Méthodologies de synthèse de réseaux de neurones pour applications de traitement de signal adaptatif et implémentation sur circuits reconfigurables dynamiquement

Sofien Chtourou

▶ To cite this version:

Sofien Chtourou. Méthodologies de synthèse de réseaux de neurones pour applications de traitement de signal adaptatif et implémentation sur circuits reconfigurables dynamiquement. Sciences de l'ingénieur [physics]. ENSTA ParisTech, 2007. Français. NNT : . pastel-00002942

HAL Id: pastel-00002942
https://pastel.hal.science/pastel-00002942

Submitted on 18 Sep 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée à

l'Institut National des Sciences Appliquées de Rennes

en vue de l'obtention du

Doctorat

Spécialités
Electronique(INSA)/ Ingénierie de Systèmes Informatiques(ENIS)

Préparée en cotutelle à
L'unité d'Electronique et Informatique (UEI)
de L'Ecole Nationale Supérieure des Techniques Avancées de Paris
et
Intelligent Control design and Optimisation of complex System (ICOS)
de L'Ecole Nationale d'Ingénieurs de Sfax

Par

Sofien CHTOUROU

(Ingénieur – *Génie Informatique*)

Méthodologies de synthèse de réseaux de neurones pour applications de traitement de signal adaptatif et implémentation sur circuits reconfigurables dynamiquement

soutenue le 04/06/2007, devant la commission d'examen:

M. Habib Youssef	*Rapporteur*
M. Mohamed Akil	*Rapporteur*
Mlle. Virginie Fresse	*Examinateur*
M. Dominique Houzet	*Examinateur*
M. Omar Hammami	*Examinateur*
M. Mohamed Chtourou	*Examinateur*

A mon père " Mekki " et ma mère " Souad "
Que dieu vous protège et vous prête une bonne santé
A toute ma famille
Pour leurs encouragements continus
A tous ceux qui me sont chers

Avant-propos

Les travaux présentés dans ce mémoire de thèse ont été effectués au sein de l'unité de recherche "**U**nité d'**E**lectronique et **I**nformatique (UEI)" de l'**E**cole **N**ationale **S**upérieure des **T**echniques **A**vancées de Paris (ENSTA) et de l'unité "**I**ntelligent **C**ontrol design and **O**ptimisation of complex **S**ystems (ICOS)" de l'**E**cole **N**ationale d'**I**ngénieurs de **S**fax (ENIS). Cette thèse entre dans le cadre d'une cotutelle entre l'**I**nstitut **N**ational des **S**ciences **A**ppliquées de Rennes (INSA) et l'ENIS.

Mes remerciements les plus sincères s'adressent à messieurs le président et les membres de jury, pour l'honneur qu'ils m'ont accordé en acceptant de juger mon travail.

Je voudrais exprimer ma reconnaissance à Monsieur Habib Youssef, Professeur à l'**I**nstitut **S**upérieur d'**I**nformatique et des **T**echnologies de **C**ommunication de Hammam Sousse (ISITC), d'avoir accepté de rapporter mon travail.

Je tiens à remercier Monsieur Mohamed Akil Professeur à l'**E**cole **S**upérieure d'**I**ngénieurs en **E**lectronique et **E**lectrotechnique de Paris (ESIEE) et au laboratoire **A**lgorithmique et **A**rchitecture des **S**ystèmes **I**nformatiques (A²SI), pour l'honneur qu'il m'a fait en acceptant de rapporter mon travail.

Je suis honoré par la présence de Mademoiselle Virginie Fresse, Maître de conférences à l'**I**nstitut **S**upérieur des **T**echniques **A**vancées de **S**aint-**E**tienne (ISTASE). Je tiens à lui exprimer mon profond respect.

Je tiens à remercier aussi Mr. Dominique Houzet professeur à l'**E**cole **N**ationale **S**upérieure d'**E**lectronique et de **R**adio électricité de **G**renoble (ENSERG) pour avoir dirigé efficacement mes travaux de recherche, pour son intérêt et son soutien.

Je tiens à remercier Mr. Omar Hammami enseignant chercheur à l'ENSTA pour son accueil dans l'unité de recherche UEI, pour son encadrement efficace, son soutien, sa disponibilité et ses encouragements. Qu'il trouve ici l'expression de ma profonde gratitude.

