non
4	Mixte	Mixte	Oui	Oui

Tout comme les deux cas précédents, un utilisateur peut être en mesure de vouloir regarder plusieurs flux simultanément sur son terminal (Cas du mutli-flux en entrée et du PiP). Ou bien, plusieurs utilisateurs souhaitant visualiser la même vidéo avec des codecs communs mais pas les mêmes formats (tableau 3.4). (Cas du multi-flux en sortie et de l'adaptation).

3.5 Conclusion

Dans ce chapitre, nous avons vu les spécificités du transcodeur vidéo et l'architecture proposée par le consortium, celle-ci est composée de quatre zones reconfigurables permettant l'implémentation de deux canaux de transcodage simultanément. Quatre types de cas d'utilisation pouvant être rencontrés lors de l'utilisation d'un canal ont été définis. Afin de valider la faisabilité de chacun d'entre eux et de définir leurs besoins en terme de ressources, nous avons modélisé un canal de transcodage et simulé chacun des quatre cas d'utilisation. Nous verrons les résultats obtenus dans la suite du manuscrit. Mais avant toute chose et afin d'alimenter correctement notre modèle, il est nécessaire de connaître les paramètres de l'application.

Dans le chapitre suivant, nous verrons en détails la façon avec laquelle nous avons développé les blocs de traitement vidéo pour le codec H.264, ainsi que les résultats obtenus. Grâce à ces nouvelles données, nous pourrons alimenter le modèle et réaliser les premières simulations du canal de transcodage.

Chapitre 4

Caractérisation des IP

Sommaire

4.1 Introduction 70

4.2 Architecture proposée 71

4.2.1 Modèle de l'encodeur H.264 71

4.2.2 Mise en ordre des données avant traitement 71

4.2.3 Transformée entière 71

4.2.4 Quantification 74

4.2.5 Prédiction intra 77

4.2.6 Codage de l'information 80

4.2.7 Boucle de contre réaction 81

4.3 Performances 84

4.3.1 Performances temporelles 84

4.3.2 Performances en surface 85

4.4 Validation des paramètres par implémentation 86

4.4.1 Architecture standalone 87

4.4.2 Transcodeur MPEG-2/H.264 91

4.4.3 Utilisation de la reconfiguration dynamique 92

4.5 Conclusion 95

4.1 Introduction

Dans le cadre du projet ARDMAHN, nous disposions initialement des IP de transcodage MPEG-2, ainsi que du bloc d'adaptation, développés par notre partenaire de projet, la société VIOTECH. Les IP H.264 quant à elles, ont dues être développées durant ma thèse. Ce développement a été réalisé en adéquation avec l'étude des IP MPEG-2, de sorte à obtenir une certaine homogénéité dans l'interfaçage des deux standards vidéo. L'utilisation d'une bibliothèque open source H.264 aurait rendu ce travail bien trop compliqué : récupérer, analyser, et modifier des sources VHDL est une opération longue et complexe. Ayant des notions du standard H.264 grâce aux précédents travaux réalisés dans ce domaine au sein du laboratoire, nous avons préféré développer nos propres IP.

Ce chapitre se décompose en deux parties, tout d'abord nous étudierons chaque IP développée et expliquerons notre démarche. Puis dans un second temps, nous les validerons en simulation, puis en implémentation, sur le FPGA choisi par le projet. Ces résultats d'implémentations seront utiles pour la caractérisation et l'alimentation de notre modèle.

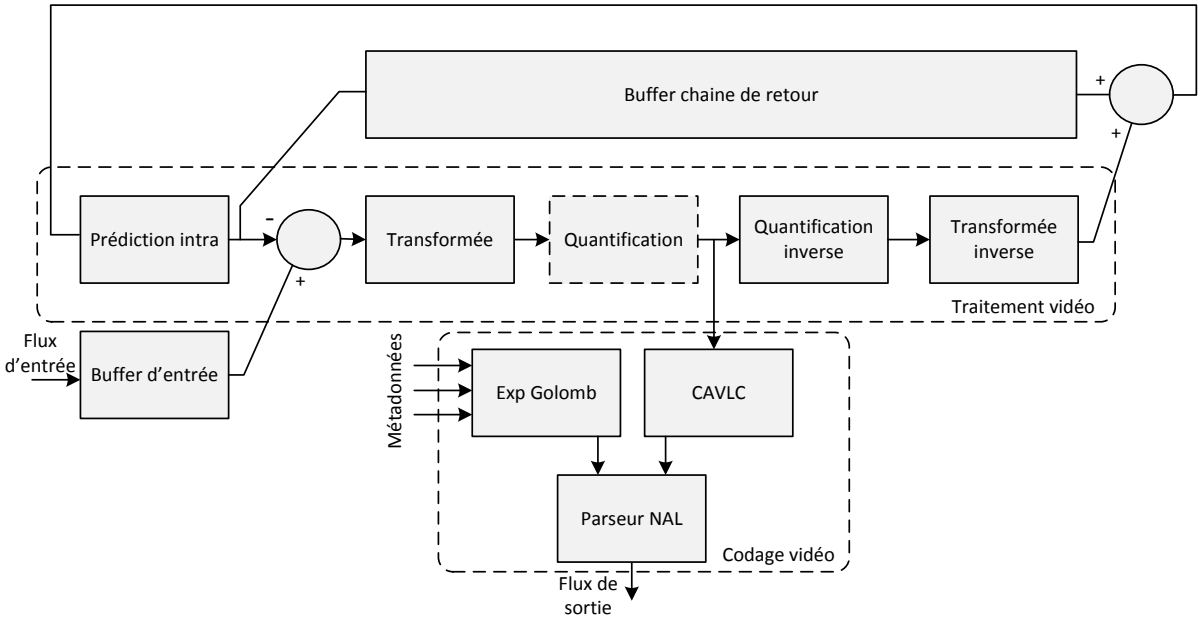


FIGURE 4.1 – Architecture de l'encodeur H.264

4.2 Architecture proposée

4.2.1 Modèle de l'encodeur H.264

La figure 4.1 montre le schéma global de l'encodeur H.264 développé, et les blocs de traitement que nous détaillerons dans la suite de ce chapitre.

4.2.2 Mise en ordre des données avant traitement

Les algorithmes de traitement vidéo ne travaillent pas tous avec le même format de données. H.264 par exemple, peut traiter des blocs de données 4x4 ou 8x8 pixels, alors que MPEG-2 ne travaille que sur un format 8x8 pixels. Il est nécessaire de découper les macroblocs en entrée de la chaîne avant de commencer le traitement. cette étape est réalisée par le buffer d'entrée, les macroblocs sont reçus et mémorisés, puis découpés pour correspondre aux formats demandés par les blocs de traitement.

4.2.3 Transformée entière

La transformée entière est une dérivée de la transformée en cosinus discret utilisée dans de nombreux codecs vidéo tels que MPEG-2. La différence entre ces deux versions vient essentiellement du fait que la transformée entière ne travaille qu'avec des coefficients entiers, la complexité de l'algorithme est alors réduite puisque les opérateurs de division et de multiplication sont remplacés par des opérateurs d'addition et de décalages binaires.

4.2.3.1 Matrice de passage pour la DCT

Considérons un bloc de données X, qui est en fait une matrice bidimensionnelle de taille 4x4, et F sa transformée à travers une DCT classique.

$$F = A.X.A^T \tag{4.1}$$

Où A est la matrice de transformation, et A^T sa transposée :

$$A = \begin{pmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{pmatrix} \text{ Avec : a = 0.5, b = 0.6532, c = 0.2706}$$

4.2.3.2 Matrice de passage pour la Transformée entière

A partir de cette matrice de transformation utilisée par la DCT, il est possible de trouver la matrice équivalente pour la transformée entière, en la multipliant par un facteur 2,5 et en prenant l'arrondi entier le plus proche, ainsi, la matrice de transformation pour une transformée entière devient :

$$Cf = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{pmatrix}$$

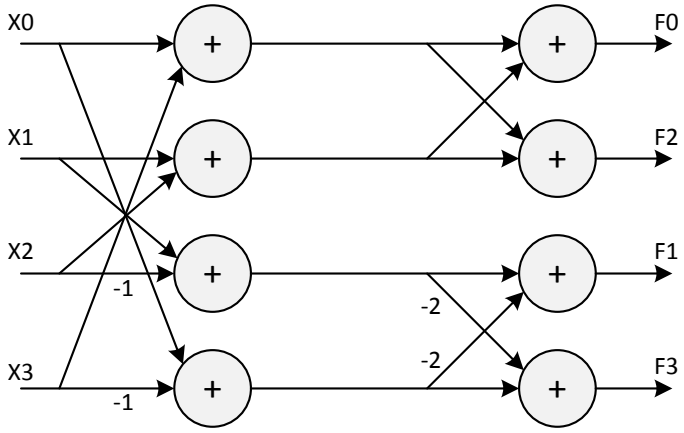
Comme on peut le constater, la nouvelle matrice est relativement simple à mettre en œuvre, et ne requière que des additions et des décalages.

4.2.3.3 Architecture utilisée et latence

L'architecture que nous avons utilisée pour la transformée entière repose sur une implémentation dite "butterfly" ou "papillon" (figure 4.2). Celle-ci à la capacité de traiter quatre coefficients en parallèle. Étant donné que la matrice de passage doit être appliquée sur l'ensemble des coefficients du bloc 4x4, ainsi que sur sa transposée, l'opérateur butterfly doit alors être répété huit fois de suite pour chaque bloc 4x4 : quatre fois sur les lignes du bloc, puis quatre fois sur ses colonnes (transposée). Les résultats des quatre premiers passages sont mémorisés pour être renvoyés en entrée de l'opérateur pour les quatre passages suivants.

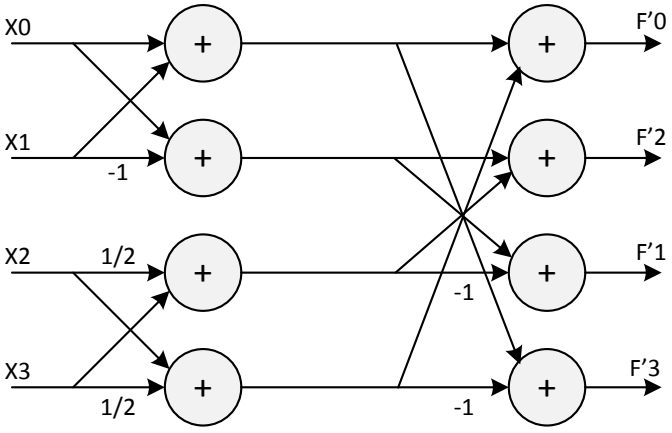
La transformée inverse (figure 4.3) permet de retrouver les coefficients originaux du bloc 4x4, sans aucune perte de donnée.

Bien qu'intéressante au niveau des ressources nécessaires pour sa mise en œuvre, l'utilisation répétée du même opérateur butterfly pour le traitement d'un seul bloc 4x4 engendre une latence non négligeable sur le système. La latence globale d'une architecture série telle que



$$\begin{pmatrix} F0 \\ F1 \\ F2 \\ F3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{pmatrix} \begin{pmatrix} X0 \\ X1 \\ X2 \\ X3 \end{pmatrix}$$

FIGURE 4.2 – Architecture Butterfly utilisée pour la transformée entière



$$\begin{pmatrix} X0 \\ X1 \\ X2 \\ X3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{pmatrix} \begin{pmatrix} F'0 \\ F'1 \\ F'2 \\ F'3 \end{pmatrix}$$

FIGURE 4.3 – Architecture Butterfly utilisée pour la transformée entière inverse

celle-ci est de 16 cycles d'horloge. Et le temps nécessaire pour le traitement d'un bloc 4x4 complet est de 32 cycles, comme le montrent les simulations réalisées (figure 4.4).

FIGURE 4.4 – Latence engendrée par la transformée entière et son inverse

4.2.4 Quantification

La classification fréquentielle réalisées par la transformée entière, tend à créer de grands écarts de valeurs entre les coefficients d'un même bloc de données. La quantification, qui peut être vue comme un filtre, est utilisée pour réduire ces écarts, en supprimant les coefficients dont la valeur serait trop basse. De paire avec le codeur entropique, cette opération permet d'obtenir un taux de compression intéressant. Le degré de filtrage de la quantification peut être ajusté en jouant sur le coefficient de quantification (QP). Un coefficient élevé filtrera un plus grand nombre de coefficients, en contre partie, la perte d'informations est définitive.

4.2.4.1 Sélection des paramètres de quantification

La quantification d'un bloc de coefficients passe par plusieurs étapes, et dépend de plusieurs variables. Le Qp, tout d'abord, permet de sélectionner les facteurs de multiplication (Qm) appliqués à chaque coefficient du bloc à traiter.

Considérons X(i,j), la matrice de coefficients issue de la transformée entière, et A(i,j) la matrice de multiplication pour laquelle :

- A correspond aux coefficients devant être multipliés par QmA ou QmiA,
- B correspond aux coefficients devant être multipliés par QmB ou QmiB,
- C correspond aux coefficients devant être multipliés par QmC ou QmiC.

$$\text{A(i,j)} = \begin{pmatrix} A & C & A & C \\ C & B & C & B \\ A & C & A & C \\ C & B & C & B \end{pmatrix}$$

L'intensité du Qm sélectionné dépend directement de la valeur du Qp appliqué en entrée. Les tableaux 4.1 et 4.2 montrent les différentes combinaisons possibles. Les valeurs supérieures à

cinq ne sont pas représentées, mais peuvent être retrouvées facilement : tous les six Qp, la valeur de Qm est divisée par deux (ou multipliée dans le cas de la quantification inverse).

TABLE 4.1 – Sélection des coefficients de multiplication (Qm) en fonction de Qp pour une quantification

Qp	QmA	QmB	QmC
0	13107	5243	8066
1	11916	4660	7490
2	10082	4194	6554
3	9362	3647	5825
4	8192	3355	5243
5	7282	2893	4559

TABLE 4.2 – Sélection des coefficients de multiplication (Qm) en fonction de Qp pour une quantification inverse

Qp	QmiA	QmiB	QmiC
0	10	16	13
1	11	18	14
2	13	20	16
3	14	23	18
4	16	25	20
5	18	29	23

Nous pouvons alors noter l'équation mathématique globale de la quantification Y(i,j) telle que :

$$Y_{(i,j)} = sign(X_{(i,j)}) \times (|X_{(i,j)}| \times (A_{(i,j,Q_m)}) >> (17 + Q_e)) \tag{4.2}$$

Où Qe est le décalage binaire de Qm tel que :

$$Q_e = \frac{Q_p}{6} \tag{4.3}$$

4.2.4.2 Architecture utilisée et latence

L'architecture utilisée pour la quantification est relativement simple, elle repose sur un registre regroupant l'ensemble des facteurs de multiplication Qm (figure 4.5). En fonction de la position (i,j) du coefficient à traiter et de la valeur de Qp, le Qm correspondant est appliqué au coefficient. Le nombre de décalage Qe quand à lui est assigné en divisant Qp par 6. Pour simplifier cette opération et ne pas avoir à réaliser de division, les valeurs de Qe sont au préalable chargées dans des registres et sont sélectionnées en testant la valeur de Qp. Le décalage quant à lui est réalisé parallèlement par un mapping des bus de données plus ou moins important en fonction de Qe.

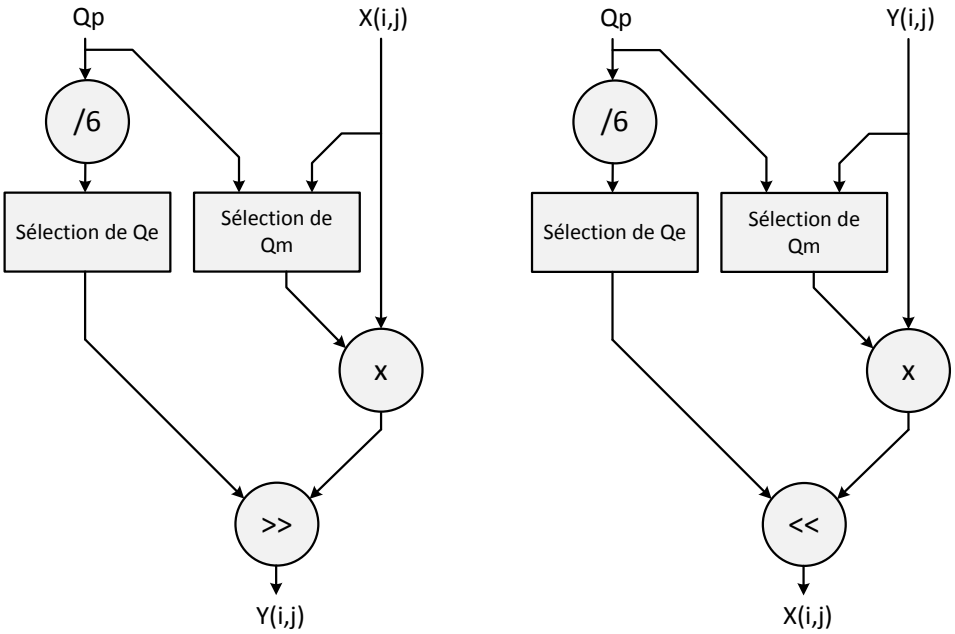


FIGURE 4.5 – Architecture utilisée pour la quantification et son inverse

Chaque coefficient demande 4 cycles d'horloge pour être quantifié, ce qui justifie la latence globale obtenue pour cette architecture (figure 4.6), tandis que le temps de traitement d'un bloc 4x4 prendra 20 cycles.

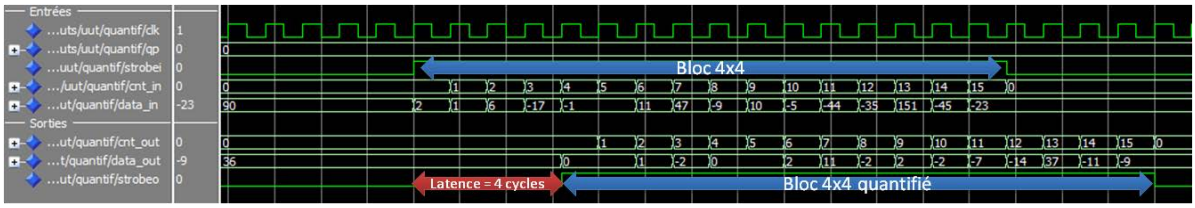


FIGURE 4.6 – Latence engendrée par la quantification et son inverse

4.2.5 Prédiction intra

La prédiction intra est la première opération de compression dans la chaîne d'encodage, c'est elle qui est chargée de traiter les redondances spatiales dans l'image, avant de transmettre les résidus à la transformée entière. Au lieu de travailler sur les pixels bruts de l'image, la prédiction crée une image prédite à partir d'un algorithme spatial (prédiction intra) et calcule la différence entre ces deux représentations (origine – prédiction).

Contrairement au reste de la chaîne H.264, la prédiction intra peut travailler aussi bien sur des blocs 4x4 que sur des blocs 8x8 et ce, sur une même image. Le traitement 4x4 est utilisé sur les parties de l'image regroupant le plus de détails, tandis que le traitement 8x8 est utilisé sur les parties de l'image moins détaillées (là où les variations de pixels sont les moins importantes), ainsi que sur l'ensemble de la chrominance.

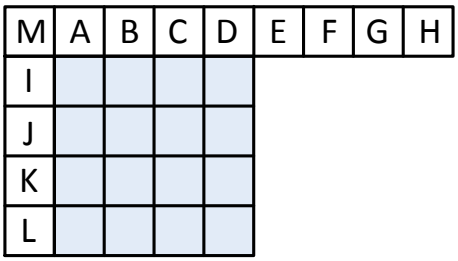


FIGURE 4.7 – Bloc prédit grâce aux pixels adjacents, notés de A à M

Pour un traitement 4x4, l'algorithme de prédiction à besoin des treize pixels adjacents du bloc en cours, neuf au dessus, et quatre à gauche (figure 4.7).

4.2.5.1 Sélection du mode de prédiction

Neuf modes de prédiction peuvent être utilisés pour la prédiction des blocs 4x4 et sont détaillés dans la figure 4.8, contre quatre modes pour la prédiction 8x8. Pour sélectionner le mode qui sera utilisé, chaque bloc de données doit d'abord être prédit à travers les neufs modes, puis un algorithme de SAD (Sum of Absolute Differences) réalise une comparaison entre le bloc d'origine et les neufs résultats pour déterminer le mode le plus adapté. Lors de l'étape inverse, le bloc d'origine peut être retrouvé grâce aux résidus et au mode à utiliser.

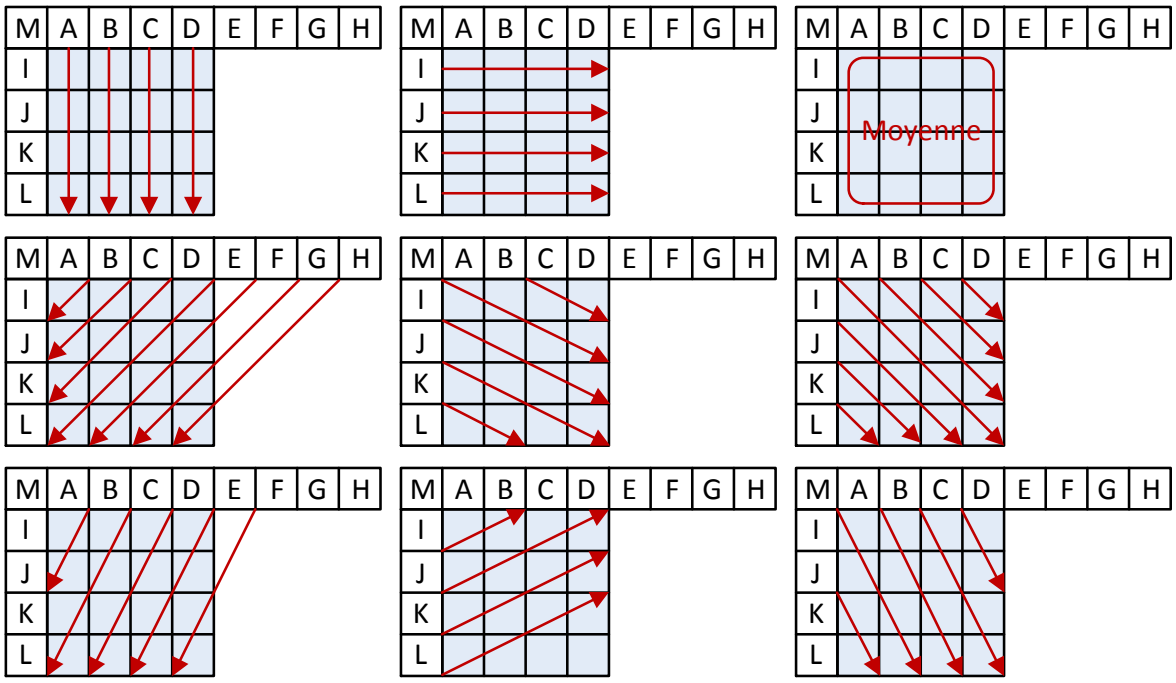


FIGURE 4.8 – Les 9 modes de prédiction utilisés

Dans certains cas, l'utilisation de l'algorithme SAD n'est pas nécessaire. Généralement, le calcul des modes est réalisé lors du premier encodage de la vidéo, puis ces derniers sont transmis avec le flux encodé. Dans le cas de transcodeurs grand public par exemple, la détermination des modes est faite par le fournisseur de contenus avant l'émission du flux vers l'abonné. Ils sont alors accessibles dans l'entête, et sont décodés en même temps que la vidéo.

4.2.5.2 Architecture utilisée et latence

L'opération de prédiction intra se décompose en plusieurs étapes (figure 4.9). Une mémorisation des pixels permet de sélectionner les neufs coefficients adjacents au bloc à prédire.

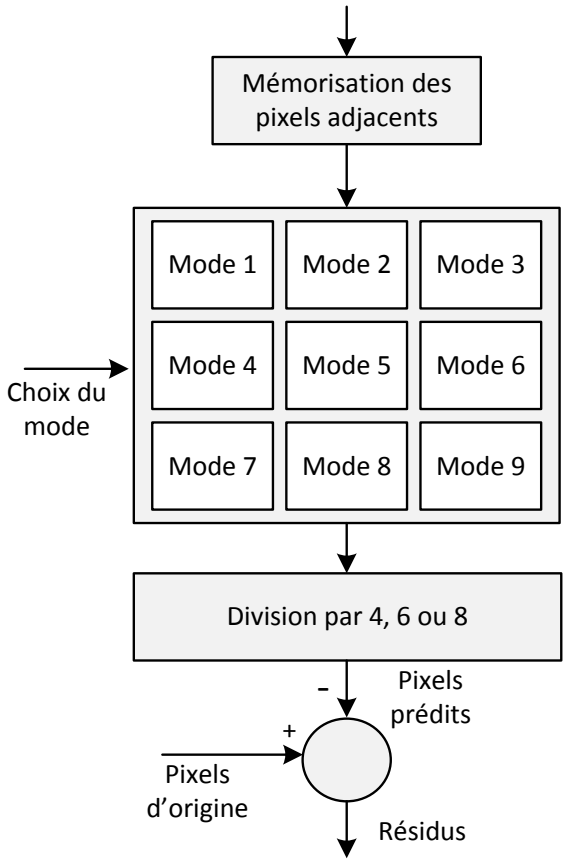


FIGURE 4.9 – Architecture utilisée pour la prédiction intra

Les coefficients sont ensuite transmis aux modules de prédiction. Chaque module possède son propre algorithme en fonction du mode de prédiction qu'il représente (figure 4.10). Le résultat est ensuite divisé par le nombre de coefficients adjacents nécessaire à son calcul (4, 6 ou 8).

La dernière étape consiste à soustraire les coefficients du bloc de données d'origine avec les coefficients du bloc de données prédites, le résultat donne les résidus envoyés sur le reste de la chaîne de codage.

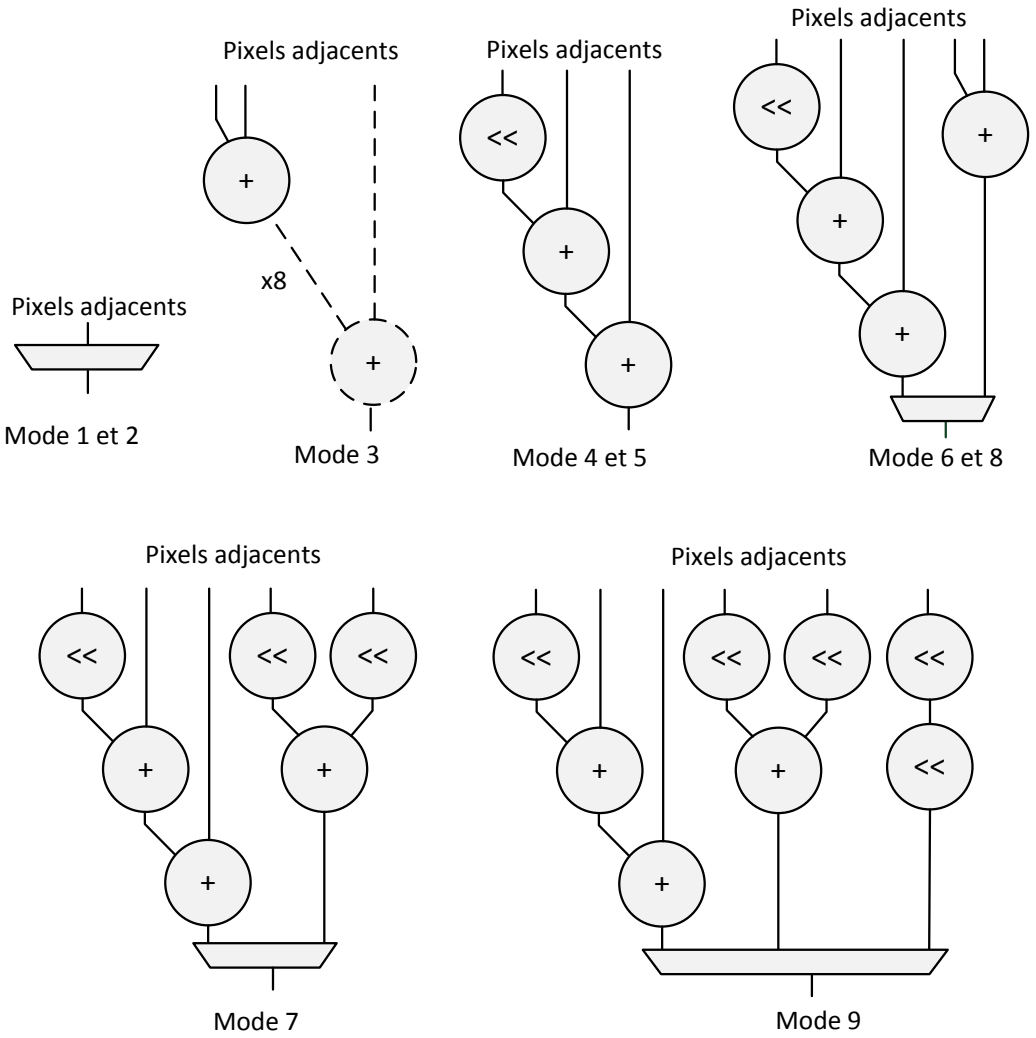


FIGURE 4.10 – Architecture utilisée pour chaque mode

4.2.6 Codage de l'information

Le codage de l'information est l'étape finale avant la transmission du flux de données encodées. L'opération est réalisée par trois parties dépendantes les unes des autres :

– Le codeur entropique CAVLC permet de coder les résidus issus des opérations de prédiction, de transformée et de quantification. CAVLC utilise un algorithme de compression appelé run-level, celui-ci permet de coder les longues suites de '0' logiques. Ce codage est important car suite aux opérations de transformation et de quantification, le flux de données contient énormément de résidus nuls. Pour optimiser la compression des informations, plusieurs tables de codage à longueurs variables codent les séquences binaires

les plus longues et redondantes, par des symboles plus courts.

– Le codeur Exp Golomp assure le codage des entêtes. Celles-ci contiennent les paramètres utilisés par les blocs de traitement, tels que les coefficients de quantification, les modes de prédiction, mais également les informations sur la vidéo, comme la résolution ou le type d'image. Idem à CAVLC, ces informations sont compressées à partir d'une table de codage.
– Le parseur met en forme la NAL (Network Abstraction Layer) grâce aux informations issues des deux opérations précédentes. Chaque NAL est composée d'une entête et d'un certain nombre de données résiduelles.

4.2.7 Boucle de contre réaction

L'encodage d'une vidéo ne se fait pas sans dégradation de la qualité d'image. Pour éviter de voir apparaître un effet de drift lors de la lecture vidéo, il est nécessaire de réaliser les opérations de prédiction dans les mêmes conditions au niveau de l'encodage et du décodage. Pour cela, la prédiction ne doit pas être faite sur l'image d'origine, mais sur une image reconstruite à partir de la transformée inverse et de la quantification inverse.

Cette nécessité implique la création d'une boucle de retour (figure 4.11) constituée des opérateurs de transformée, quantification, quantification inverse et transformée inverse. Chaque bloc 4x4 doit d'abord être traité par ces quatres opérateurs avant de pouvoir être réutilisé dans la prédiction du bloc suivant. Cette pratique a pour conséquence de voir apparaître une latence entre le traitement de deux blocs successifs, et une discontinuité dans le flux de données envoyé au codeur entropique.

Cette latence (figure 4.12) est de 56 cycles d'horloge, et correspond au temps de traitement de la transformée, de la quantification et de leur opération inverse. Sachant qu'un macrobloc se décompose en 16 blocs de luminance et 8 blocs de chrominance, la perte de temps pour le traitement complet d'un macrobloc peut être calculée à 1344 cycles d'horloge.

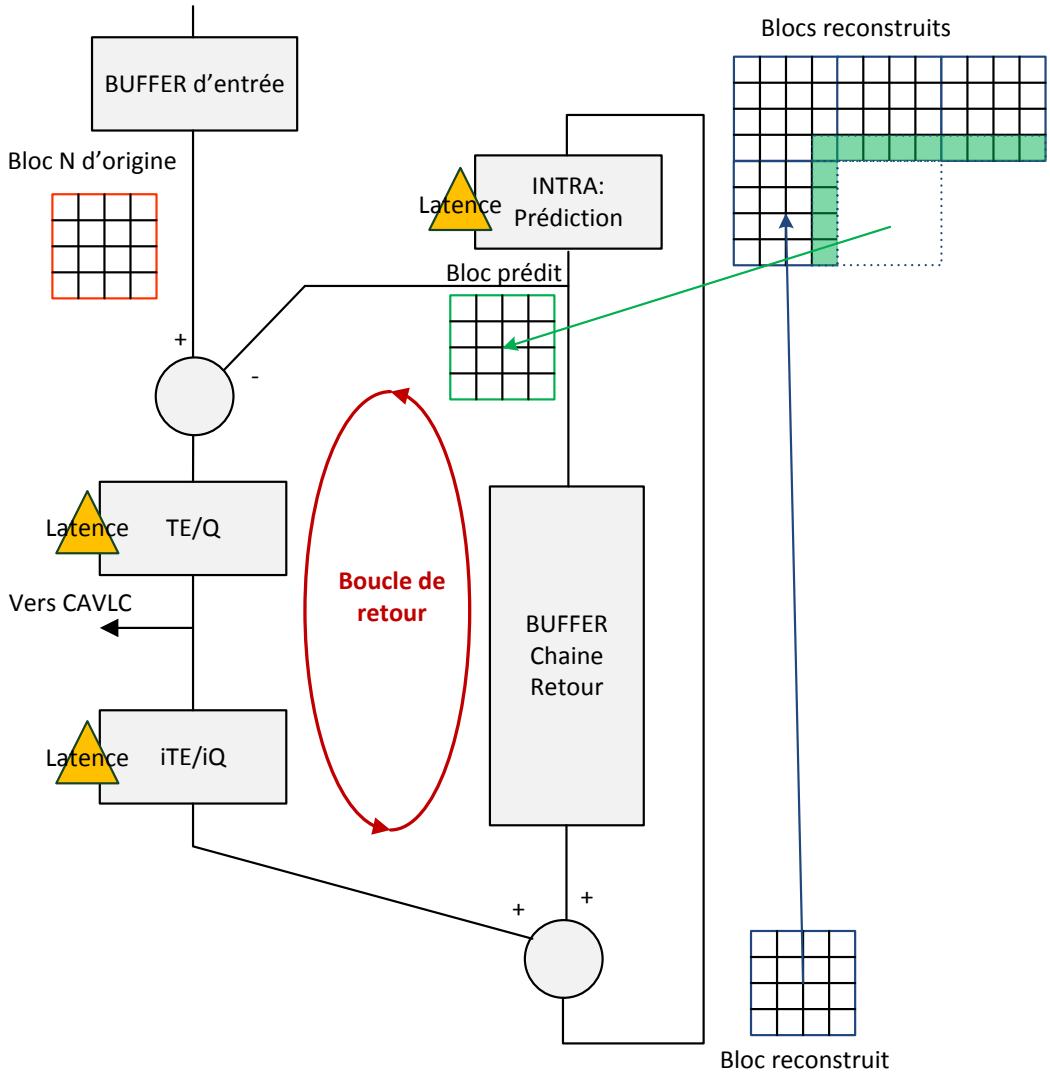


FIGURE 4.11 – Boucle de retour et latences induites

4.2.7.1 Améliorations apportées à la boucle de retour

La latence vient de la dépendance entre les blocs de données 4x4 successifs, et de l'impossibilité de pouvoir commencer le traitement sur un nouveau bloc avant le retour du précédant. Une possibilité afin de réduire cette latence est de désolidariser les blocs entre deux traitements successifs.

Au lieu de traiter tous les blocs de luminance à la suite, la solution consiste à interposer un bloc de chrominance entre deux blocs de luminance afin d'exploiter le temps de traitement inutilisé et disponible (figure 4.13).

Ainsi, les 8 blocs de chrominance peuvent être intercalés entre les 16 blocs de luminance,

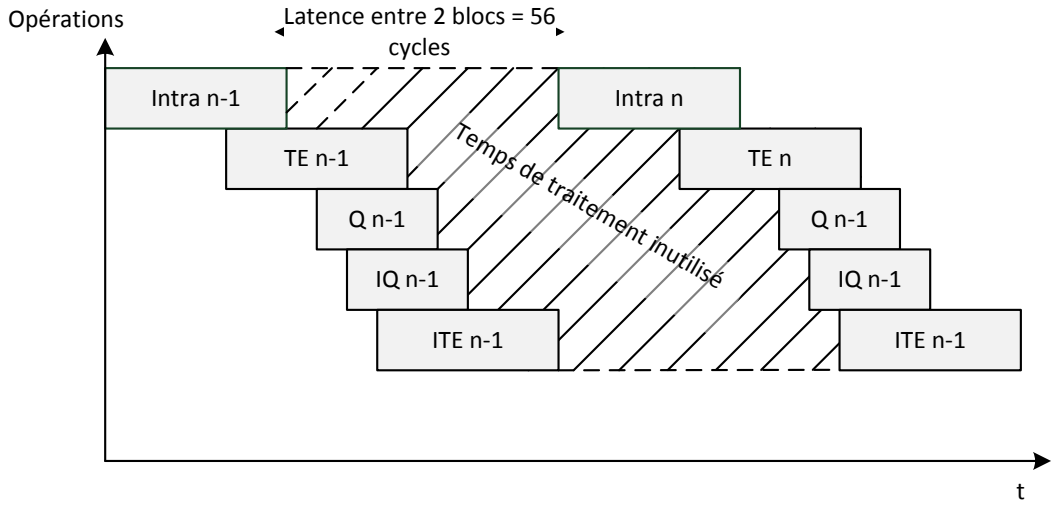


FIGURE 4.12 – Latence induite par la boucle de contre réaction

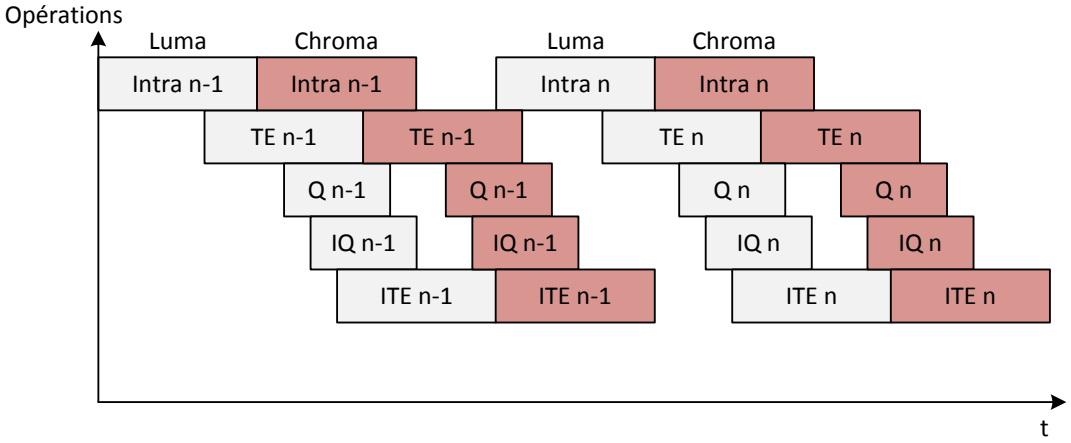


FIGURE 4.13 – Diminution de la latence grâce au traitement alterné de la luminance et de la chrominance

le traitement d'un macrobloc complet par la boucle de retour ne nécessite plus que 896 cycles d'horloge au lieu de 1344, soit une amélioration de 34%.

Au niveau de l'architecture, cette amélioration ne demande rien de plus qu'une FIFO et un multiplexeur en entrée du module de prédiction intra pour la mémorisation des blocs de chrominance pendant le traitement de la luminance.

4.3 Performances

Maintenant que le nombre de cycles d'horloge nécessaires au traitement d'un macrobloc complet est estimé, il nous faut déterminer la fréquence maximale de notre architecture. L'obtention des ces deux éléments servira au calcul du débit maximal théorique atteignable par le transcodeur, et par conséquent, déterminera les formats vidéo pouvant être traités en temps réel.

Ces premiers résultats sont obtenus par simulation, et ne servent qu'à estimer les performances du transcodeur. Plus loin dans ce chapitre, des résultats plus précis seront obtenus par implémentation sur carte.

4.3.1 Performances temporelles

L'estimation de la fréquence maximale est obtenue grâce aux outils Xilinx ISE, par synthèse post placement et routage sur la cible Xilinx Virtex6 XC6VLX540T utilisée dans le projet ARDMAHN. Le tableau 4.3 regroupe les résultats obtenus. Les blocs de traitement, puis l'architecture complète ont été synthétisés indépendamment.

TABLE 4.3 – Fréquences maximales obtenues pour chaque bloc de traitement

	Fréquences max
Transformée	182 Mhz
Transformée inverse	204 Mhz
Quantification	230 Mhz
Quantification inverse	464 Mhz
Préduction intra	191 Mhz
CAVLC	163 Mhz
Exp Golomb	289 Mhz
Parseur	234 Mhz
Architecture complète	**132 Mhz**

Cette première estimation montre que la fréquence maximale de travail ne pourra pas dépasser 132 Mhz. nous avions déterminé que 896 cycles d'horloge étaient nécessaires pour le traitement d'un macrobloc complet. Nous pouvons maintenant convertir cette valeur pour obtenir le temps d'exécution **Te** d'un macrobloc, et calculer le débit **De** maximal capable d'être

reçu et traité par l'encodeur :

$$Te_{min} = Cycles_{MB} \times \frac{1}{F_{max}} = 896 \times \frac{1}{132 \times 10^6} = 6.79\mu s \tag{4.4}$$

$$De_{max} = \frac{1}{Te_{min}} = \frac{1}{6.79 \times 10^{-6}} = 147275 Macroblocs/sec \tag{4.5}$$

L'architecture pourra en théorie, recevoir et traiter en temps réel 147 275 Macroblocs/sec. A titre de comparaison, nous retrouvons dans le tableau 4.4, les débits nécessaires pour certains formats vidéos.

TABLE 4.4 – Exemple de débits nécessaires pour certaines résolutions

Résolution	Images/sec	Débits (Macroblocs/sec)
720x480	30	40 500
720x576	25	40 500
1280x720	25	90 000
	30	108 000
1920x1080	25	202 500
	50	243 000
	60	486 000

La détermination du débit de sortie est plus complexe, étant donné que le codeur entropique (CAVLC) délivre des informations de tailles variables. Mais grâce au débit d'entrée, il est donc possible d'obtenir une approximation du débit de sortie à partir du taux de compression de l'encodeur. Nous reviendrons sur ces deux paramètres plus loin dans ce chapitre, lors des tests d'implémentation.