Je tiens à remercier vivement Mr. Mohamed Chtourou Maître de conférences à l'ENIS, qui n'a cessé de me prodiguer ses conseils et ses suggestions. Qu'il trouve ici l'expression de ma profonde gratitude.

Pour terminer, je tiens à remercier tous ceux qui, par leurs remarques et leurs conseils, ont contribué à la réalisation de ce travail.

I. Introduction générale ... 8
II. Méthodologies Statique et Dynamique de Conception et Synthèse de Systèmes sur Puces Reconfigurables ... 13
II.1. Introduction ... 14
II.2. Flot de conception de systèmes sur puces ... 15
II.2.1. Plateforme d'implémentation reconfigurable de type FPGA ... 15
II.2.2. Flot de conception classique de SOPCs ... 15
__II.2.2.1. Spécification haut niveau ... 17
__II.2.2.2. Partitionnement logiciel / matériel ... 18
__II.2.2.3. Synthèse de la partie matérielle ... 19
__II.2.2.4. Partie software et processeurs embarqués ... 21
II.3. Systèmes sur puces reconfigurables dynamiquement ... 21
II.3.1. Evaluation d'une architecture reconfigurable ... 21
II.3.2. Caractérisation d'une architecture reconfigurable ... 22
__II.3.2.1. La granularité des ressources de traitement ... 22
__II.3.2.2. La communication processeur - accélérateur ... 22
__II.3.2.3. La reconfigurabilité ... 23
__II.3.2.4. L'organisation mémoire ... 23
II.3.3. Reconfiguration au niveau système ... 23
II.3.4. Reconfiguration au niveau fonctionnel ... 24
II.3.5. Reconfiguration au niveau portes logiques ... 25
II.4. Méthodologies statiques de partitionnement logiciel/matériel ... 25
II.4.1. Partitionnement manuel ... 27
II.4.2. L'algorithme de partitionnement GCLP ... 27
II.4.3. L'algorithme de partitionnement MAGELLAN ... 28
II.4.4. L'algorithme de partitionnement COSYN ... 29
II.5. Méthodologies statiques de synthèse d'architectures de systèmes sur puce ... 30
II.5.1. Construction de l'architecture ... 30
__II.5.1.1. Ordonnancement ... 31
__II.5.1.2. Affectation des ressources ... 31
II.5.2. Techniques d'exploration et d'évaluation ... 31
II.5.3. Objectif d'optimisation ... 32
II.6. Méthodologies dynamiques de synthèse de systèmes sur puce dynamiquement reconfigurables ... 33
II.6.1. Construction d'une architecture reconfigurable ... 34
II.6.2. Méthodologies de gestion des configurations de l'architecture ... 35
II.7. Limitations des méthodologies statiques/dynamiques de conception de systèmes sur puce ... 36
II.8. Conclusion ... 38
II.9. Références ... 39
III. Application Adaptative et Variabilité des Charges de Travail ... 44
III.1. Introduction ... 45
III.2. Domaine d'application ... 46
III.3. Aspects variables des charges de travail ... 48
III.3.1. Variabilité dirigée par la nature des données ... 48
III.3.2. Variabilité dirigée par l'environnement d'exécution du système ... 49
III.3.3. Variabilité dirigée par la diversité des applications ... 50
III.4. Origines de la variabilité des microarchitectures des processeurs embarqués ... 51
III.4.1. Mémoire cache ... 52
III.4.2. Branchement ... 53