4.3.2 Performances en surface

Comme pour les performances temporelles, la surface occupée par l'encodeur est obtenue par synthèse, et est détaillée dans le tableau 4.5.

L'architecture complète ne nécessite que 17% des LUT disponibles sur la cible XC6VLX540T. Pour ces tests, les options du synthétiseur ont été choisies de sorte à privilégier la fréquence de l'architecture, plutôt que ses ressources de placement. Là encore, ces résultats ne sont qu'une

TABLE 4.5 – Ressources utilisées pour l'implémentation de chaque bloc de traitement

Fonctions	Slices Registers	Slices LUT	DSP48
Transformée	536	1630	0
Quantification	56	113	1
Transformée inverse	526	1192	0
Quantification inverse	31	71	1
Prédiction intra	763	12454	0
CAVLC	203	1898	0
Exp Golomb	109	376	0
Parseur	146	1870	0
Architecture complète	9060	25709	2
Ressources disponibles sur le FPGA XC6VLX540T	301440	150720	768

estimation, et ne prennent pas en compte les contraintes de surface liées à la création des zones reconfigurables. Ces résultats seront affinés lors de l'implémentation sur carte.

4.4 Validation des paramètres par implémentation

L'implémentation sur carte est réalisée en trois étapes :

- Une première architecture, dite "Standalone", est développée afin de caractériser les IP indépendamment de la chaîne de transcodage.
- La seconde étape consiste à implémenter le transcodeur H.264/Adaptation/MPEG-2, et de vérifier que l'interconnexion entre ces blocs se fait correctement. Les codes sources MPEG-2 utilisés sont fournis par la société VIOTECH, partenaire dans le projet ARDMAHN.
- Enfin, la dernière étape met en place l'utilisation de la reconfiguration dynamique dans la chaîne de transcodage.

A partir de ces trois architectures, les données utiles à l'alimentation de notre modèle seront récupérées. Ces informations concernent en particulier : les temps d'exécution, de reconfiguration, ou encore les débits en entrée et en sortie de la chaîne. La suite de ce paragraphe est construit autour de ces implémentations, et des résultats obtenus.

4.4.1 Architecture standalone

L'architecture standalone (figure 4.14) est la base de toutes nos implémentations. Le but étant de tester et caractériser les IP, nous occulterons les problématiques liées à la transmission des flux en temps réel. Des données sont placées dans la mémoire RAM de la carte de développement, et sont en permanence disponibles pour nos blocs, ce qui permet d'éviter toute rupture dans le flux entrant. De la même façon, les données en sortie sont stockées dans la RAM, et sont récupérées manuellement après un test.

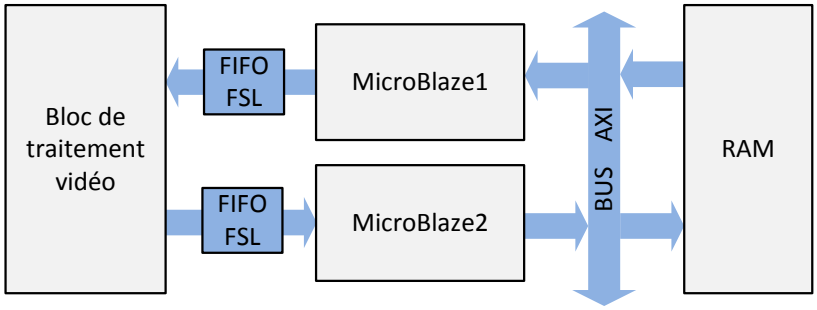


FIGURE 4.14 – Architecture standalone

La communication entre la mémoire et le bloc de traitement se fait via un bus de type AXI (Advanced eXtensible Interface) et une liaison FSL (Fast Simplex Link), contrôlés par des processeurs Microblaze. Le protocole FSL communique grâce à des instructions bloquantes, deux Microblaze sont alors nécessaires afin de désolidariser toute la gestion du flux entrant et sortant, et de ne pas impacter le fonctionnement des IP et leurs débits respectifs.

4.4.1.1 Caractérisation des IP de transcodage (Hors décodeur H.264)

– **Détermination du débit d'entrée :**

Afin de déterminer le débit d'entrée maximal de l'encodeur en pratique, une fonction logicielle de mesure a été développée et intégrée au Microblaze 1. Une mémoire de type FIFO est placée entre l'encodeur et le processeur. Pour valider notre mesure, cette FIFO ne doit jamais se vider, sous peine d'interrompre l'exécution de l'encodeur (par manque de données), et de fausser la mesure de débit. Si ce cas venait à arriver, un signal serait directement transmis au Microblaze pour lui signaler que la mesure est erronée et non exploitable.

Pour notre test, nous stockons une vidéo de quelques images brutes dans la RAM, et nous appliquons une fréquence de 125Mhz à l'encodeur. Nous relevons un débit de 525 coefficients envoyés à l'encodeur toutes les 10μs. Soit 52 500 000 coefficients/sec, ou encore 136 718 Macroblocs/sec. Soit quasiment le même résultat que notre estimation théorique, avec une fréquence de travail a 132Mhz (147 275 Macroblocs/sec).

– **Détermination du débit de sortie :**

Comme nous l'expliquions plus haut dans ce chapitre, le débit de sortie est de taille variable (CAVLC), à cause de cela, il n'est pas possible de définir un débit fixe, seule une moyenne peut être déterminée. Nous avons réalisé des tests sur une dizaine de flux vidéo, dans les mêmes conditions que pour la détermination du débit d'entrée. Une moyenne est alors réalisée grâce aux débits annoncés par la fonction de mesure.

Sur les dix flux encodés, les débits varient autour de 13000 Macroblocs/s à la fréquence de 125Mhz. La moyenne calculée sur l'ensemble des échantillons est de 13672 Macroblocs/s.

Une seconde méthode consiste à déterminer le taux de compression moyen de l'encodeur grâce aux tailles de fichiers de séquences brutes et encodées. Nous connaissons le débit maximal en entrée, nous pouvons alors calculer le débit moyen de sortie. Pour un coefficient de quantification QP égal à 0, le taux de compression moyen est de 90%, appliqué aux 136718 Macroblocs/sec d'entrée, nous retrouvons un résultat quasi similaire au précédent.

TABLE 4.6 – Débits en entrée et sortie des blocs de traitement pour une fréquence d'horloge à 125Mhz

	Taux de (%) compression (%)	Débits compressés	Débits décompressés
Blocs MPEG-2	85,5%	18,1 Mo/s	125 Mo/s
Blocs H.264	90%	5,2 Mo/s	53 Mo/s

De la même façon, nous avons réalisé des tests de débits sur les blocs de traitement MPEG-2. Les résultats temporels obtenus sont présentés dans le tableau 4.6.

4.4.1.2 Caractérisation du décodeur H.264

A l'heure actuelle, le décodeur H.264 n'étant pas finalisé (le décodeur entropique CAVLD n'est pas encore développé), il nous est impossible de réaliser les tests d'implémentation complets. Toutefois, ses blocs et algorithmes étant en grande partie identiques à ceux utilisés dans la partie encodage, il est assez facile d'extrapoler ses caractéristiques, pour obtenir une estimation assez proche de la réalité. La procédure consiste à implémenter le transcodeur H.264/H.264 sans les blocs de codage entropique manquants (figure 4.15), les données sont alors contournées et transmises directement de la quantification de l'encodeur, à la quantification inverse du décodeur. Ce test permet de déterminer le débit de sortie du décodeur, est par application du taux de compression de CAVLC, de calculer son débit d'entrée.

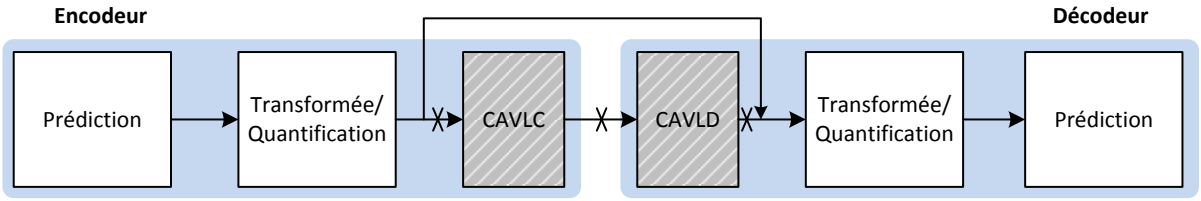


FIGURE 4.15 – Bypass de CAVLC pour la caractérisation du décodeur

Contrairement à l'encodeur, le décodeur ne possède pas de boucle de retour, les performances de ce dernier s'en retrouvent grandement améliorées. Les débits en entrée et en sortie sont de l'ordre de 125 Mo/s et 12,5 Mo/s.

Pour le calcul de la surface nécessaire au décodeur H.264, nous avons considéré que CAVLD était équivalent à CAVLC.

4.4.1.3 Caractérisation du bloc d'adaptation

Comme nous l'avons dit, il a été décidé de garder le bloc d'adaptation statique, la surface qu'il occupe n'a donc aucun impact sur la détermination des tailles des zones reconfigurables. Concernant son temps d'exécution et sa latence, ils varient en fonction du ratio de l'adaptation souhaité. Pour une réduction d'image par exemple, le traitement consiste à fusionner les macroblocs par groupe de quatre (diminution par 2) ou seize (diminution par 4) comme illustré sur la figure 4.16.

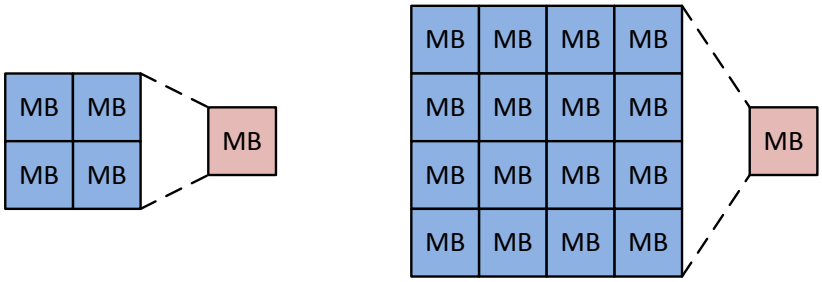


FIGURE 4.16 – Réduction d'image par deux et par quatre

Pour une division par 1 (ratio 1 :1), la résolution n'est pas modifiée, dès qu'un macrobloc est disponible, il est transmis en sortie, l'adaptation se comporte alors comme un buffer qui génère une latence de 384 cycles d'horloges équivalent aux 384 pixels du macrobloc.

La division par 2 (ratio 1 :2) nécessite l'utilisation de quatre macroblocs (deux sur la première ligne, et deux sur la seconde). Le traitement de l'image se faisant par ligne, il faut alors traiter toute la première ligne avant d'avoir accès aux macroblocs de la seconde ligne. La latence pour cette opération dépend donc de la taille de l'image brute, pour une image full HD de résolution 1920x1080 par exemple, une ligne est composée de 120 macroblocs, la latence sera de 46848 cycles (384x122), soit 0.47ms à la fréquence de 100Mhz.

De la même manière, l'adaptation aura une latence de quatre lignes pour une division par 4 (ratio 1 :4), soit 139776 cycles (384x364) ou 1.4ms, toujours pour une résolution de 1920x1080.

TABLE 4.7 – Latence et temps de traitement du bloc d'adaptation pour différentes résolutions

Résolutions	Type de réduction	Latence (ms)	Temps d'exécution pour une image (ms)	Temps de traitement total pour une image (ms)
1920x1080	2	0,47	15,4	15.9
	4	1.4	7.7	9.1
1280x720	2	0.31	6.9	7.2
	4	0.94	3.4	4.3
720x576	2	0.17	3.1	3.3
	4	0.53	1.5	2

Le tableau 4.7 résume les temps et latences du bloc d'adaptation, pour quelques résolutions. Les temps d'exécution correspondent aux temps nécessaires pour traiter une image complète,

en considérant que chaque pixel est traité en un cycle d'horloge à 100Mhz, nous pouvons alors déterminer le temps de traitement d'un pixel, qui est 1/100Mhz = 10ns. Enfin, le temps total nécessaire au traitement complet d'une image est obtenu en additionnant la latence et le temps d'exécution d'une image.

4.4.2 Transcodeur MPEG-2/H.264

Cette seconde architecture est utilisée pour valider le transcodage vidéo entre les codecs MPEG-2 et H.264 (figure 4.17). Comme précédemment, les flux d'entrée et de sortie sont stockés et récupérés manuellement dans la mémoire RAM externe.

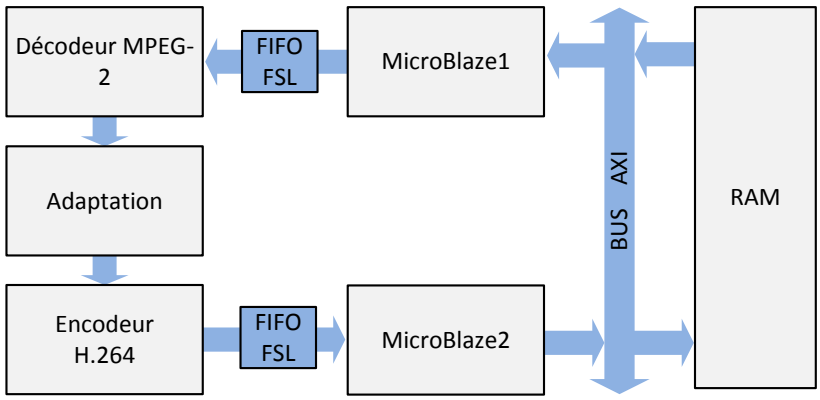


FIGURE 4.17 – Architecture du transcodeur MPEG-2/H.264

Des wrappers communs pour les standards MPEG-2 et H.264 ont été développés, la communication entre les IP se fait en toute transparence quel que soit le codec utilisé. Des mémoires tampons ont été placées entre chaque bloc, de sorte à fluidifier les flux de données. Un indicateur de FIFO vide permet la mise en attente des IP en aval.

La figure 4.18 montre une grosse disparité lors du placement des blocs de transcodage, à première vue, notre architecture utilise quasiment tout l'espace disponible du FPGA, mais comme le montre le tableau 4.8, seulement 5% et 20% des slices registers et LUT disponibles sont utilisées.

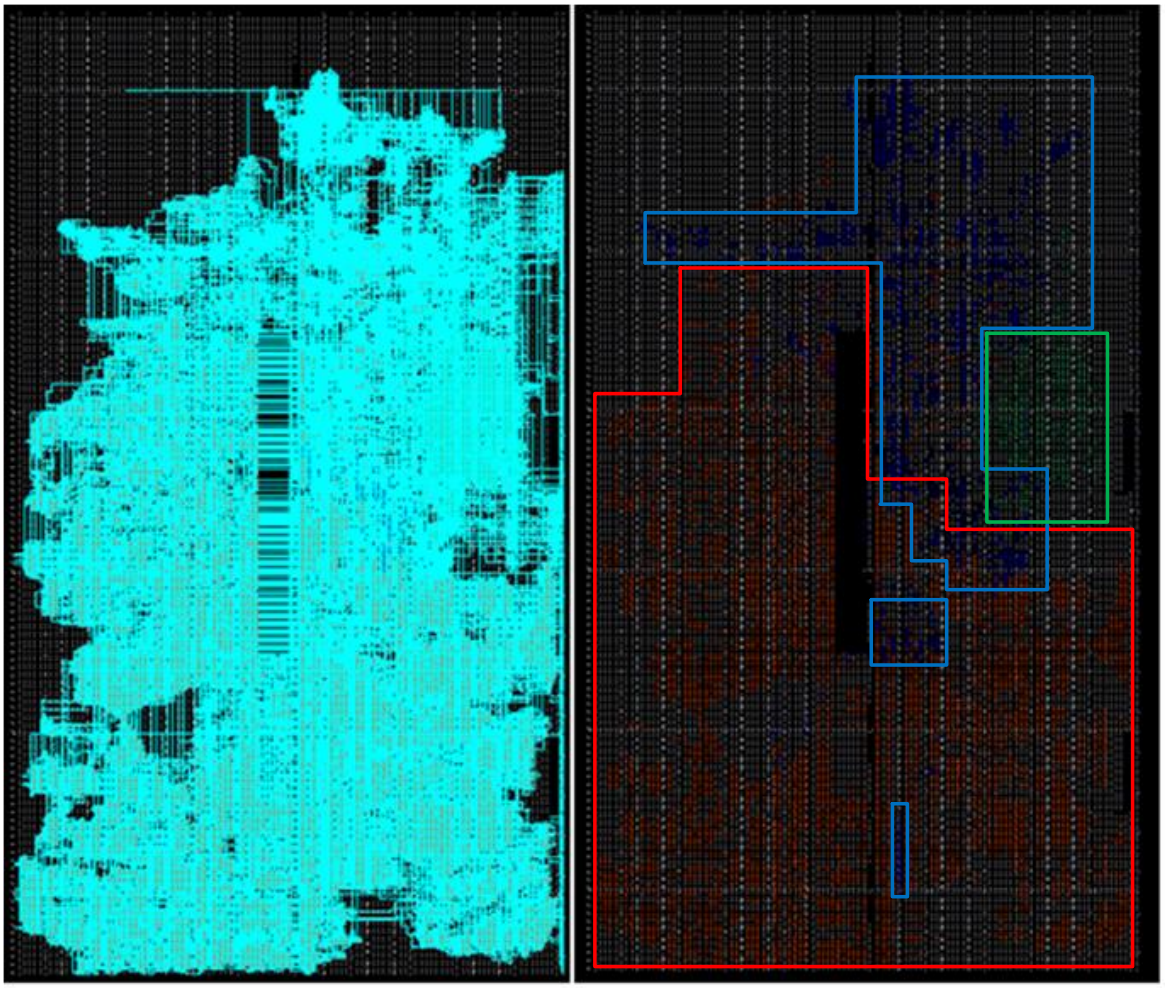


FIGURE 4.18 – Occupation du FPGA (Vert : Microblazes, rouge : transcodeur, bleu : autres)

TABLE 4.8 – Ressources nécessaires à l'implémentation du transcodeur

	Utiles	Disponibles	Utilisation
Slices registers	17 494	301 440	5%
Slices LUT	30 872	150 720	20%
DSP48	18	768	2%

4.4.3 Utilisation de la reconfiguration dynamique

Cette dernière architecture (figure 4.19) est quasi identique à la précédente, mis à part, le module Xilinx HWICAP (HardWare Internal Configuration Access Port) utile à la gestion des bitstreams de reconfiguration.

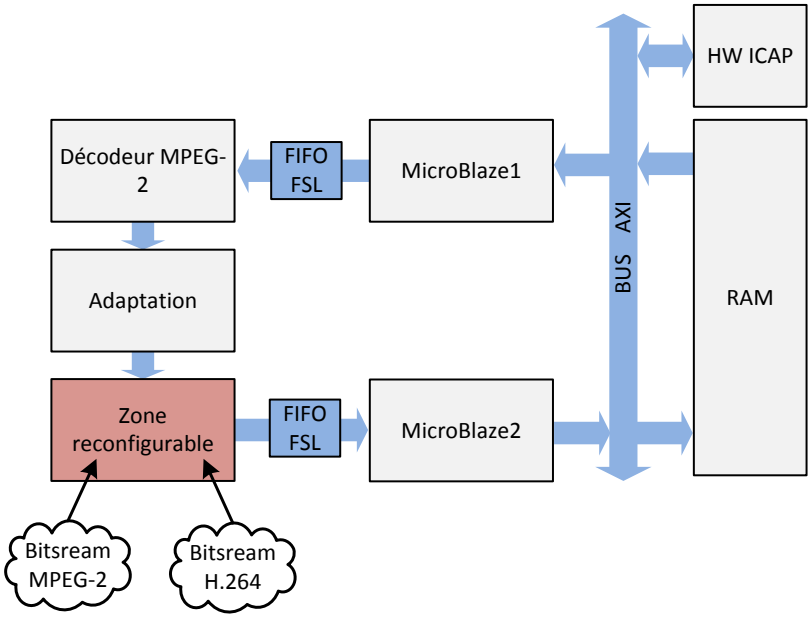


FIGURE 4.19 – Architecture du transcodeur reconfigurable

L'absence des IP de décodage H.264 nous oblige à ne réaliser la reconfiguration que sur la partie d'encodage, le reste du design reste statique. Les bitstreams de configuration pour l'encodeur H.264 et MPEG-2 sont générés avec l'outil Xilinx PlanAhead. Nous connaissons la surface occupée par chaque IP (tableau 4.9), nous pouvons alors délimiter manuellement une zone reconfigurable de taille équivalente à la plus grosse tâche, ici l'encodeur H.264 (figure 4.20 et 4.21).

TABLE 4.9 – Ressources nécessaires à l'implémentation du transcodeur reconfigurable

Ressources	Zone reconfigurable	H.264	Utilisation	MPEG-2	Utilisation
Slices Registers	51 200	5 006	10%	5 147	11%
Slices LUT	25 600	19 850	78%	5 314	21%
DSP48	128	2	2%	16	13%

Afin d'éviter tout problème de placement et de routage, nous augmentons la taille de la zone reconfigurable de 10% à 20%. Comme on peut le remarquer, l'encodeur H.264 utilise beaucoup de slices LUT, c'est donc cette valeur qui est prise en compte pour la création de la zone reconfigurable, laissant alors une grande part de slices registers et de DSP48 non utilisés.

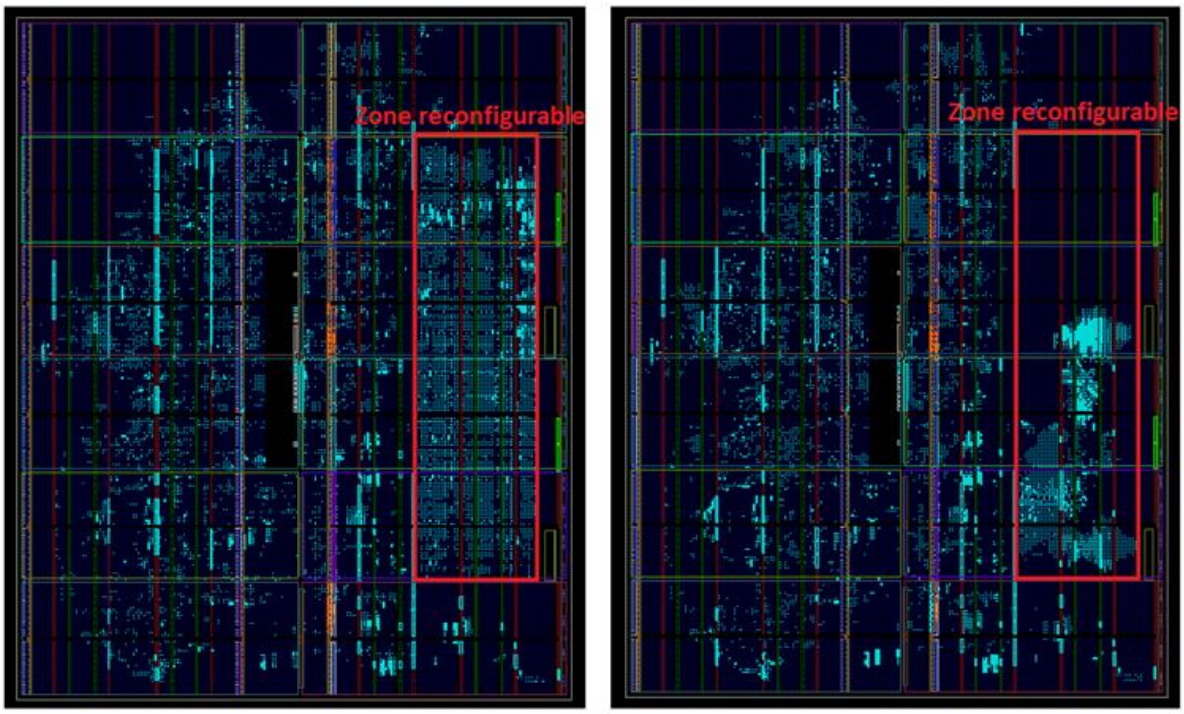


FIGURE 4.20 – Implémentation de l'architecture reconfigurable. A gauche H.264, à droite MPEG-2 sont implémentés dans la zone reconfigurable

Encore une fois, les tests effectués ici n'ont pas pour but de valider l'auto-adaptabilité du transcodeur, mais seulement les interconnexions et les communications entre les IP, et à déterminer les informations nécessaires à l'alimentation de notre modèle (telle que la taille des zones reconfigurables).

Pour notre test, nous stockons un flux vidéo MPEG-2 dans la RAM, et nous reconfigurons la zone manuellement (grâce à l'outil Xilinx Impact) pour basculer d'un transcodage MPEG-2/Adaptation/MPEG-2, à un transcodage MPEG-2/Adaptation/H.264, et vice versa. Les flux vidéo MPEG-2 et H.264 issus des deux transcodages sont ensuite récupérés dans deux emplacements différents de la RAM, puis décodés et visualisés grâce aux décodeurs de références.

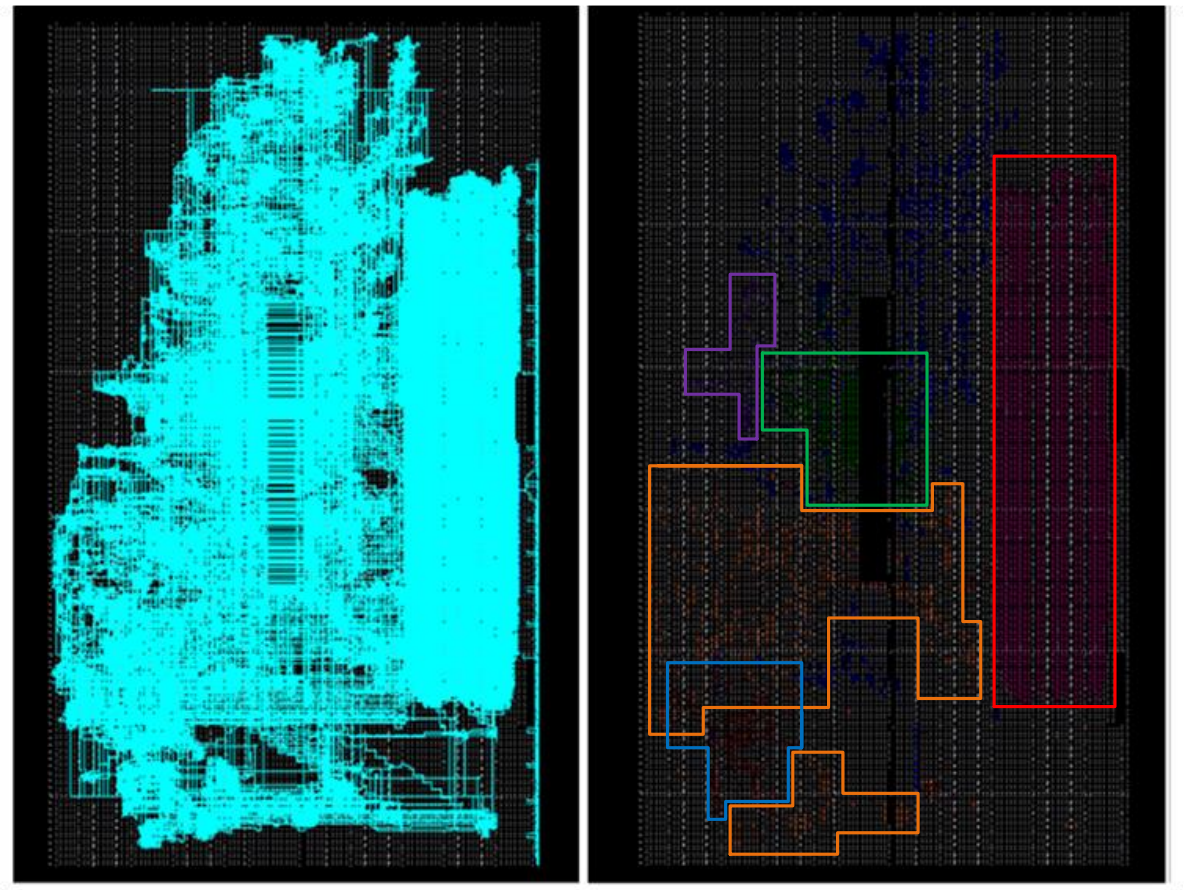


FIGURE 4.21 – Occupation du FPGA (Vert : Microblazes, violet : ICAP, orange : décodeur, bleu : adaptation, rouge : zone reconfigurable)

4.5 Conclusion

Nous avons vu dans ce chapitre les différentes étapes de développement des IP H.264 et leurs tests. L'implémentation d'une chaîne de transcodage complète (décodage H.264, adaptation, encodage MPEG-2) sur carte a permis de nous assurer de l'homogénéité des données entre les blocs de traitement de types différents. Les résultats obtenus en simulation et en implémentation vont nous permettre d'alimenter correctement notre modèle.

Dans le chapitre suivant, nous allons réutiliser ces informations afin de simuler les quatre cas d'utilisation définis précédemment.

Chapitre 5

Simulation et raffinement du modèle

Sommaire

5.1 Introduction 98

5.2 Synthèse des paramètres de l'application 98

5.2.1 L'encodage et le décodage 98

5.2.2 Les zones reconfigurables 98

5.3 Résultats de simulation 99

5.3.1 Cas d'utilisation mono-flux 100

5.3.2 Les cas d'utilisation multi-flux en entrée 104

5.3.3 Les cas d'utilisation multi-flux en sortie 108

5.3.4 Les cas d'utilisation multi-flux en entrée et sortie 111

5.4 Synthèse des résultats de simulation 115

5.5 Discussion sur la validité des résultats 116

5.6 Conclusion 118

5.1 Introduction

Dans les chapitres précédents, nous avons proposé une méthodologie de modélisation et de simulation d'architectures reconfigurables. Nous avons caractérisé des IP de transcodage vidéo et déterminé leurs attributs. Dans ce chapitre, nous allons utiliser les informations recueillies pour alimenter notre modèle, et simuler les cas d'utilisation définis dans le cadre du projet ARDMAHN. Grâce aux résultats obtenus, nous saurons si les IP utilisées sont adaptées aux contraintes du projet. Dans le cas contraire, une estimation des performances attendues sera donnée.

Ce chapitre se décompose en deux parties : tout d'abord, nous ferons une synthèse des informations collectées dans les chapitres précédents. Puis nous détaillerons et commenterons les résultats de simulation obtenus.

5.2 Synthèse des paramètres de l'application

Dans ce premier paragraphe, nous allons récapituler les éléments qui caractérisent la chaîne de transcodage utilisée dans le projet ARDMAHN.

5.2.1 L'encodage et le décodage

Le tableau 5.1 synthétise les temps, les débits et les surfaces occupés par les IP H.264 et MPEG-2. Comme nous l'avons expliqué précédemment, les caractéristiques du décodeur H.264 peuvent être extrapolés grâce aux données obtenues pour l'encodeur.

Les temps d'exécution correspondent aux temps moyens nécessaires pour traiter un pixel. Il s'agit en fait du chemin critique de chaque tâche.

5.2.2 Les zones reconfigurables

De la même manière que pour les tâches, les caractéristiques des zones reconfigurables sont issues des implémentations réalisées sur FPGA Xilinx Virtex 6. L'architecture interne de ce FPGA est découpée en frames, qui correspondent aux plus petits éléments reconfigurables.

TABLE 5.1 – Informations connues sur les différentes tâches du transcodeur pour une fréquence de travail de 125MHz

Tâches	Temps d'exécution	Débit max d'entrée	Débit max de sortie	Slice LUTs	Slice Registers	DSP48
Décodeur MPEG-2	8 ns	18,1 Mo/s	125 Mo/s	3710	3190	10
Décodeur H.264	8 ns	12,5 Mo/s	125 Mo/s	23970	8470	1
Encodeur MPEG-2	8 ns	125 Mo/s	18,1 Mo/s	5370	4680	16
Encodeur H.264	19 ns	53 Mo/s	5,2 Mo/s	25710	9060	2

Trois types de frames sont disponibles : les CLB, les BRAM, et les DSP. La composition et la taille de chacune d'entre elles peuvent être connues grâce à l'outil Xilinx Planhead (tableau 5.2).

TABLE 5.2 – Calcul de la taille de chaque frame

Ressources	Composition				Taille (octets)
Frame CLB	320 LUT	640 Register	40 slice L	40 Slice M	11664
Frame BRAM	8 BRAM	8 Fifo36	X	X	50544
Frame DSP	16 DSP48	X	X	X	9072

Précédemment (tableau 4.9), nous avons déterminé la surface et la composition des zones reconfigurables, nécessaires à l'implémentation des IP de transcodage. Grâce à ces données, nous pouvons déduire la composition en frames des zones reconfigurables : 80 frames CLB, 1 frame BRAM et 1 frame DSP, et la taille des bitstreams de configuration : 933ko (80x11664 octets) pour les CLB, 50ko (1x50544 octets) pour les BRAM et 9ko (1x9072 octets) pour les DSP, soit un bitstream total de 992ko.

Sachant que l'ICAP (32bits) a un taux de transfert théorique de 400Mo/s à la fréquence de 100Mhz [LKLJ09], le téléchargement d'une configuration de 992ko prendra 2,5ms.

5.3 Résultats de simulation

Les résultats de simulation du canal de transcodage sont regroupés en quatre parties, chacune correspond à un des quatre types de cas d'utilisation définis dans le chapitre 3 (figures 3.12, 3.13, 3.14 et 3.15). Pour chaque cas, les simulations sont réalisées avec les paramètres les plus

contraignants (qui peut le plus, peut le moins), nous utiliserons une résolution de 720x576 pixels pour modéliser un flux SD et 1920x1080 pour un flux HD, et nous fixons un frame rate de 20 images/seconde. Pour chaque résultat, nous analyserons les performances obtenues et proposerons des méthodes afin d'améliorer celles-ci.

5.3.1 Cas d'utilisation mono-flux

5.3.1.1 Modèle de l'application

Le cas d'utilisation mono-flux est le plus simple à mettre en œuvre. Mis à part lors du démarrage de la simulation, aucune reconfiguration dynamique n'est requise. La figure 5.1 présente le graphe de tâche de ce scénario. Un flux HD est traité en temps réel, nécessitant la configuration du décodeur et de l'encodeur H.264.

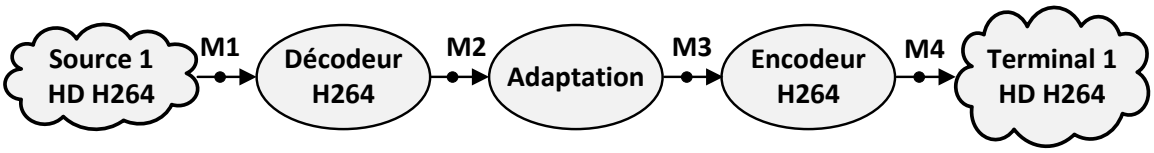


FIGURE 5.1 – Modèle de l'application mono-flux

Les indices M1 à M4 correspondent aux points de mémorisation du flux vidéo.

5.3.1.2 Résultats

Les chronogrammes 5.2 illustrent l'évolution des mémoires au cours de la simulation. Comme nous l'avons vu dans le chapitre précédant, les performances de l'encodeur H.264 ne permettent pas de traiter un flux HD 1920x1080 en temps réel. La chaîne de transcodage reçoit ce flux, mais n'est pas capable de le traiter assez rapidement. Il en résulte un goulot d'étranglement en entrée du décodeur, la mémoire M3 n'arrive pas à se vider aussi vite qu'elle se remplit, le flux vidéo transmis au terminal de lecture est alors cadencé par l'encodeur.

Les résultats de simulations nous informent que le débit de sortie du flux vidéo vers le terminal de lecture est de 5,2Mo/s alors que celui-ci devrait être de 6,2Mo/s pour respecter l'émis-

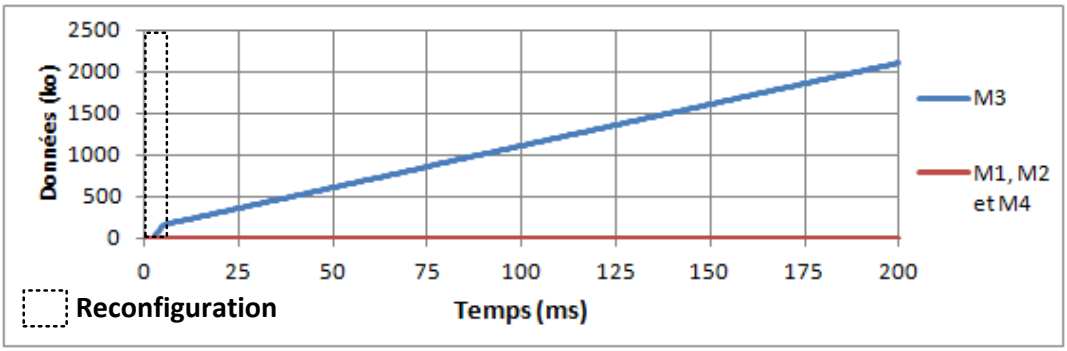


FIGURE 5.2 – Evolution des mémoires intermédiaires pour le cas mono-flux avec les paramètres actuels

sion temps réel d'une vidéo de cette résolution. Cette différence de débits risque d'engendrer de fortes perturbations sur la qualité visuelle pour l'utilisateur.

5.3.1.3 Modifications possibles

Cette première simulation nous a permis d'identifier l'IP dont les performances empêchent la réalisation de ce scénario. A partir de l'analyse de ces résultats, plusieurs modifications sont permises :

Nous savons que le débit de sortie du transcodeur est lié à la résolution d'image et au frame rate, nous pouvons alors jouer sur un de ces deux paramètres pour le contrôler. Ou inversement, déduire les limites en résolution et frame rate à ne pas dépasser avec l'architecture actuelle :

– Un flux 1920x1080 à 17 images/s correspond à un débit de 5,2Mo/s.

– Un flux 1440x1050 à 20 images/s correspond à un débit de 4,5Mo/s.

Au dessus de ces deux combinaisons de paramètres, nous savons que le transcodeur ne sera pas capable de traiter la vidéo en temps réel.

Une seconde modification consiste à améliorer les performances des blocs de traitement H.264. Bien que ce travail ne soit pas du ressort de notre simulateur, il est tout de même capable de déterminer les objectifs à atteindre.

Pour cela, nous réalisons de nouvelles simulations en retirant les contraintes temporelles que nous avions attribuées à l'encodeur H.264. A la place, nous lui indiquons une plage de valeurs

à explorer.

Plusieurs simulations sont réalisées avec des temps d'exécution décroissant, jusqu'à permettre le respect du débit de 6,2Mo/s en sortie. La plage de temps, et le pas d'exploration sont précisés par l'utilisateur, bien sûr, ces paramètres influent directement sur le nombre et le temps de simulation. Pour gagner du temps, mieux vaut commencer avec un pas élevé, puis le réduire une fois une première estimation déterminée.

Les chronogrammes 5.3 correspondent aux résultats obtenus pour un temps d'exécution de 16,1ns au lieu des 19ns antérieurs. Le débit de sortie de 6,2Mo/s est bien respecté, ainsi que la résolution (1920x1080) et le frame rate (20 images/s) du flux vidéo. Cette solution répond aux attentes et ne nécessite que 124ko de capacité mémoire au point M3.

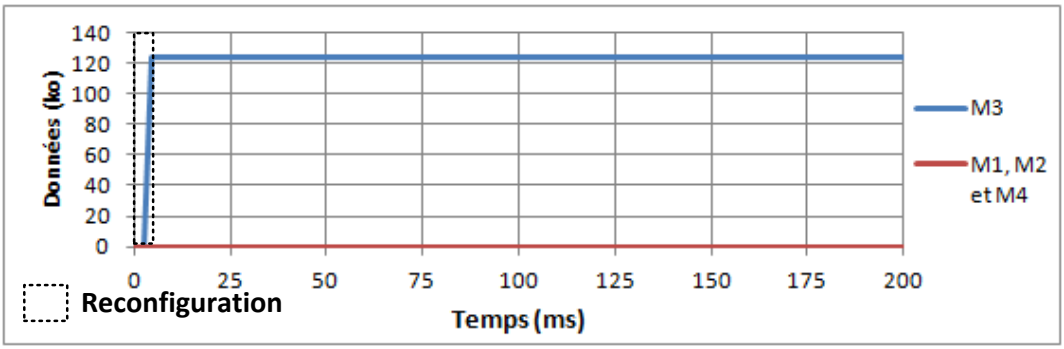


FIGURE 5.3 – Evolution des mémoires intermédiaires pour le cas mono-flux avec modification des paramètres des IP

Un résumé des différentes combinaisons et performances obtenues est présenté dans le tableau 5.3, les cases en gras correspondent aux paramètres modifiés.

TABLE 5.3 – Performances maximales pouvant être atteintes pour le cas mono-flux

Résolution max	Frame rate max (images/sec)	Texec Enc. H.264 (ns)	Mémoires nécessaires (ko)	Taux d'occupation de la zone reconfigurable (% du temps total)
1920x1080	**16**	19	124	≈ 0% en reconfiguration ;
1440x1050	20	19	124	≈ 100% en exécution H.264 ;
1920x1080	20	**16.1**	124	et 0% en exécution MPEG-2

– **Exploitation des résultats :**

Grâce à l'étude réalisée au préalable sur les IP H.264, et à l'exploitation des résultats de simulation, nous pouvons définir précisément les performances attendues par l'encodeur H.264. Nous avons vu dans le chapitre précédant que ses performances temporelles dépendaient essentiellement de la boucle de retour réalisée par la quantification, la transformée, et leur inverse (figure 5.4). En reprenant l'équation 4.5, nous pouvons déterminer l'étendue des modifications à réaliser sur celle-ci, pour atteindre le temps d'exécution de 16,1ns :

$$Te_{pixel} = Cycles_{MB} \times \frac{1}{F_{max}} \times \frac{1}{Pixels_{MB}} \quad (5.1)$$

$$Cycles_{MB} = Te_{pixel} \times F_{max} \times Pixels_{MB} \quad (5.2)$$

$$Cycles_{MB} = 16,1ns \times 125Mhz \times 384 = 773cycles/MB \quad (5.3)$$

Où ***CyclesMB*** est le nombre de cycles nécessaires à la boucle de retour pour traiter un MB,
Tepixel le temps d'exécution pour un pixel,
et ***PixelsMB*** le nombre de pixels dans un MB.