III.4.3. Impact sur le CPI 54
III.5. Analyse de la variabilité des charges de travail 55
III.5.1. Détection et prédiction de phases 57
III.5.2. Analyse de la variabilité 58
III.6. Conclusion 60
III.7. Références 61
IV. Méthodologie d'implémentation de systèmes sur puce reconfigurables à base d'estimateurs de performances 67
IV.1. Introduction 68
IV.2. Décomposition multi résolutions et multi composants d'un SOPC reconfigurable 69
IV.2.1. Architecture d'un SOPC reconfigurable au niveau système 69
IV.2.2. Décomposition multi résolutions d'un SOPC 70
IV.3. Méthodologie de reconfiguration multi résolutions 71
IV.3.1. Analyse multi niveaux de la variabilité 72
IV.3.1.1. Variabilité au niveau application 72
IV.3.1.2. Variabilité au niveau des modules 75
a) Architecture et algorithme d'apprentissage 75
b) Implémentation et environnement de test 76
c) Analyse de variabilité 77
IV.3.1.3. Variabilité au niveau instruction 78
a) Extraction des traces d'instructions 78
b) Caractéristiques des traces d'instructions 79
IV.3.2. Flot de conception dynamique de systèmes sur puce reconfigurables 82
IV.4. Techniques de prédiction 83
IV.4.1. Techniques de prédiction statistiques 84
IV.4.2. Techniques de prédiction à base de réseaux de neurones 85
IV.4.2.1. Réseaux de neurones et algorithme d'apprentissage 85
IV.4.2.2. Application des réseaux de neurones à la prédiction des séries temporelles 88
IV.4.3. Prétraitement 90
IV.5. Conclusion 92
IV.6. Références 93
V. Synthèse haut niveau automatique et optimisée de prédicteurs neuronaux intégrés sur puce reconfigurable dynamiquement 96
V.1. Introduction 97
V.2. Exploration des architectures SOC : motivations et objectives 98
V.3. Flot de synthèse et de conception comportementale SystemC 98
V.3.1. Flot de synthèse comportemental SystemC 98
V.3.2. Flot de conception SystemC 100
V.4. Flot d'exploration automatique surface/performance 104
V.4.1. Flot d'exploration de descriptions SystemC 104
V.4.2. Résultats d'exploration en performance/surface 105
V.5. Flot d'exploration automatique surface/performance/ consommation d'énergie 108
V.5.1. Estimation de la consommation d'énergie 108
V.5.2. Flot d'exploration en performance/surface/consommation d'énergie 109
V.5.3. Résultats d'exploration en performance/surface/ consommation d'énergie 111
V.5.3.1. Exploration de la synthèse des architectures FFT 111
V.5.3.2. Exploration de la synthèse des architectures SOM 112
V.6. Flot d'exploration des paramètres d'un réseau de neurones récurrent 114

V.6.1. Algorithme d'optimisation multi objectifs 114
__V.6.1.1. Notions et définitions 115
__V.6.1.2. Algorithme génétique multi objectifs 116
__V.6.1.3. Non dominating Sorting Genetic Algorithm (NSGA II) 117
V.6.2. Flot d'exploration des architectures de réseaux de neurones récurrents........ 118
V.6.3. Résultats d'exploration........ 119
__V.6.3.1. Exploration exhaustive mono objectif........ 119
__V.6.3.2. Optimisation mono objectif........ 122
__V.6.3.3. Optimisation multi objectifs 123
__a) Présentation d'ORINCO-DALE 124
__b) Résultats d'exploration multi objectifs........ 125
V.7. Conclusion 126
V.8. Références 128
VI. Prédiction neuronale pour la reconfiguration dynamique de systèmes sur puce........ 130
VI.1. Introduction........ 131
VI.2. Environnement d'implémentation : PEK-1.0........ 132
VI.3. architecture neuronale appliquée à la prédiction de traces d'instructions 133
VI.3.1. Application de la prédiction neuronale des adresses d'instruction........ 133
VI.3.2. modèle neuronal hybride appliqué à la prédiction 136
__VI.3.2.1. Classification incrémentale 137
__VI.3.2.2. Stratégie de prédiction neuronale hybride........ 138
__VI.3.2.3. Résultats d'implémentation........ 139
VI.4. Système reconfigurable au niveau des applications 141
VI.4.1. Chaîne de compression d'images JPEG2000 141
__VI.4.1.1. Transformée en ondelettes 142
__VI.4.1.2. Codeur entropique........ 143
VI.4.2. Architecture SystemC du codeur entropique 145
__VI.4.2.1. Conception 145
__VI.4.2.2. Implémentation en SystemC 148
VI.4.3. Codeur entropique parallèle et reconfigurable en ligne 148
__VI.4.3.1. Prédiction neuronale 150
__VI.4.3.2. Système de décision 152
__a) Approche d'ordonnancement avec des contraintes de délai local 152
__b) Approche d'ordonnancement avec des contraintes de délai global........ 156
VI.6. Conclusion 160
VI.6. Références........ 161
VII. Conclusion générale 163