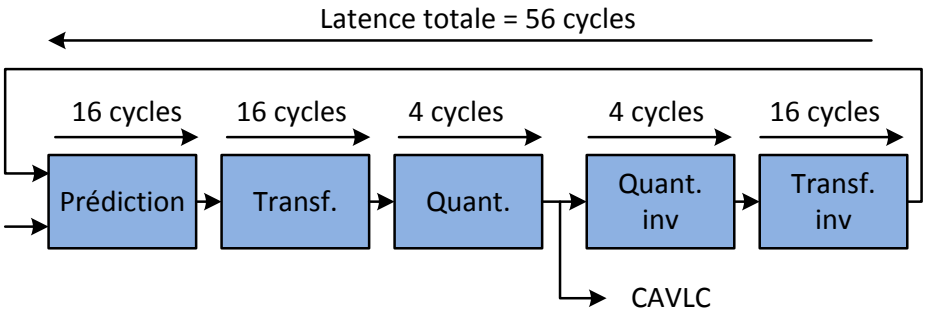


FIGURE 5.4 – Rappel : boucle de retour H.264 sans modification, et latence induite

La latence de la boucle de retour doit donc passer de 896 cycles par macrobloc (valeur actuelle), à 773 cycles. Soit une latence maximale de 48 cycles entre le traitement de deux blocs 4x4 successifs, au lieu de 56. Nous pouvons encore affiner ce résultat, et déduire les modifications nécessaires au niveau des blocs de traitement : pour atteindre les 48 cycles d'horloge, il suffit par exemple de réduire la latence de la transformée à 12 cycles, au lieu des 16 actuelles (en parallélisant son architecture, et ses opérateurs "butterfly" (figure 4.2) par exemple)

Bien sûr, le fait de retravailler les IP pour les rendre plus performantes risque d'avoir une incidence sur la surface occupée par celles-ci. Une fois ces modifications effectuées, il sera toujours possible d'intégrer ces nouvelles données au modèle pour refaire les simulations.

5.3.2 Les cas d'utilisation multi-flux en entrée

5.3.2.1 Modèle de l'application

Le second cas d'utilisation nécessite quant à lui une reconfiguration. Dans le scénario étudié et présenté sur la figure 5.5, deux sources vidéo HD et SD sont envoyées sur le transcodeur pour être traitées puis transmises sur un même terminal de lecture.

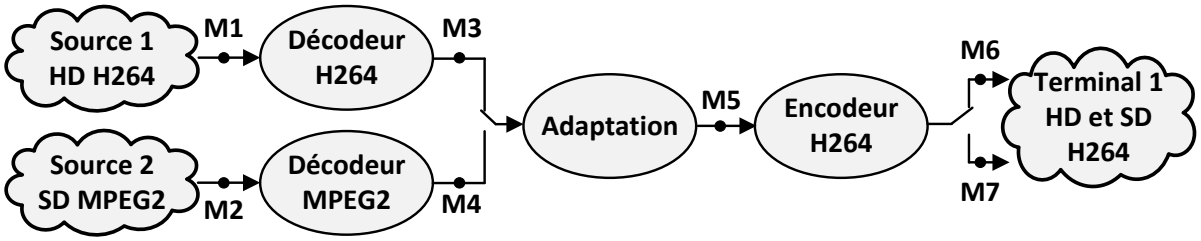


FIGURE 5.5 – Modèle de l'application multi-flux en entrée

Le décodage des sources est réalisé en alternance grâce à la reconfiguration du décodeur H.264 et MPEG-2. Puisque le terminal ne prend en compte que le codec H.264, l'encodage des flux est fait par la même IP. Cette dernière partie ne nécessite pas de reconfiguration, mais un simple multiplexage en entrée.

5.3.2.2 Résultats

Sur les chronogrammes 5.6, M1 et M2 représentent les mémoires de régulation des flux d'entrée de la chaîne : les flux HD et SD arrivent continuellement à son entrée, lorsqu'un des encodeur n'est pas disponible, le flux qui lui est dédié doit être mémorisé jusqu'à sa configuration. La mémoire M2 stock le flux MPEG-2, tandis que M1 stock le flux H.264. M6 et M7 ont un rôle similaire, et régulent les flux de sortie vers le terminal de lecture.

On pouvait s'y attendre, pour les mêmes raisons que précédemment, les blocs de traitement

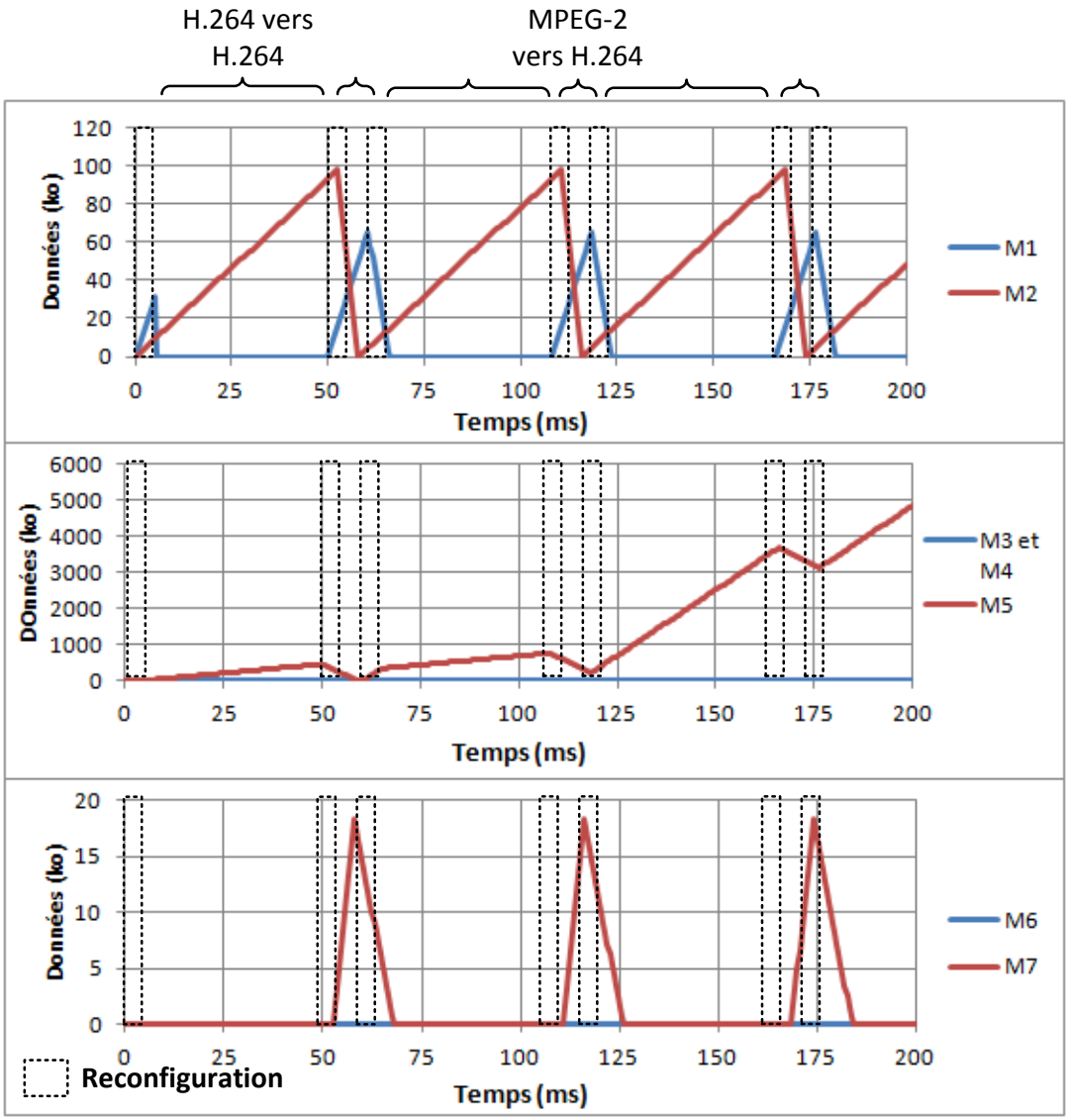


FIGURE 5.6 – Evolution des mémoires intermédiaires pour le cas multi-flux en entrée avec les paramètres actuels

ne sont pas suffisamment performants pour réaliser ce scénario. L'IP d'encodage H.264, trop limitée en débit, entraîne un remplissage croissant de la mémoire M5 à son entrée.

En sortie, les débits moyens sont de 4,2Mo/s au lieu des 6,2Mo/s requis pour le flux H.264, et 700ko/s au lieu de 1860ko/s pour le flux MPEG-2.

5.3.2.3 Modifications possibles

De la même façon que pour le cas d'utilisation précédant, il est possible de jouer sur plusieurs paramètres afin de mener à bien ce scénario. La réduction de la résolution ou du frame rate du flux H.264, ou encore la modification de l'encodeur H.264. Le tableau 5.4 présente les solutions testées par simulation.

TABLE 5.4 – Performances maximales pouvant être atteintes pour le cas multi-flux en entrée

Résolution max	Frame rate max (images/sec)	Texec Enc. H.264 (ns)	Mémoires nécessaires (ko)	Taux d'occupation de la zone reconfigurable (% du temps total)
1920x1080	**13**	19	400	10% en reconfiguration ;
1440x960	20	19	400	80% en exécution H.264 ;
1920x1080	20	**12.9**	400	et 10% en exécution MPEG-2

Les chronogrammes 5.7 correspondent aux simulations effectuées avec l'encodeur H.264 modifié. On peut constater aux points M6 et M7, que les temps d'exécution sont idéalement calculés pour masquer les temps de reconfiguration des décodeurs. Avec cet ordonnancement, les mémoires M6 et M7 ne sont jamais vidées complètement, et les flux transmis au terminal ne sont jamais interrompus.

Les débits moyens mesurés en sorties sont de 6,2Mo/s pour le flux HD et et 1,86Mo/s pour le flux SD, ce qui correspond aux limites de transmission temps réel pour de tels flux. Les mémoires intermédiaires nécessitent un total de 400ko de capacité. Et la zone reconfigurable est occupée à 80% du temps par les blocs H.264, ce qui parait logique étant donné la différence de résolution entre les flux HD et SD.

– **Exploitation des résultats :**

Ici aussi, le manque de performance provient de la boucle de retour H.264, et des blocs de

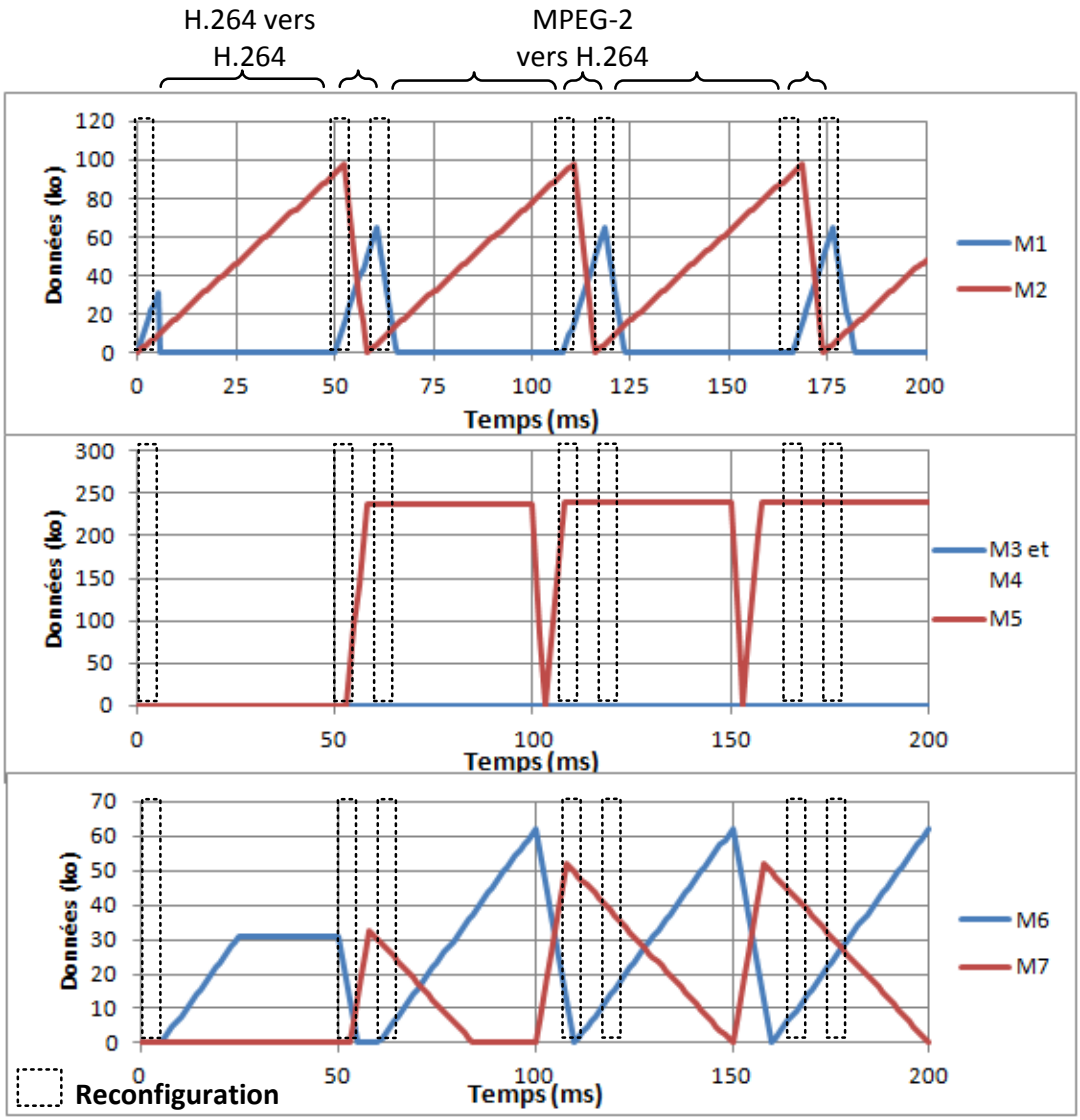


FIGURE 5.7 – Evolution des mémoires intermédiaires pour le cas multi-flux en entrée avec modification des paramètres des IP

transformée et quantification. Nous pouvons calculer les latences maximales requises :

$$Cycles_{MB} = Te_{pixel} \times F_{max} \times Pixels_{MB} \quad (5.4)$$

$$Cycles_{MB} = 12,9ns \times 125Mhz \times 384 = 619cycles/MB \quad (5.5)$$

Soit une latence maximale de 38 cycles entre le traitement de deux bloc 4x4 successifs, au lieu de 56. Encore une fois, il est possible d'atteindre cette objectif en ramenant la latence de la transformée à 7 cycles d'horloge.

5.3.3 Les cas d'utilisation multi-flux en sortie

5.3.3.1 Modèle de l'application

Ce troisième cas d'utilisation est à l'opposé du précédant, cette fois-ci, une seule source est transmise au transcodeur pour une lecture simultanée sur deux terminaux (figure 5.8).

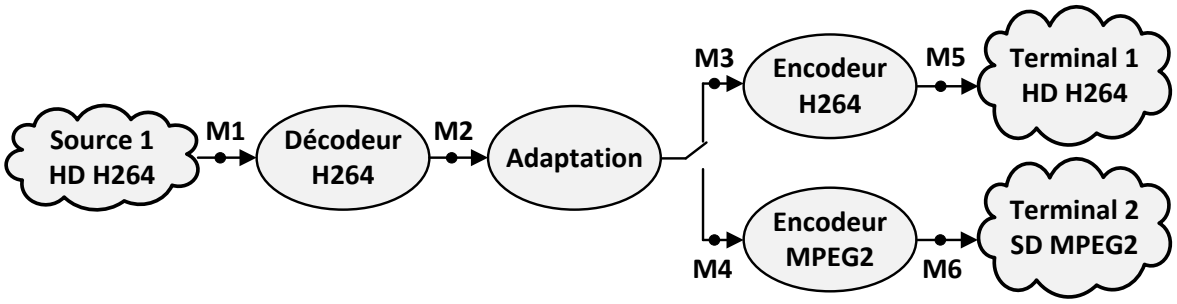


FIGURE 5.8 – Modèle de l'application multi-flux en sortie

La source est un flux HD qui nécessite un décodeur H.264 statique, tandis que les terminaux sont mixtes à la fois en résolution et en codage, soit deux types d'adaptations à réaliser sur le même flux.

5.3.3.2 Résultats

Le flux HD est emmagasiné sans arrêt en entrée de l'encodeur H.264 qui ne peut traiter les données en temps réel. Ce cas est encore plus critique que dans les scénarios précédents, car cet encombrement apparaît en entrée d'une zone reconfigurable. En conséquence, l'ordonnanceur de reconfiguration attribue une forte priorité à l'encodeur H.264 dont la mémoire d'entrée (M3) grandit sans arrêt, l'encodeur MPEG-2 n'est alors jamais configuré, entraînant là aussi une croissance sans fin, mais moins rapide, du contenu de M2. (figure 5.9).

Dans ces conditions, l'utilisateur du terminal HD recevra un flux saccadé, tandis que l'utilisateur du terminal SD ne recevra rien.

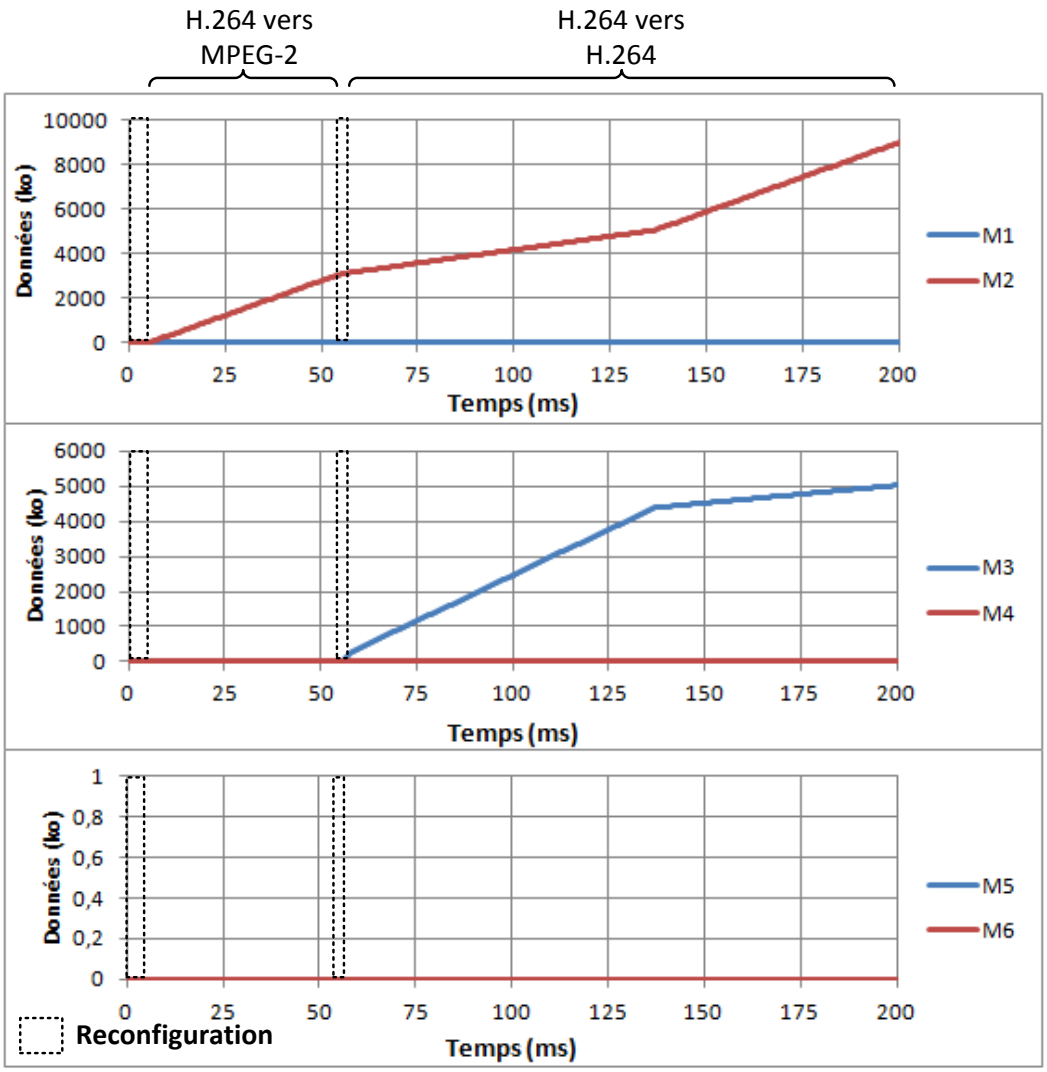


FIGURE 5.9 – Evolution des mémoires intermédiaires pour le cas multi-flux en sortie avec les paramètres actuels

5.3.3.3 Modifications possibles

Une première série de simulations a été réalisée pour déterminer le temps d'exécution idéal de l'encodeur H.264. Malheureusement, dans ce cas de figure, l'action sur l'encodeur seule est insuffisante pour lever les contraintes. Celui-ci étant en aval de l'adaptation, son débit de traitement est cadencé par elle, et ne peut dépasser 100Mo/s.

Pour répondre aux contraintes, une révision du bloc d'adaptation est également nécessaire.

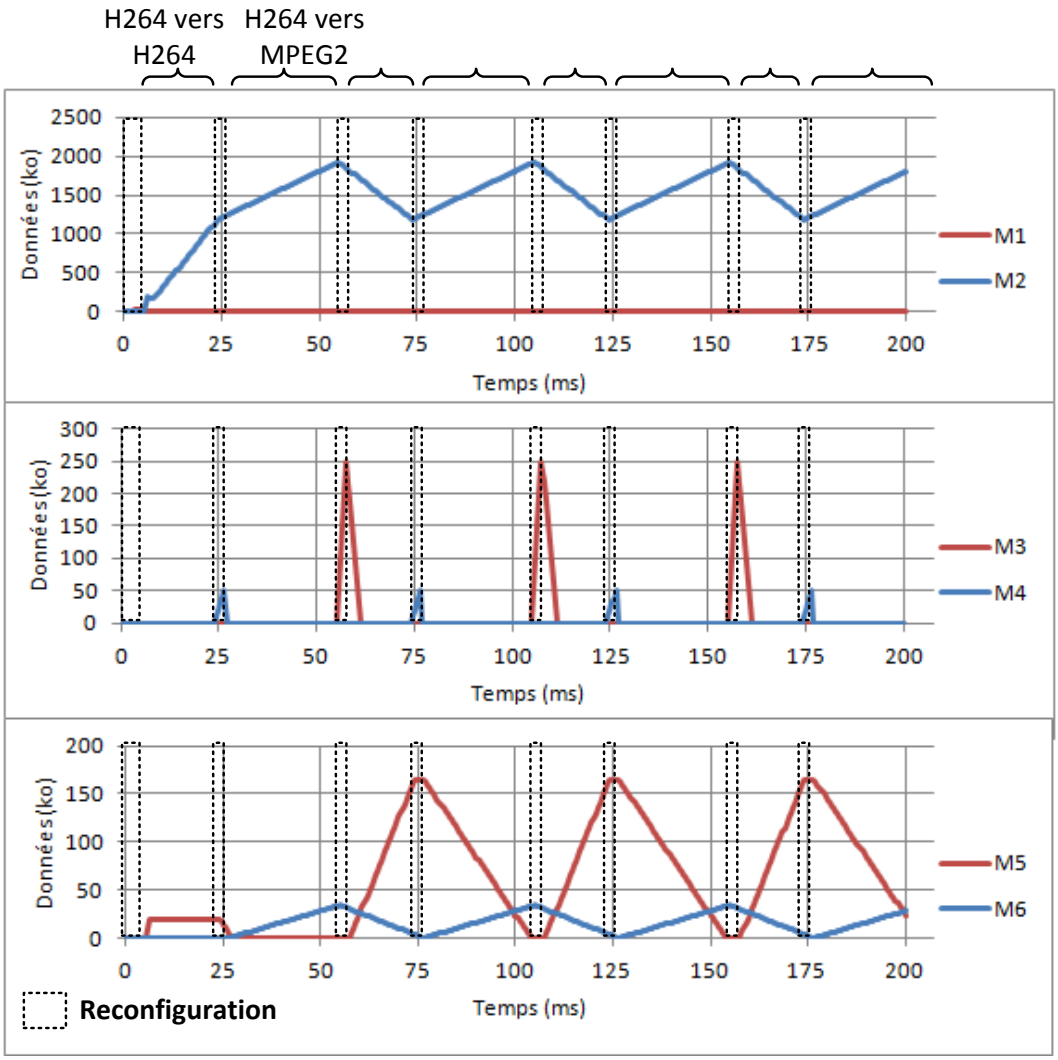


FIGURE 5.10 – Evolution des mémoires intermédiaires pour le cas multi-flux en sortie avec modification des paramètres des IP

Nous diminuons donc les temps d'exécution de ces deux blocs de la même manière, jusqu'à obtenir les débits de sorties attendus. Les différentes modifications et combinaisons de paramètres possibles sont présentées dans le tableau 5.5.

Sur observation des chronogrammes 5.10, avec ces nouveaux paramètres, la zone reconfigurable est occupée à 57% du temps par l'encodeur MPEG-2, contre 32% pour l'encodeur H.264. La longue occupation de la zone par l'encodeur MPEG-2 s'explique par les nouvelles performances des IP H.264 : ces dernières ont un temps d'exécution en deçà des IP MPEG-2

TABLE 5.5 – Performances maximales pouvant être atteintes pour le cas multi-flux en sortie

Résolution max	Frame rate max (images/sec)	Texec Enc. H.264 (ns)	Texec Adapt. (ns)	Mémoires nécessaires (ko)	Taux d'occupation de la zone reconfigurable (% du temps total)
1920x1080	**6**	19	10	500	10% en reconfiguration
1024x600	20	19	10	500	32% en exécution H.264
1920x1080	20	**6.1**	**6.1**	500	57% en exécution MPEG-2

(8ns), par conséquence, leurs traitements nécessitent moins de temps.

– **Exploitation des résultats :**

Le temps d'exécution idéal pour ces deux IP est de 6,1ns, pour une latence de 292 cycles par macrobloc, ou 18 cycles par bloc 4x4 :

$$Cycles_{MB} = Te_{pixel} \times F_{max} \times Pixels_{MB} \quad (5.6)$$

$$Cycles_{MB} = 6,1ns \times 125Mhz \times 384 = 292cycles/MB \quad (5.7)$$

Cette fois-ci, la modification seule du bloc de transformée ne pourra suffire pour atteindre cette latence. Une optimisation du bloc de prédiction et/ou de quantification s'avère utile également.

Une importante mémoire de 2300ko est nécessaire pour ce scénario, ce chiffre s'explique par la nécessité de mémoriser temporairement le flux HD décompressé en entrée du bloc d'adaptation (mémoire M2).

5.3.4 Les cas d'utilisation multi-flux en entrée et sortie

5.3.4.1 Modèle de l'application

Ce dernier cas d'utilisation (figure 5.11) est une combinaison des deux précédents dans le sens où cette fois, deux sources vidéo sont transmises au transcodeur afin d'être acheminées vers deux terminaux de lecture.

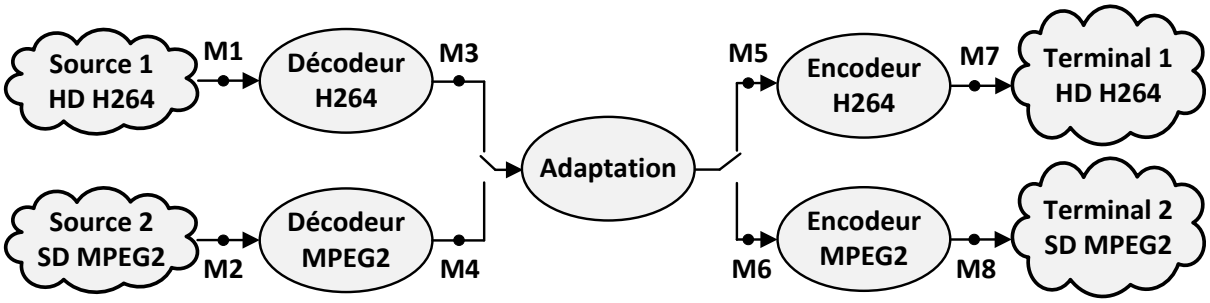


FIGURE 5.11 – Modèle de l'application multi-flux en entrée et sortie

Il s'agit donc ici de traiter un flux HD et SD en parallèle sur une seule chaîne de transcodage, pour cela, toutes les IP de transcodage sont sollicitées, la reconfiguration dynamique est utilisée sur les deux zones reconfigurables.

5.3.4.2 Résultats

Les premiers résultats obtenus (figure 5.12) peuvent être analysés en deux temps :

- Côté décodage, les flux entrants sont temporisés par les mémoires M1 et M2, puis consommés en alternance par le décodeur H.264 et MPEG-2. Ce cycle est régulier et ne pose aucun problème d'encombrement de données.
- Au niveau de l'encodage, nous observons également une alternance entre les deux codecs. Toutefois, on se rend vite compte que le flux HD n'est pas encodé assez rapidement, et s'engouffre dans la mémoire M5. Autre conséquence de ce manque de rapidité : l'encodeur H.264 occupe à lui seul, quasiment 90% du temps la zone reconfigurable, l'encodeur MPEG-2 n'est alors que rarement présent. De ce fait, le flux SD n'est pas traité en totalité, entraînant là aussi une croissance sans fin du contenu de M6.

5.3.4.3 Modifications possibles

Le tableau 5.6 présente les modifications de paramètres possibles afin de traiter ce scénario convenablement. Cette fois-ci, il n'est pas utile de modifier l'IP d'adaptation, une solution a été repérée en ramenant le temps d'exécution de l'encodeur H.264 à 11,1ns.

Les chronogrammes 5.13 montrent une simulation effectuée avec ce nouveau paramètre,

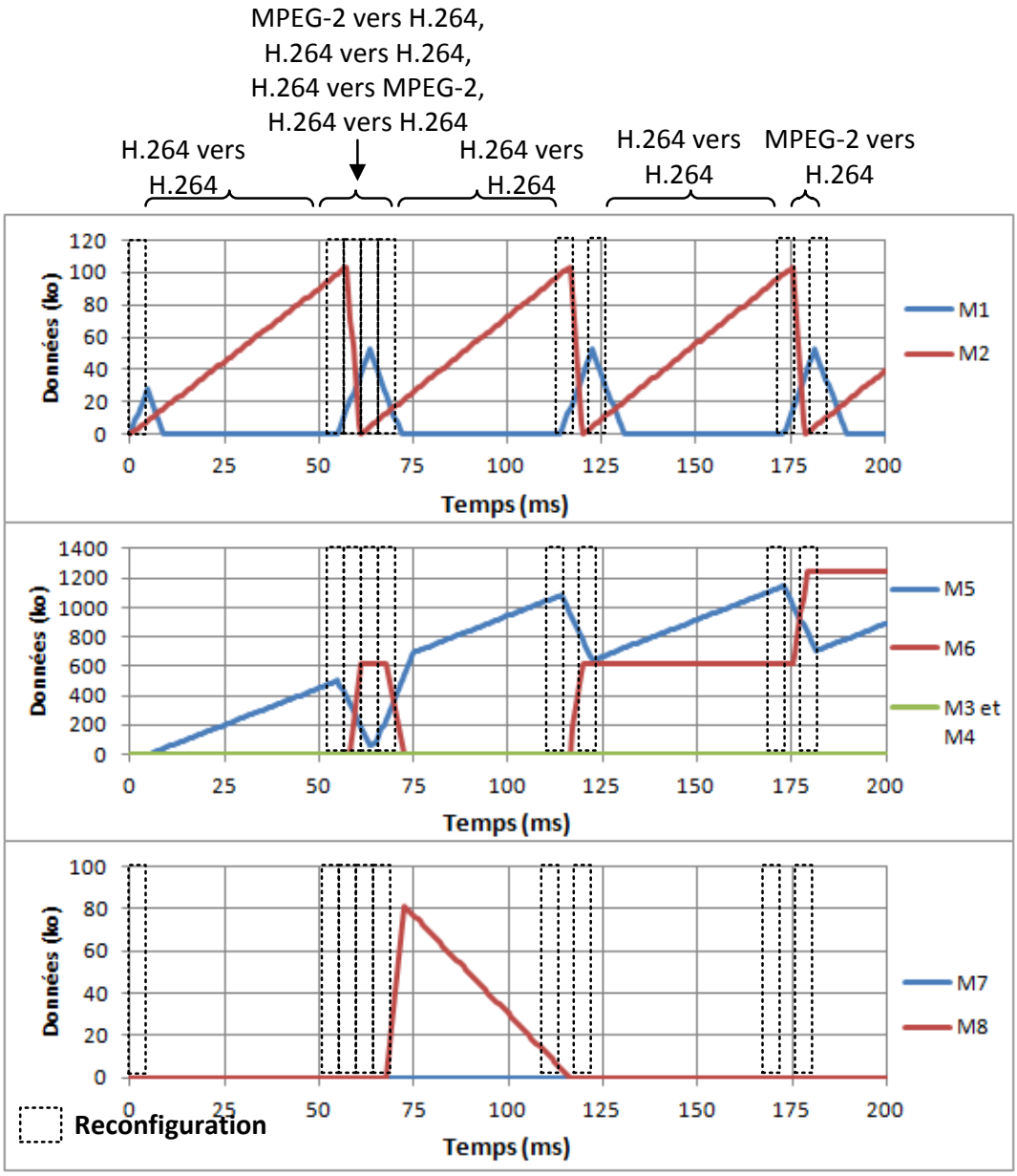


FIGURE 5.12 – Evolution des mémoires intermédiaires pour le cas multi-flux en entrée et sortie avec les paramètres actuels

les zones reconfigurables sont occupées à 70% par les traitements H.264 contre seulement 10% pour MPEG-2, et 20% par les reconfigurations. Aucune mémoire interne n'est utile, seuls 375ko sont nécessaires à la synchronisation des flux en entrée et en sortie de la chaîne de transcodage.

On peut également noter que les reconfigurations des deux zones sont réalisées quasi simultanément étant donné la nature séquentielle du graphe de tâches (décodage, adaptation puis

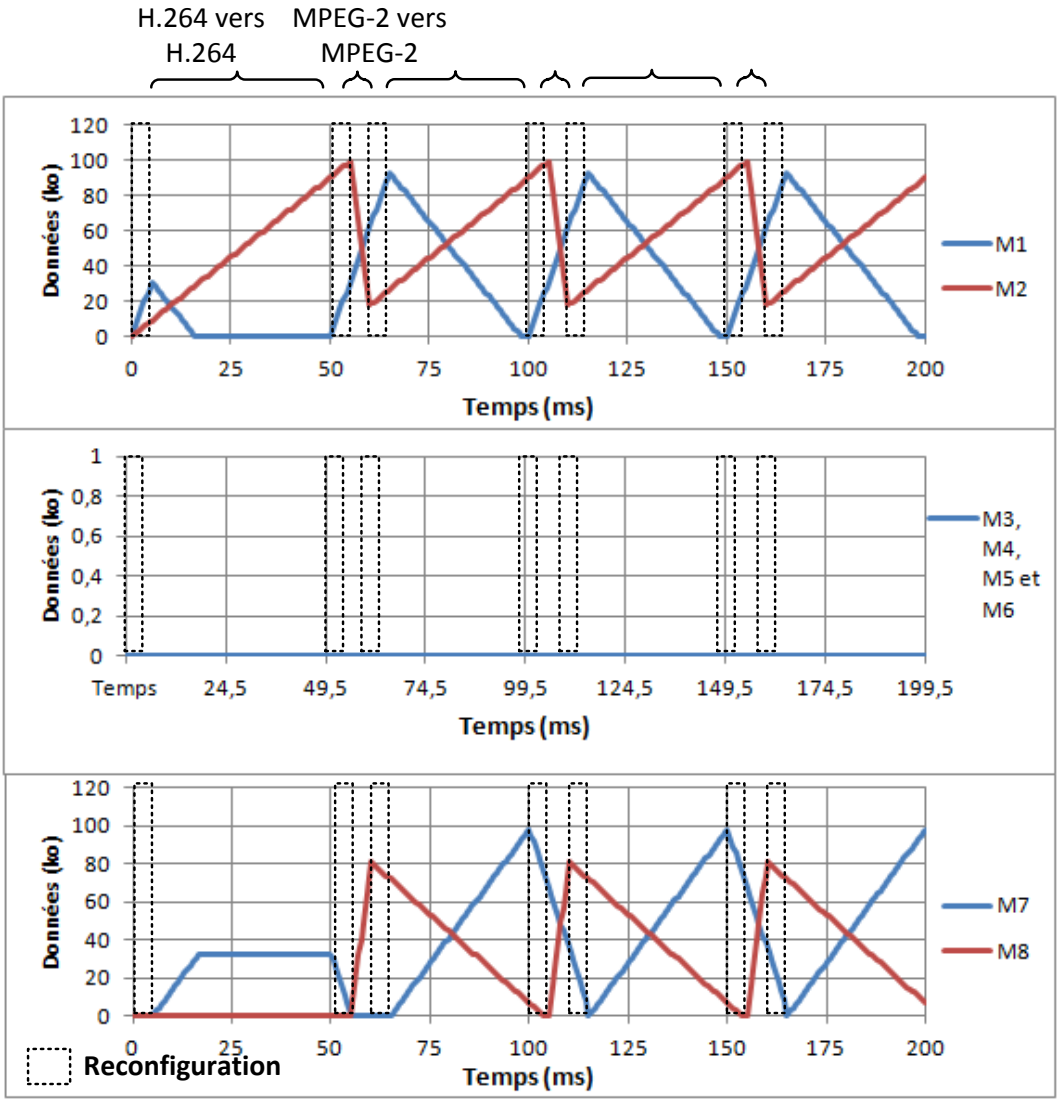


FIGURE 5.13 – Evolution des mémoires intermédiaires pour le cas multi-flux en entrée et sortie avec modification des paramètres des IP

encodage).

– **Exploitation des résultats :**

La latence est de 528 cycles par macrobloc dans ce cas, ou 33 cycles par bloc 4x4. Une diminution de la latence de la transformée à 4 cycles d'horloge suffit ici :

$$Cycles_{MB} = Te_{pixel} \times F_{max} \times Pixels_{MB} \tag{5.8}$$

$$Cycles_{MB} = 11,1ns \times 125Mhz \times 384 = 528cycles/MB \tag{5.9}$$

TABLE 5.6 – Performances maximales pouvant être atteintes pour le cas multi-flux en entrée et sortie

Résolution max	Frame rate max (images/sec)	Texec Enc. H.264 (ns)	Mémoires nécessaires (ko)	Taux d'occupation de la zone reconfigurable (% du temps total)
1920x1080	**11**	19	375	20% en reconfiguration
1280x900	20	19	375	70% en exécution H.264
1920x1080	20	**11.1**	375	10% en exécution MPEG-2

5.4 Synthèse des résultats de simulation

Dans ce chapitre, nous avons exploré par simulation l'ensemble des cas d'utilisation rencontrés par la chaîne de transcodage vidéo.

Sans modification des IP, le tableau 5.7 présente les limites du transcodeur pour chaque cas d'usage, en terme de frame rate et de résolution.

TABLE 5.7 – Synthèse des performances maximales pour chaque cas d'utilisation

	Flux 1		Flux 2	
Cas d'utilisation	Résolution max	Frame rate max	Résolution max	Frame rate max
Mono	1920x1080	**16**	X	X
flux	**1440x1050**	20	X	X
Multi flux	1920x1080	**13**	720x576	20
en entrée	**1440x1050**	20	720x576	20
Multi flux	1920x1080	**6**	720x576	20
en sortie	**1024x600**	20	720x576	20
Multi flux	1920x1080	**11**	720x576	20
en entrée et sortie	**1280x900**	20	720x576	20

La méthodologie d'exploration a permis d'identifier les points de l'architecture à accélérer, avec une stratégie de modification d'un minimum d'IP. Le cas d'utilisation 3 : "multi-flux en sortie" est celui pour lequel les contraintes de temps sont les plus importantes. La diminution du temps d'exécution de l'encodeur H.264 et de l'adaptation permettrait la réalisation de celui-ci,

et par conséquent, de tous les autres. Le tableau 5.8 résume les améliorations nécessaires sur les IP de transcodage pour la réalisation de chaque scénario.

TABLE 5.8 – Synthèse des modifications nécessaires pour chaque cas d'utilisation

		Temps d'exécution	
Cas d'utilisation	IP	Sans modification	Après modification
Mono	Enc. H.264	19ns	**16,1ns**
flux	Adapt.	10ns	10ns
Multi flux	Enc. H.264	19ns	**12,9ns**
en entrée	Adapt.	10ns	10ns
Multi flux	Enc. H.264	19ns	**6,1ns**
en sortie	Adapt.	10ns	**6,1ns**
Multi flux	Enc. H.264	19ns	**11,1ns**
en entrée et sortie	Adapt.	10ns	10ns

Pour terminer, les cas d'utilisation ont été simulés une nouvelle fois avec les nouveaux paramètres (temps d'exécution des IP H.264 et d'adaptation égal à 6,1ns). Cette étape a permis de valider le fonctionnement de chaque scénario, et de déterminer les mémoires nécessaires. La figure 5.14 reprend l'architecture définie dans le projet ARDMAHN (pour une seule chaîne de transcodage), on y retrouve les nouveaux temps d'exécution, les capacités mémoires et leur emplacement. Dans cette configuration (Mémoires et modifications des IP), la chaîne est capable de traiter chacun des quatres cas d'utilisation. Il est tout à fait envisageable de multiplier cette chaîne pour traiter un nombre plus grand de flux (quatre, six, huit, etc).