Liste des tableaux

Chapitre II : Méthodologies Statique et Dynamique de Conception et Synthèse de Systèmes sur Puces Reconfigurables
Table. II. 1 : Algorithmes de partitionnement logiciel/ matériel ... 26

Chapitre IV : Méthodologie d'implémentation de systèmes sur puce reconfigurables à base d'estimateurs de performances
Table. IV. 1 : Mesures statistiques de la trace de performance du codeur entropique ... 74
Table. IV. 2 : Mesures statistiques de la trace de performance de quatre neurones ... 78
Table. IV. 3 : Caractéristiques des traces d'instructions d'exécution sur un processeur généraliste... 80
Table. IV. 4 : Caractéristiques des traces d'instructions d'exécution sur un processeur embarqué de type PowerPC ... 82

Chapitre V : Synthèse haut niveau automatique et optimisée de prédicteurs neuronaux intégrés sur puce reconfigurable dynamiquement
Table. V. 1 : Options de synthèse supportées par CSCC ... 101
Table. V. 2: Options explorées dans la figure V.7 ... 105
Table. V. 3: Options explorées dans la figure V.8 ... 106
Table. V. 4 : Architectures et caractéristiques des individus crées dans le cas d'une exploration mon objectif. 122
Table. V. 5 : Architectures et caractéristiques des individus crées dans le cas d'une exploration multi objectifs ... 126

Chapitre VI : Prédiction neuronale pour la reconfiguration dynamique de systèmes sur puce

Table. VI. 1 : Performances de prédiction en utilisant des bases de test de différentes tailles... 134
Table. VI. 2 : Résultats de prédiction en utilisant le modèle neuronal hybride... 139
Table. VI. 3 : Résultats d'apprentissage et de généralisation de coefficients non normalisés d'une base d'apprentissage de grande taille ... 140
Table. VI. 4 : Résultats d'apprentissage et de généralisation relatifs àla prédiction des séries de performances 150

Liste des figures

Chapitre I : Introduction générale

Fig. I.1 - Prédiction ITRS de la complexité des applications implémentées 9
Fig. I.2 - Prédiction ITRS en terme de méthodologies de conception........ 10

Chapitre II : Méthodologies Statique et Dynamique de Conception et Synthèse de Systèmes sur Puces Reconfigurables

Fig. II. 1 : Flot de conception SOC 16
Fig. II. 2 : Architecture et composants de la bibliothèque SystemC 19
Fig. II. 3 : Les niveaux d'abstraction du flot d'implémentation........ 20
Fig. II. 4 : Structure de l'architecture LX [39] 24
Fig. II. 5 : Partitionnement manuel : Coware........ 27
Fig. II. 6 : Organigramme de l'algorithme GCLP 28
Fig. II. 7 : Organigramme de l'algorithme MAGELLAN........ 29
Fig. II. 8 : Organigramme de l'algorithme COSYN........ 30

Chapitre III : Application Adaptative et Variabilité des Charges de Travail

Fig. III. 1 : Architecture d'un processeur PowerPC 52
Fig. III. 2 : Flot de conception au niveau système d'architectures reconfigurables [70]........ 56
Fig. III. 3 : Environnement d'analyse appliqué pour le développement et l'évaluation d'architectures reconfigurables........ 57

Chapitre IV : Méthodologie d'implémentation de systèmes sur puce reconfigurables à base d'estimateurs de performances

Fig. IV. 1: Architecture standard d'une description haut niveau d'un système sur puce reconfigurable........ 69
Fig. IV. 2 : Niveaux de granularité d'une architecture embarquée au niveau système 70
Fig. IV. 3 : Flot d'implémentation d'un système sur puce reconfigurable........ 72
Fig. IV. 4 : Variation de la performance de codage par segment d'image : (a) codage appliqué sur l'image Lena.ppm, (b) codage appliqué sur l'image Barbara.ppm 73
Fig. IV. 5 : Variation de la performance de codage par segment d'image : (a) codage appliqué sur l'image baboon.ppm, (b) codage appliqué sur l'image fruit.ppm 73
Fig. IV. 6 : Histogrammes des traces de performance du codeur entropique (a) application sur l'image 'Lena.ppm', (b) application sur l'image 'barbara.ppm', (c) application sur l'image 'baboon.ppm', (d) application sur l'image fruit.ppm........ 74
Fig. IV. 7 : Carte topologique auto adaptative de Kohonen........ 75
Fig. IV. 8 : Environnement d'exécution et de détection de performances des neurones d'un SOM........ 76
Fig. IV. 9 : Variation de la performance d'apprentissage par segment d'image et par neurone (a) performances du neurone (2, 14), (b) performances du neurone (3, 2), (c) performances du neurone (5, 16), (d) performances du neurone (12, 8)........ 77
Fig. IV. 10 : Les entités d'entrées/sorties de l'outil PIN-2.0........ 79
Fig. IV. 11 : Traces d'adresses des instructions (a) trace d'exécution de l'application LS, (b) trace d'exécution de l'application CP, (c) trace d'exécution de l'application GZIP 80
Fig. IV. 12 : Histogrammes des traces d'adresses d'instructions : (a) application sur le programme LS, (b) application sur le programme CP, (c) application sur le programme GZIP 81
Fig. IV. 13 : Traces d'adresses des instructions (a) trace d'exécution de l'application FFT, (b) trace d'exécution de l'application CRC........ 82
Fig. IV. 14 : Flot de conception d'un système sur puce reconfigurable........ 83
Fig. IV. 15 : Neurone formel........ 86

Fig. IV. 16 : Différents types de fonctions de transfert : a) fonction à seuil, b) fonction linéaire, c) fonction sigmoïde 87
Fig. IV. 17 : Classification d'architectures de réseaux de neurones........ 87
Fig. IV. 18 : Réseau de neurones récurrent de type NARX 89
Fig. IV. 19 : Structure décomposée d'un réseau NARX (n=3) 89

Chapitre V : Synthèse haut niveau automatique et optimisée de prédicteurs neuronaux intégrés sur puce reconfigurable dynamiquement

Fig. V. 1 : Flot de synthèse comportementale........ 99
Fig. V. 2: Flot de conception à partir d'un niveau de description comportementale........ 101
Fig. V. 3 : Effet de la compilation en exécution spéculative sur un code SystemC 102
Fig. V. 4 : Effet de la période sur l'architecture synthétisée : (a) synthèse en fixant la période à 30 ns, (b) synthèse en fixant la période à 22ns. 103
Fig. V. 5 : Flot d'exploration automatique à deux dimensions (surface, performance) des architectures synthétisées au niveau SystemC comportemental........ 104
Fig. V. 6: Scripte de synthèse supporté par CSCC........ 105
Fig. V. 7: Exploration des option d'ordonnancement sur la synthèse du FFT: (G1) : effet de l'option extend_latency, (G2) : effet de l'option effort_level........ 106
Fig. V. 8 : Exploration des option d'ordonnancement sur la synthèse du FIR: (G1) : effet de l'option extend_latency, (G2) : effet de l'option effort_level........ 107
Fig. V. 9 : Effet de l'option permettant l'exécution spéculative sur les résultats de synthèse 108
Fig. V. 10 : Flot d'estimation de la consommation d'énergie en utilisant 'power compiler' 109
Fig. V. 11 : Flot d'exploration automatique à trois dimensions (surface, performance, consommation d'énergie) des architectures synthétisées au niveau SystemC comportemental 110
Fig. V. 12 : Scripte de simulation par Modelsim 110
Fig. V. 13 : Exploration à trois dimensions de la synthèse du FFT : (a) cas speculative_execution=false et extend_latency=false, (b) cas speculative_execution=false et extend_latency=true, (c) cas speculative_execution=true et extend_latency=false, (d) cas speculative_execution=true et extend_latency=true 111
Fig. V. 14: Architecture implémentée effectuant l'apprentissage et le test d'un réseau SOM........ 112
Fig. V. 15 : Exploration à trois dimensions des architectures synthétisées du SOM : (a) cas speculative_execution=false et extend_latency=false, (b) cas speculative_execution=true et extend_latency=false, (c) cas speculative_execution=false et extend_latency=false, (d) cas speculative_execution=false et extend_latency=true 113
Fig. V. 16 : Algorithme génétique 116
Fig. V. 17 : Algorithme NSGAII 118
Fig. V. 18 : Flot d'exploration d'architectures de réseaux de neurones de prédiction 118
Fig. V. 19 : Variation de l'erreur d'apprentissage en fonction des paramètres du réseau de neurones de prédiction : (a) prédiction en utilisant un réseau de neurones à deux neurones d'entrée, (b) prédiction en utilisant un réseau de neurones à quatre neurones d'entrée, (c) prédiction en utilisant un réseau de neurones à six neurones d'entrée, (d) prédiction en utilisant un réseau de neurones à onze neurones d'entrée........ 121
Fig. V. 20 : Exploration des architectures neuronaux en optimisant l'erreur d'apprentissage........ 123
Fig. V. 21 : Flot de conception d'ORINOCO-DALE [8]........ 124
Fig. V. 22 : Evaluation des fonctions objectifs des architectures neuronales explorées: (a) évaluation de l'erreur d'apprentissage, (b) évaluation de la consommation d'énergie........ 125