5.5 Discussion sur la validité des résultats

Toutes les simulations présentées au cours de ce chapitre ont été réalisées à partir des caractéristiques expérimentales présentées au chapitre 4. Ainsi, le nombre et la nature des ressources utilisées pour chaque IP permettent d'obtenir le temps de reconfiguration, et le débit d'entrée et de sortie sont utilisés pour déterminer les temps d'exécution et de latence. Ces données sont les points d'entrée des simulations. Bien que très proches de la réalité ces simulations demandent

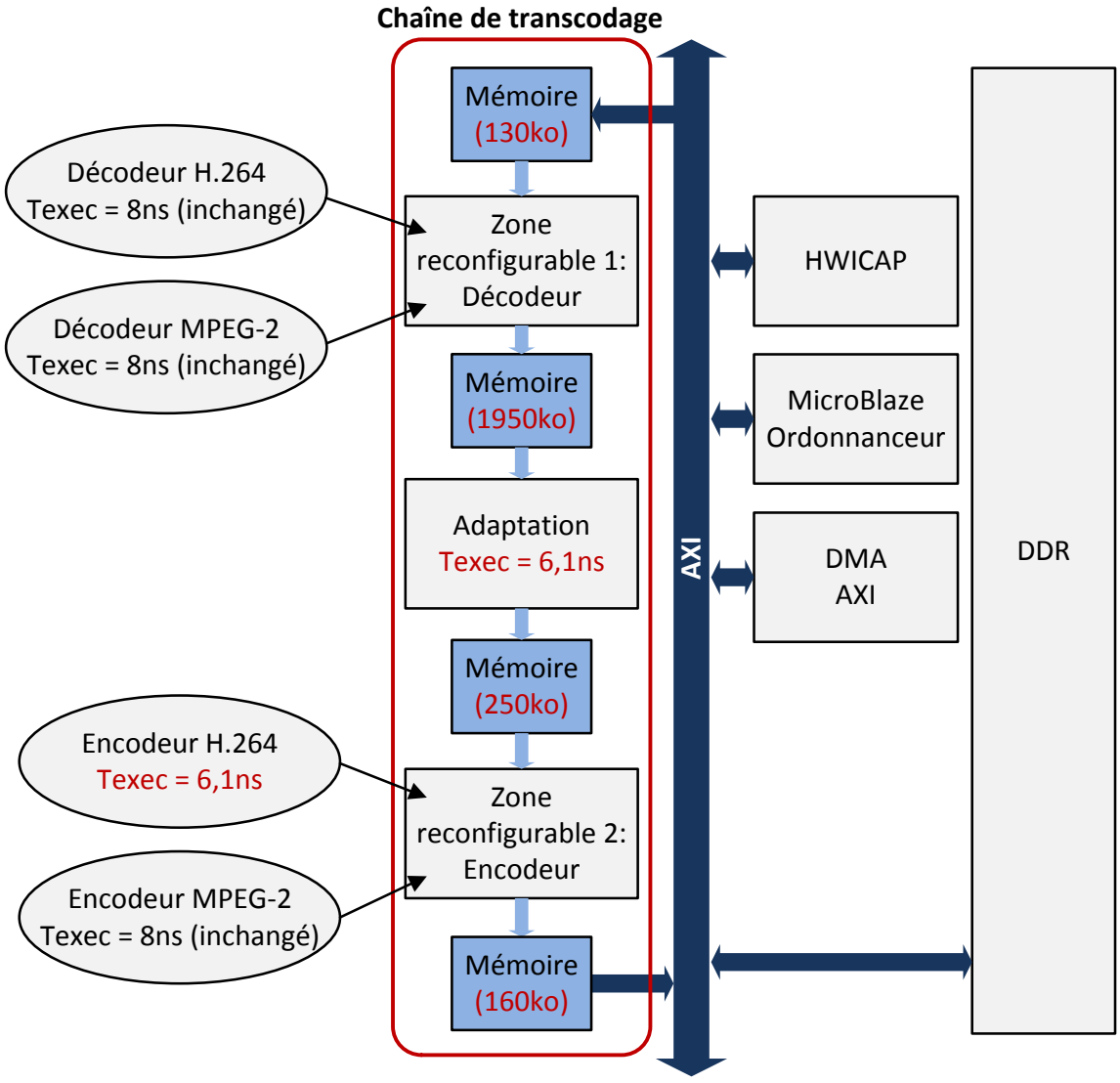


FIGURE 5.14 – Architecture avec dimensionnement des mémoires

à être corroborées avec une analyse expérimentale, ne serait-ce que pour s'assurer qu'aucun paramètre imprévu ne modifie le comportement globale de l'architecture.

A la date de la rédaction de la thèse, le projet est achevé depuis quelques mois déjà. Pour différentes raisons techniques et organisationnelles, l'architecture n'avait pas pu être évaluée dans sa globalité. Seule une évaluation sur une architecture pour MPEG-2 a pu être réalisée et a permis de conforter les solutions proposées.

Dans la suite il sera nécessaire de prévoir d'instrumenter une plate-forme, de sorte à mesurer

les différents débits entrants et sortants des canaux de communication, tout en jouant les scénarios de reconfiguration dynamique. Par ailleurs, en fonction de la nature des mémoires à utiliser (BRAM, DRAM externe, etc), les modèles de simulations des canaux de communication seront à ajuster.

Après ce travail expérimental important, il est envisageable d'accroître les performances des IP d'adaptation et d'encodage H.264 afin de respecter les performances définies par la phase de raffinement de la méthodologie, et ainsi vérifier les performances de la dernière architecture.

5.6 Conclusion

Dans ce chapitre, nous avons alimenté et simulé notre modèle. Cette phase constituait la dernière étape de notre méthodologie vers l'évaluation d'une application reconfigurable dynamiquement à base d'IP réutilisées.

Nous sommes partis d'une bibliothèque d'IP de traitement vidéo, et du cahier des charges du projet ARDMAHN, à savoir : réaliser un transcodeur auto-adaptable par reconfiguration dynamique, capable de traiter plusieurs flux simultanément. Après une phase de caractérisation des IP, et de l'application. nous avons déterminé par la simulation de notre modèle, les performances attendues par la chaîne de transcodage, et identifié les IP à améliorer pour atteindre les limites de débit vidéo. De ces résultats, nous avons organisé une architecture de transcodage complète, en spécifiant les points à améliorer, et les ressources à mettre en place.

Grâce à cette démarche, nous avons validé une première méthode d'ordonnancement de la reconfiguration pour cette application, et estimé les ressources spécifiques à sa mise en place. Sans ce travail, il aurait été difficile, voir impossible, de repérer les IP inappropriées, et encore plus, de définir une liste des améliorations à leurs apporter.

Conclusion générale et perspectives

Avec les dernières avancées technologiques, les systèmes embarqués requièrent toujours plus de puissance de calcul et d'adaptabilité. La technologie de reconfiguration dynamique sur FPGA est une solution envisageable en réponse à ces besoins grandissants. Les architectures matérielles sont capables d'atteindre des vitesses de traitement élevées grâce à une implémentation et une exécution spatiale et parallèle. Elles ont la capacité de pouvoir modifier leur fonctionnement en cours d'exécution, grâce à la reconfiguration dynamique, assurant une certaine adaptabilité au circuit. Malgré ses caractéristiques intéressantes, cette technologie peine à se démocratiser dans les applications industrielles. Le surcoût lié au temps de développement par rapport à une solution statique est rédhibitoire à son utilisation. Le manque d'outils simples et efficaces explique en grande partie la difficulté de mise en œuvre de cette technologie. Aucune méthode officielle ne permet de définir des ressources utiles, ou des performances atteignables par une architecture à reconfiguration dynamique partielle.

Dans le cadre de cette thèse, nous avons proposé une méthodologie et un outil de modélisation pour la conception d'architectures reconfigurables dynamiquement. Le choix d'un scénario de reconfiguration est un travail complexe et décisif, les ressources et les performances de l'architecture dépendent directement de celui-ci. Basé sur la bibliothèque open-source SystemC, notre outil permet la simulation et la vérification de ces scénarios. La fonction SC_SPAWN et les thread dynamiques définies dans ce langage, autorisent la modification des tâches au cours de la simulation. Nous employons cette particularité pour modéliser le comportement de la reconfiguration dynamique dans notre travail.

Nous avons vu que beaucoup de travaux proposent des algorithmes pour le dimensionnement et l'ordonnancement des zones reconfigurables. Mais peu d'entre eux prennent en considération les contraintes fixées par l'application. Notre méthodologie repose sur une étude approfondie du fonctionnement statique de l'application, et une caractérisation de chacun de ses éléments avant de les modéliser. Grâce à cette étape, nous nous assurons du respect des contraintes et performances de l'application statique, dans le domaine dynamique.

Cette approche se place dans le contexte de la conception par utilisation ou réutilisation d'IP. Chacune d'entre elle est caractérisée en configuration statique afin d'en extraire les paramètres nécessaires à la méthodologie proposée. La simulation du modèle ainsi constitué, permet de mesurer le degré d'engorgement sur les canaux de communication. Une exploration par libération des paramètres de chaque bloc de traitement est ensuite réalisée, pour identifier le ou les IP constituants éventuellement le goulot d'étranglement de l'architecture.

Pour valider notre modèle, nous avons testé notre méthode sur une application de transcodage vidéo, dont les fortes contraintes temporelles, et les importantes ressources de calcul et de mémorisation, en font un candidat d'évaluation intéressant. Le transcodage s'appuie sur une bibliothèque d'IP MPEG-2 caractérisées sur une architecture statique. Les IP H.264 quant à elles, ont été développées au cours de la thèse et caractérisées de la même façon. Les différents cas d'usage de l'application ont ensuite été modélisés et alimentés pas les paramètres issus de chaque IP.

Enfin, la simulation de l'application, a permis d'identifier avec précision les besoins de performances de chaque IP, et de quantifier la chaîne de transcodage reconfigurable. Des propositions d'améliorations, et d'ordonnancements ont ensuite été faites afin de respecter les contraintes temps réel. Elles portent sur la modification des IP d'encodage H.264 et d'adaptation, dont les latences doivent être revues et améliorées.

La méthode proposée prend ainsi forme en un flot de conception (figure 1) intéractif entre les outils de modélisation SystemC et les outils de développement Xilinx. Et passe par des étapes de caractérisation des IP et de l'application, de modélisation de l'architecture reconfigurable, de

simulation de l'application, et enfin, d'exploration et de raffinement des résultats, afin de mener le développeur d'une architecture reconfigurable jusqu'à la phase finale d'implémentation sur carte.

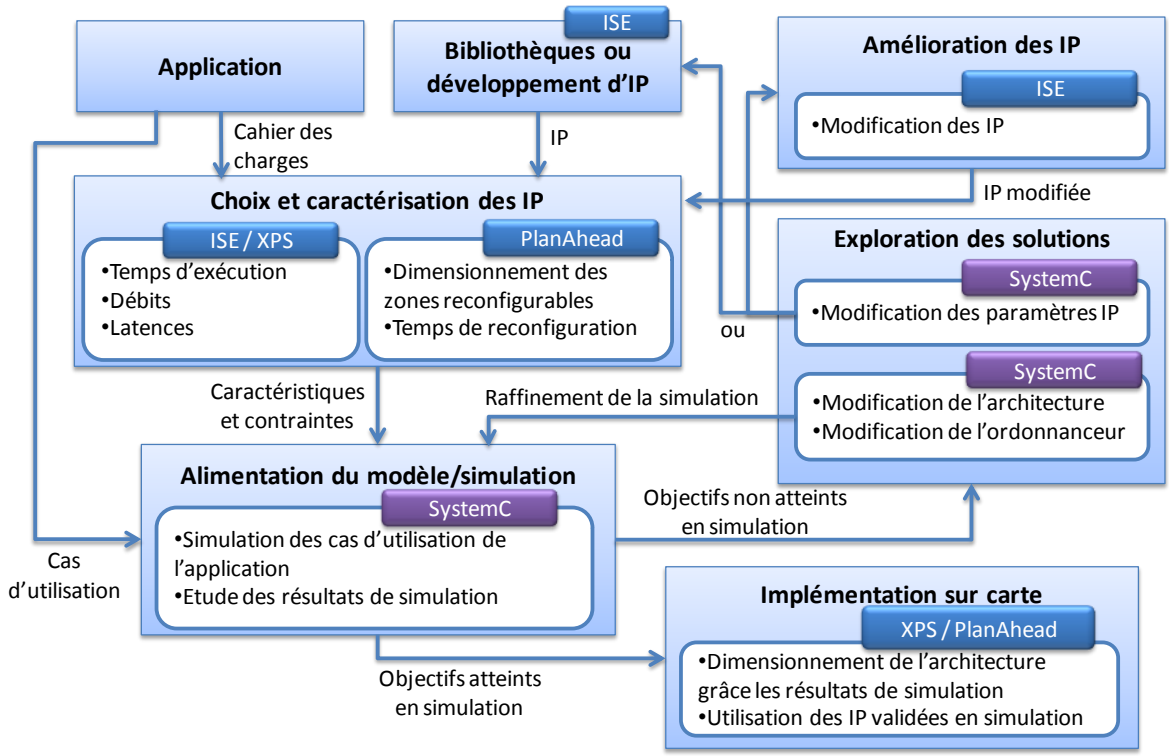


FIGURE 1 – Flot de conception proposé

La poursuite de ce travail, au niveau application, serait de reprendre le développement des IP d'encodage H.264 afin d'atteindre la latence déduite par la simulation. Puis d'implémenter l'ensemble de l'architecture pour comparer les résultats réels avec la simulation.

En ce qui concerne notre modèle, les résultats présentés ne prennent en compte qu'une seule méthode d'ordonnancement de la reconfiguration. Par la suite, il serait intéressant d'étudier d'autres algorithmes, et de les intégrer à notre outil. De cette façon, l'utilisateur serait libre d'exploiter les résultats de plusieurs ordonnanceurs, pour ne retenir que les plus adaptés.

Enfin, une dernière amélioration à apporter à notre outil, serait d'automatiser l'exploration des instants de reconfiguration. Le rapport entre temps de reconfiguration et temps de calcul

dépend de plusieurs paramètres, comme la taille du bitstream et la taille des paquets de données à traiter. Dans le cas de notre application le paquet est une image entière. Or, ce paramètre a été choisi arbitrairement pour notre application, il serait intéressant de pouvoir explorer, à la baisse ou à la hausse, la taille de ce paquet et d'en étudier l'influence.

Annexes

Sommaire

1 Glossaire 124

2 Rapports de synthèse : surface occupée par l'encodeur H.264 126

3 Interfaces du transcodeur H.264/MPEG-2 130

4 Codes Microblaze 131

1 Glossaire

ARDMAHN :	Architecture Reconfigurables Dynamiquement pour l'Auto-adaptation en Home Networking
ADSL :	Asymmetric Digital Subscriber Line
VDSL :	Very high rate Digital Subscriber Line
FTTH :	Fiber To The Home
ASIC :	Application-Specific Integrated Circuit
CODEC :	COde-DECode
HD :	High Definition
SD :	Standard Definition
HEVC :	High Efficiency Video Coding
MPEG :	Moving Picture Experts Group
MB :	MacroBloc
DCT :	Discrete Cosine Transform
IDCT :	Inverse Discrete Cosine Transform
TE :	Transformée Entière
ITE :	Transformée Entière Inverse
QP :	Quantification Parameter
VLC :	Variable Length Coding
CAVLC :	Context-based Adaptive Variable Length Coding
PSNR :	Peak Signal to Noise Ratio
GPU :	Graphics Processing Units
CPU :	Central Processing Unit
SOC :	System On Chip
FPGA :	Field Programmable Gate Array

CLB :	Configurable Logic Block
BRAM :	Block RAM
DSP :	Digital Signal Processor
IOB :	Input, Output Block
FPS :	Frame Per Second
GOP :	Group Of Pictures
UML :	Unified Modeling Language
SysML :	System Modeling Language
MARTE :	Modeling and Analysis of Real-Time Embedded Systems
MDE :	Model Driven Engineering
RTL :	Register Transfer Level
IP :	Intellectual Property
TLM :	Transaction Level Modeling
CABA :	Cycled Accurate, Bit Accurate
UTF :	UnTimed Functional
TF :	Timed Functional
BCA :	Bus Cycle Accurate
PiP :	Picture in Picture
SAD :	Sum of Absolute Differences
NAL :	Network Abstraction Layer
FSL :	Fast Simplex Link
AXI :	Advanced eXtensible Interface
FIFO :	First In, First Out
HWICAP :	HardWare Internal Cofiguration Access Port

2 Rapports de synthèse : surface occupée par l'encodeur H.264

Dans cette annexe, sont regroupés les rapports de synthèse post placement et routage des IP développées pour le standard H.264. Ainsi que le rapport de synthèse de l'encodeur H.264 complet.

Transformée entière			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	536	301440	0%
Number of Slice LUTs	1630	150720	1%
Number of fully used LUT-FF pairs	387	1779	21%
Number of bonded IOBs	45	720	6%
Number of BUFG/BUFGCTRLs	1	32	3%

FIGURE 1 – Rapport de synthèse post place and root pour l'IP transformée entière

Prédiction intra			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	763	301440	0%
Number of Slice LUTs	12454	150720	8%
Number of fully used LUT-FF pairs	365	12852	2%
Number of bonded IOBs	102	600	17%
Number of BUFG/BUFGCTRLs	1	32	3%

FIGURE 2 – Rapport de synthèse post place and root pour l'IP prédiction intra

Quantification			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	56	301440	0%
Number of Slice LUTs	113	150720	0%
Number of fully used LUT-FF pairs	40	129	31%
Number of bonded IOBs	51	600	8%
Number of BUFG/BUFGCTRLs	1	32	3%
Number of DSP48E1s	1	768	0%

FIGURE 3 – Rapport de synthèse post place and root pour l'IP quantification

Transformée inverse			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	526	301440	0%
Number of Slice LUTs	1192	150720	0%
Number of fully used LUT-FF pairs	390	1328	29%
Number of bonded IOBs	49	600	8%
Number of BUFG/BUFGCTRLs	1	32	3%

FIGURE 4 – Rapport de synthèse post place and root pour l'IP transformée inverse

Quantification inverse			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	31	301440	0%
Number of Slice LUTs	71	150720	0%
Number of fully used LUT-FF pairs	23	79	29%
Number of bonded IOBs	51	600	8%
Number of BUFG/BUFGCTRLs	1	32	3%
Number of DSP48E1s	1	768	0%

FIGURE 5 – Rapport de synthèse post place and root pour l'IP quantification inverse

CAVLC			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	203	301440	0%
Number of Slice LUTs	1898	150720	1%
Number of fully used LUT-FF pairs	156	1945	8%
Number of bonded IOBs	60	600	10%
Number of BUFG/BUFGCTRLs	1	32	3%

FIGURE 6 – Rapport de synthèse post place and root pour l'IP CAVLC

Exp Golomb			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	109	301440	0%
Number of Slice LUTs	376	150720	0%
Number of fully used LUT-FF pairs	100	385	25%
Number of bonded IOBs	63	600	10%
Number of BUFG/BUFGCTRLs	1	32	3%

FIGURE 7 – Rapport de synthèse post place and root pour l'IP Exp Golomb

Parseur			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	146	301440	0%
Number of Slice LUTs	1870	150720	1%
Number of fully used LUT-FF pairs	129	1887	6%
Number of bonded IOBs	71	600	11%
Number of BUFG/BUFGCTRLs	1	32	3%

FIGURE 8 – Rapport de synthèse post place and root pour l'IP parseur

Architecture complète			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	9060	301440	3%
Number of Slice LUTs	25709	150720	17%
Number of fully used LUT-FF pairs	4503	30266	14%
Number of bonded IOBs	232	600	38%
Number of BUFG/BUFGCTRLs	1	32	3%
Number of DSP48E1s	2	768	0%

FIGURE 9 – Rapport de synthèse post place and root pour l'encodeur complet

3 Interfaces du transcodeur H.264/MPEG-2

La figure 10 présente le transcodeur H.264/MPEG-2 implémenté, et les différents éléments d'interconnexion mis en place.

– Une mémoire RAM est utilisée pour stocker les flux encodés en entrée et en sortie, celle-ci communique avec les deux Microblaze à travers un bus de communication AXI.

– Deux liaisons FSL réalise la synchronisation entre les Microblaze et le transcodeur. Ces liaison utilise des mémoire FIFO.

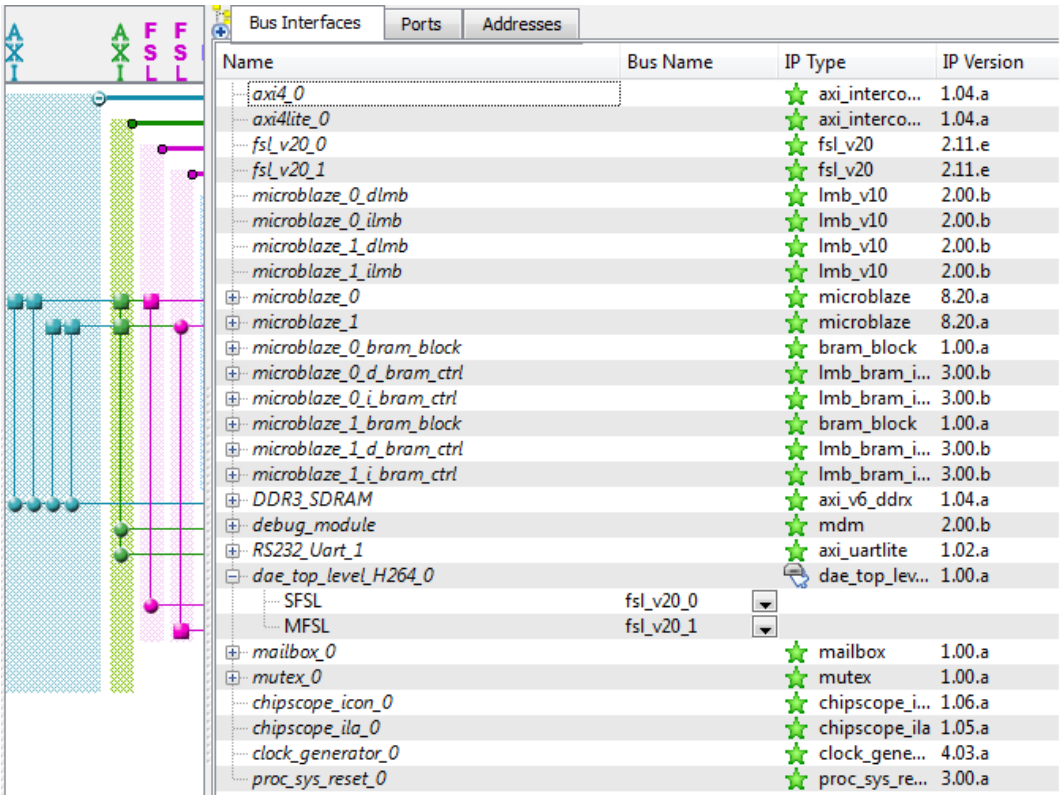


FIGURE 10 – Architecture et interfaces du transcodeur H.264/MPEG-2

4 Codes Microblaze

– Description : Programme Microblaze 0 : Ce programme va lire les données 8 bits dans la RAM, les concatène sur 32 bits, et les transmet au transcodeur via la FSL

```
Xuint8  Reg_In[4];
int i=0;
int j=0;
Xuint8 data_in[4];
int main()
{
    Xuint8 *MemRAM;
    MemRAM= (Xuint8 *)0xc1000000;
    for(i = 0; i < 8000; i++)
    {
        for(j = 0; j < 4; j++)
        {
            Reg_In[j]=MemRAM[(4*i+j)];
        }
        putfslx(Reg_In[3]+Reg_In[2]*256+Reg_In[1]*256*256+Reg_In[0]
        *256*256*256,0,FSL_DEFAULT);
    }
    return 0;
}
```

– Description : Programme Microblaze 1 : Ce programme récupère les données en sortie du transcodeur, et les enregistre dans la RAM

```
int i=0;
Xuint32 data;
int main()
{
    Xuint32 *MemRAM2;
    MemRAM2= (Xuint32 *)0xc2000000;
    while(1)
    {
        getfslx(MemRAM2[i],0,FSL\_DEFAULT);
        i++;
    }
    return 0;
}
```

Publications personnelles

[1] E. Dabellani, H. Rabah, N. Marques, Y. Berviller, S. Jovanovic, and S.Weber, Modeling and FPGA implementation of reconfigurable transcoder for real time video adaptation, IEEE International Conference on Electronics, Circuits, and Systems (ICECS).

[2] N. Marques, H. Rabah, S. Jovanovic, E. Dabellani, and S. Weber, Methodology and reconfigurable architecture for effective placement of variable-size hardware tasks, Colloque national, GDR SOC-SIP 2013, Lyon, France.

[3] N. Marques, H. Rabah, S. Jovanovic, E. Dabellani, and S. Weber, Methodology and reconfigurable architecture for effective placement of variable-size hardware tasks, (AHS-2013) NASA/ESA Conference on Adaptive Hardware and Systems.

[4] E. Dabellani, Modélisation et estimation des performances de la reconfiguration dynamique dans le cadre du transcodage vidéo, Ecole d'hiver Francophone sur les Technologies de Conception des Systèmes embarqués Hétérogènes 2012.

[5] N. Marques, H. Rabah, E. Dabellani, and S.Weber, A novel framework for the design of adaptable reconfigurable regions for the placement of variable-sized ip cores, Embedded Systems Letters, IEEE, 2012, Article soumis en Juin 2012.

[6] E. Dabellani, N. Marques, H. Rabah, Y. Berviller, and S. Weber, Utilisation des threads dynamiques pour la modélisation de la reconfiguration dynamique, Colloque national, GDR SOC-SIP 2011, Cergy, France.

[7] M. Guarisco, E. Dabellani, N. Marques, H. Rabah, Y. Berviller, and S. Weber, An effcient vlsi implementation of h.264/avc intra-frame transcoder, in Electronics, Circuits and Systems (ICECS), 18 th IEEE International Conference on, dec. 2011, pp. 1 - 4.

[8] N. Marques, H. Rabah, E. Dabellani, and S. Weber, Partially reconfigurable entropy encoder for multi standards video adaptation, in Consumer Electronics (ISCE), IEEE 15th International Symposium on, june 2011, pp. 492 - 496.

[9] N. Marques, H. Rabah, E. Dabellani, and S. Weber, Effcient reconfigurable entropy coder for embedded multi-standards video adaptation, in Computer Vision and Pattern Recognition Workshops (CVPRW), IEEE Computer Society Conference on, june 2011, pp. 144 - 149.

[10] E. Dabellani, N. Marques, H. Rabah, Y. Berviller, and S. Weber, Reconfiguration dynamique d'un analyseur syntaxique pour le transcodage et l'adaptation vidéo, Colloque national, GDR SOC-SIP 2010, Cergy, France.

[11] N. Marques, H. Rabah, E. Dabellani, Y. Berviller, and S. Weber, Architecture reconfigurable pour le transcodage vidéo mpeg appliqué à la conservation et l'archivage de contenus vidéo, in 5éme workshop Applications Médicales de l'Informatique, Amina 2010, Monastir, Tunisie.

[12] M. Guarisco, N. Marques, E. Dabellani, Y. Berviller, H. Rabah, and S. Weber, Conception de sopc pour application multimédia, 11èmes journées pédagogiques, JPCNFM 2010, Saint Malo, France.

Bibliographie

[AKK08] Kenji ASANO, Junji KITAMICHI et Kenichi KURODA : Dynamic module library for system level modeling and simulation of dynamically reconfigurable systems. *Journal of computers*, 3(2):55–62, 2008. x, 39, 40

[ARD] Consortium ARDMAHN : Site du projet ardmahn - http ://ardmahn.org/. 2

[BKH+07] Alisson V BRITO, Matthias KUHNLE, Michael HUBNER, Juergen BECKER et Elmar UK MELCHER : Modelling and simulation of dynamic and partially reconfigurable systems using systemc. *In VLSI, 2007. ISVLSI'07. IEEE Computer Society Annual Symposium on*, pages 35–40. IEEE, 2007. 39

[Bla10] David C BLACK : *SystemC : From the ground up*, volume 71. Springer, 2010. 48

[BLN+06] K BABIONITAKIS, G LENTARIS, K NAKOS, D REISIS, N VLASSOPOULOS, G DOUMENIS, G GEORGAKARAKOS et J SIFNAIOS : An efficient h. 264 vlsi advanced video encoder. *In Electronics, Circuits and Systems, 2006. ICECS'06. 13th IEEE International Conference on*, pages 545–548. IEEE, 2006. 28

[BSM10] Alisson Vasconcelos BRITO, George SILVEIRA et Elmar Uwe Kurt MELCHER : A methodology for modelling and simulation of dynamic and partially reconfigurable systems. *Dynamic Modelling*, 1, 2010. 39

[CAK+09] Ngai-Man CHEUNG, Oscar C AU, Man-Cheung KUNG, Peter HW WONG et Chun Hung LIU : Highly parallel rate-distortion optimized intra-mode decision on multicore graphics processors. *Circuits and Systems for Video Technology, IEEE Transactions on*, 19(11):1692–1703, 2009. 24

[CFAK10] Nagai-Man CHEUNG, Xiaopeng FAN, Oscar C AU et Man-Cheung KUNG : Video coding on multicore graphics processors. *Signal Processing Magazine, IEEE*, 27(2):79–89, 2010. 24

[CLSG06] Chih-Da CHIEN, Keng-Po LU, Yi-Hung SHIH et Jiun-In GUO : A high performance cavlc encoder design for mpeg-4 avc/h. 264 video coding applications. *In Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4–pp. IEEE, 2006. 36

[CRGM11] Juan Antonio CLEMENTE, Javier RESANO, Carlos GONZALEZ et Daniel MOZOS : A hardware implementation of a run-time scheduler for reconfigurable systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(7):1263–1276, 2011. 43, 56

[CWLY09] Yi-Chih CHAO, Shih-Tse WEI, Bin-Da LIU et Jar-Ferr YANG : Combined cavlc decoder, inverse quantizer, and transform kernel in compact h. 264/avc decoder. *Circuits and Systems for Video Technology, IEEE Transactions on*, 19(1):53–62, 2009. 36

[CWYL06] Yi-Chih CHAO, Shih-Tse WEI, Jar-Ferr YANG et Bin-Da LIU : Combined cavlc decoder and inverse quantizer for efficient h. 264/avc decoding. *In Circuits and Systems, 2006. APCCAS 2006. IEEE Asia Pacific Conference on*, pages 259–262. IEEE, 2006. 36

[DBKW05] Matthias DICK, Jens BRANDT, Verena KAHMANN et Lars WOLF : Adaptive transcoding proxy architecture for video streaming in mobile networks. *In Image Processing, 2005. ICIP 2005. IEEE International Conference on*, volume 3, pages III–700. IEEE, 2005. 23

[DCB+09] C DEKNUDT, P CORLAY, AS BACQUET, FX COUDOUX, M GUARISCO, H RABAH, Y BERVILLER et S WEBER : Transrating by frequencies selectivity for h. 264/avc intra pictures. *In Broadband Multimedia Systems and Broadcasting, 2009. BMSB'09. IEEE International Symposium on*, pages 1–7. IEEE, 2009. 36

[DMA+13] François DUHEM, Fabrice MULLER, Willy AUBRY, Bertrand Le GAL, Daniel NÉGRU et Philippe LORENZINI : Design space exploration for partially reconfigurable architectures in real-time systems. *Journal of Systems Architecture*, 2013. 42

[DML11a] François DUHEM, Fabrice MULLER et Philippe LORENZINI : Farm : Fast reconfiguration manager for reducing reconfiguration time overhead on fpga. *In Reconfigurable Computing : Architectures, Tools and Applications*, pages 253–260. Springer, 2011. 54

[DML+11b] François DUHEM, Fabrice MULLER, Philippe LORENZINI *et al.* : Dynamic and partial reconfiguration transaction-level modeling in systemc. *In Colloque GDR SoC/SiP (System On Chip-System In Package)*, 2011. 42

[Dou] Benjamin DOURIEZ : Communication internet : La vérité sur le débit des connexions internet - http ://www.60millions-mag.com/actualites/articles/la_verite_sur_le_debit_des_connexions_internet. 9

[DPW99] Didier DEMIGNY, Michel PAINDAVOINE et Serge WEBER : Architecture à reconfiguration dynamique : Application au traitement temps réel des images. *TSI. Technique et science informatiques*, 18(10):1087–1112, 1999. 34

[DT05] Vijay DEGALAHAL et Tim TUAN : Methodology for high level estimation of fpga power consumption. *In Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 1, pages 657–660. IEEE, 2005. 2

[FFm] Projet FFMPEG : Site web ffmpeg - http ://www.ffmpeg.org/. 22

[FUJ09] FUJITSU : Full hd h.264/mpeg-2 encoder-transcoder ics. Rapport technique, FUJITSU, 2009. 26

[GM07] Tony Gladvin GEORGE et N MALMURUGAN : The architecture of fast h. 264 cavlc decoder and its fpga implementation. *In Intelligent Information Hiding and*

Multimedia Signal Processing, 2007. IIHMSP 2007. Third International Conference on, volume 2, pages 389–392. IEEE, 2007. 36

[GO02] Eike GRIMPE et Frank OPPENHEIMER : Aspects of object oriented hardware modelling with systemc-plus. *In System on Chip Design Languages*, pages 213–223. Springer, 2002. 41

[Groa] Object Management GROUP : Site web marte - http ://www.omgmarte.org/. 37

[Grob] Object Management GROUP : Site web uml - http ://www.uml.org/. 37

[HEN08] Jean-Pierre HENOT : Powerful video compression for professional hd applications mustang asic. Rapport technique, THOMSON, 2008. 26

[HSB06] M HUBNER, Christian SCHUCK et Jürgen BECKER : Elementary block based 2-dimensional dynamic and partial reconfiguration for virtex-ii fpgas. *In Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006. 34

[Ini] Open SystemC INITIATIVE : Site web systemc - http ://www.systemc.org/. 38

[Ins] Fraunhofer Heinrich Hertz INSTITUTE : Site web jm - http ://iphome.hhi.de/suehring/tml/. 22

[JZy09] Zhang JIAN et Deng ZHI-YONG : Low complexity transcoder for mpeg2 to h. 264. *In Computer Science and Engineering, 2009. WCSE'09. Second International Workshop on*, volume 1, pages 242–245. IEEE, 2009. 22

[Kao05] Cindy KAO : Benefits of partial reconfiguration. *Xcell journal*, 55:65–67, 2005. 2

[KOW08] Cyril KOWALISKI : Badaboom 1.0 uses nvidia gpus to transcode video. Rapport technique, nvidia, 2008. 25

[LAH04] Oussama LAYAIDA, Slim Ben ATALLAH et Daniel HAGIMONT : A framework for dynamically configurable and reconfigurable network-based multimedia adaptations. *Journal of Internet Technology*, 5(4):363–372, 2004. 23

[LBM+06] Patrick LYSAGHT, Brandon BLODGET, Jeff MASON, Jay YOUNG et Brendan BRIDGFORD : Invited paper : Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas. *In Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, pages 1–6. IEEE, 2006. 28

[LKLJ09] Ming LIU, Wolfgang KUEHN, Zhonghai LU et Axel JANTSCH : Run-time partial reconfiguration speed investigation and architectural design space exploration. *In Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 498–502. IEEE, 2009. 99

[MD02] Wei-hsiu MA et David Hung-Chang DU : Reducing bandwidth requirement for delivering video over wide area networks with proxy server. *Multimedia, IEEE Transactions on*, 4(4):539–550, 2002. 23

[MFA+07] Sandro MOIRON, Sergio FARIA, Pedro ASSUNCAO, Vitor SILVA et Antonio NAVARRO : H. 264/avc to mpeg-2 video transcoding architecture. *In Proceedings of the Conference on Telecommunications*, pages 449–452, 2007. ix, xiii, 22, 23

[MRDW11a] Nicolas MARQUES, Hassan RABAH, E DABELLANI et S WEBER : Efficient reconfigurable entropy coder for embedded multi-standards video adaptation. *In Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 144–149. IEEE, 2011. 34, 36

[MRDW11b] Nicolas MARQUES, Hassan RABAH, E DABELLANI et S WEBER : Partially reconfigurable entropy encoder for multi standards video adaptation. *In Consumer Electronics (ISCE), 2011 IEEE 15th International Symposium on*, pages 492–496. IEEE, 2011. 36

[Par] SysML PARTNERS : Site du projet sysml - http ://www.sysml.org/. 37

[PHBB07] Katarina PAULSSON, Michael HÜBNER, Salih BAYAR et Jürgen BECKER : Exploitation of run-time partial reconfiguration for dynamic power management in xilinx spartan iii-based systems. *In ReCoSoc*, pages 1–6, 2007. 2

[Pol] Alexis POLTI : Tp systemc inf 227 - http ://perso.telecom-paristech.fr/ polti/enseignement/tp_systemc/. x, 38

[QMD08] Imran Rafiq QUADRI, Samy MEFTALI et J-L DEKEYSER : Marte based modeling approach for partial dynamic reconfigurable fpgas. *In Embedded Systems for Real-Time Multimedia, 2008. ESTImedia 2008. IEEE/ACM/IFIP Workshop on*, pages 47–52. IEEE, 2008. x, 37, 38

[QMM+09] Imran Rafiq QUADRI, Alexis MULLER, Samy MEFTALI, Jean-Luc DEKEYSER *et al.* : Marte based design flow for partially reconfigurable systems-on-chips. *In 17th IFIP/IEEE international conference on very large scale integration (VLSI-SoC 09)*, 2009. 38

[RB07] Choudhury A RAHMAN et Wael BADAWY : Cavlc encoder design for real-time mobile video applications. *Circuits and Systems II : Express Briefs, IEEE Transactions on*, 54(10):873–877, 2007. 36

[RF08] Andreas RAABE et Armin FELKE : A systemc language extension for high-level reconfiguration modelling. *In Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 55–60. IEEE, 2008. 40

[RHA08] Andreas RAABE, Philipp A HARTMANN et Joachim K ANLAUF : Rechannel : Describing and simulating reconfigurable hardware in systemc. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(1):15, 2008. x, 40, 41

[RRMC07] Elena Perez RAMO, Javier RESANO, Daniel MOZOS et Francky CATTHOOR : Reducing the reconfiguration overhead : a survey of techniques. *In ERSA*, pages 191–194, 2007. 42, 54, 56

[Sim] Bruno SIMÕES : General-purpose computing on the gpu (gpgpu) - http ://www.think-techie.com/2009/09/general-purpose-computing-on-gpu-gpgpu.html. ix, 25

[SLB04] Bo SHEN, Sung-Ju LEE et Sujoy BASU : Caching strategies in transcoding-

enabled proxy systems for streaming media distribution networks. *Multimedia, IEEE Transactions on*, 6(2):375–386, 2004. 23

[SNH+09] Andreas SCHALLENBERG, Wolfgang NEBEL, Andreas HERRHOLZ, Philipp A HARTMANN et Frank OPPENHEIMER : Osss+ r : A framework for application level modelling and synthesis of reconfigurable systems. *In Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 970–975. IEEE, 2009. 41

[SPC05] Kibum SUH, Seongmo PARK et Hanjin CHO : An efficient hardware architecture of intra prediction and tq/iqit module for h. 264 encoder. *ETRI journal*, 27(5): 511, 2005. 36

[WHZ+01] Dapeng WU, Yiwei Thomas HOU, Wenwu ZHU, Ya-Qin ZHANG et Jon M PEHA : Streaming video over the internet : approaches and directions. *Circuits and Systems for Video Technology, IEEE Transactions on*, 11(3):282–300, 2001. 23

[WIL09] Derek WILSON : Avivo video converter redux and ati stream quicklook. Rapport technique, ATI, 2009. 25

[WP03] Herbert WALDER et Marco PLATZNER : Online scheduling for block-partitioned reconfigurable devices. *In Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, page 10290. IEEE Computer Society, 2003. 43

[XC07] Ke XU et Chiu Sing CHOY : Low-power h. 264/avc baseline decoder for portable applications. *In Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pages 256–261. IEEE, 2007. 28

[XC08a] Ke XU et Chiu-Sing CHOY : A five-stage pipeline, 204 cycles/mb, single-port sram-based deblocking filter for h. 264/avc. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(3):363–374, 2008. 28

[XC08b] Ke XU et Chiu-Sing CHOY : A power-efficient and self-adaptive prediction en-

gine for h. 264/avc decoding. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(3):302–313, 2008. 28

[XCCP06] Ke XU, Chiu-Sing CHOY, Cheong-Fat CHAN et Kong-Pong PUN : Power-efficient vlsi implementation of bitstream parsing in h. 264/avc decoder. *In Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4–pp. IEEE, 2006. 28

[XCCP07] Ke XU, Chiu-Sing CHOY, Cheong-Fat CHAN et Kong-Pang PUN : Priority-based heading one detector in h. 264/avc decoding. *EURASIP Journal on Embedded Systems*, 2007(1):18–18, 2007. 28

[XIL96] XILINX : Xc6200 field programmable gate arrays, 1996. 2

[XLGC10] Ke XU, Tsu-Ming LIU, Jiun-In GUO et Chiu-Sing CHOY : Methods for power/throughput/area optimization of h. 264/avc decoding. *Journal of Signal Processing Systems*, 60(1):131–145, 2010. ix, 28, 29

Résumé

Malgré des avantages certains en terme d'adaptabilité et en gain de surface, la reconfiguration dynamique sur FPGA a du mal à être utilisée dans l'industrie. Le manque de moyens et de méthodes d'évaluation d'une telle architecture en est la cause majeure. Pire, aucun outil officiel ne permet aux développeurs de déterminer facilement un ordonnancement de la reconfiguration adapté pour une architecture donnée.

Cette thèse s'inscrit dans ce contexte et propose une méthodologie de modélisation SystemC d'architectures reconfigurables dynamiquement. Cet outil d'aide à la conception permet de faire gagner un temps considérable lors de la phase de conception en fournissant une première estimation des performances et des ressources nécessaires au développement de l'architecture. Il permet également le développement et la validation de scénarios d'ordonnancement de la reconfiguration, tout en respectant les contraintes temps réel liées à l'application.

Afin de valider notre modèle sur une application réelle, des IP de transcodage vidéo ont été développées et seront détaillées. Cette application consiste en la réalisation d'un transcodeur H.264/MPEG-2, rendu auto-adaptable grâce à l'utilisation de la reconfiguration dynamique.

Ces travaux ont été menés dans le cadre du projet ARDMAHN financé par l'Agence Nationale de la Recherche portant la référence ANR-09-SEGI-001.

Mots-clés: *Reconfiguration dynamique, modélisation SystemC, FPGA, transcodage vidéo, H.264, architecture auto-adaptative.*

Abstract

Despite clear benefits in terms of flexibility and surface efficiency, dynamic reconfiguration of FPGAs is still finding it hard to break through into massive industrial project. One of the main reasons is the lack of means and methods for evaluation of reconfigurable architectures. Worse, main FPGA vendors do not provide official tools allowing developers to easily determine an optimal scheduling reconfiguration for a specific architecture.

Within this framework, the proposed research work described in this thesis proposes a methodology for modeling dynamically reconfigurable architectures based on SystemC. The proposed methodology allows designers to save significant time during the design phases of an application-specific reconfigurable architecture by providing an initial estimate of performance and resources needed for its development. It also allows development and validation of scheduling reconfiguration scenarios, while respecting real-time constraints associated with the given application.

To validate our methodology on a real application, video transcoding IP have been developed and tested. This application consists in the realization of a H.264/MPEG-2 transcoder made self-adaptable through the use of dynamic reconfiguration.

This work was conducted as a part of the ARDMAHN project sponsored by the National Research Agency (Agence Nationale de Recheche) with the reference number ANR-09-SEGI-001.

Keywords: *Dynamic Reconfiguration, SystemC modeling, FPGA, video transcoding, H.264, self-adaptive architecture.*