Chapitre VI : Prédiction neuronale pour la reconfiguration dynamique de systèmes sur puce

Fig. VI. 1 : Diagramme d'état de l'algorithme d'apprentissage et de test du système de prédiction neuronale implémenté........ 134
Fig. VI. 2 : Apprentissage et test de généralisation d'un réseau de neurones récurrents appliqué à la prédiction: (a) résultat d'apprentissage, (b) résultat de généralisation........ 135
Fig. VI. 3 : Modèle neuronal hybride appliqué à la prédiction 137

Fig. VI. 4 : Diagramme d'état du modèle neuronal hybride appliqué à la prédiction 139
Fig. VI. 5 : Résultats d'apprentissage de la série LS en adoptant un SOM à une, trois, 6 et 10 sorties (classes) 140
Fig. VI. 6 : Chaîne de compression JPEG2000 142
Fig. VI. 7 : Effet de la transformée en ondelettes sur une image 143
Fig. VI. 8 : Représentation en bit-plan d'un code-block 144
Fig. VI. 9 : Composition d'un 'stripe étendu' 145
Fig. VI. 10 : Traitement de 'Coefficient Bit Modeling' 146
Fig. VI. 11 : Pipeline préliminaire 146
Fig. VI. 12 : Pipeline respectant la dépendance des données 147
Fig. VI. 13 : Diagramme d'état des différentes passes 147
Fig. VI. 14 : Architecture du module 'Coefficient Bit Modeling ' 148
Fig. VI. 15 : Architecture parallèle et reconfigurable de codage entropique 149
Fig. VI. 16 : Diagramme d'état de l'architecture reconfiguarble du codeur entropique parallèle. 149
Fig. VI. 17 : Résultats de prédiction des performances de codage d'une partie des code-blocs de 'Lena.ppm' (apprentissage) 150
Fig. VI. 18: Résultats de prédiction des performances de codage d'une partiedes code-blocs de 'baboon.ppm' (généralisation) 151
Fig. VI. 19 : Résultats de prédiction des performances de codage d'une partie des code-blocs de 'Barabara.ppm' (généralisation) 151
Fig. VI. 20 : Résultat de prédiction des performances de codage des code-blocs de 'fruit.ppm' 152
Fig. VI. 21 : Effet de l'ordonnancement selon l'approche locale : (a) distribution initiale des code-blocs, (b) distribution adoptée par le système de décision 153
Fig. VI. 22 : Nombre de codeurs par époque de reconfiguration 153
Fig. VI. 23 : Taux d'occupation des différents codeurs : (a) codeur P1, (b) codeur P2, (c) codeur P3, (d) codeur P4, (e) codeur P5. 155
Fig. VI. 24 : Effet de l'ordonnancement selon l'approche globale : (a) distribution initiale des code-blocs, (b) distribution adoptée par le système de décision 156
Fig. VI. 25 : Nombre de codeurs par époque de reconfiguration 157
Fig. VI. 26 : Taux d'occupation selon l'approche globale des différents codeurs : (a) codeur P1, (b) codeur P2, (c) codeur P3, (d) codeur P4, (e) codeur P5 158

I. Introduction générale

Les progrès dans les techniques de conception et dans la technologie des semi-conducteurs ont permis l'intégration de systèmes embarqués de complexité croissante sur une puce. Initialement complètement dédiés, ces systèmes incluront comme prédit par l'ITRS[1] (voir Fig. I.1) un nombre de plus en plus croissant de composants programmables constitués de processeurs associés à des unités spécialisées de traitement.

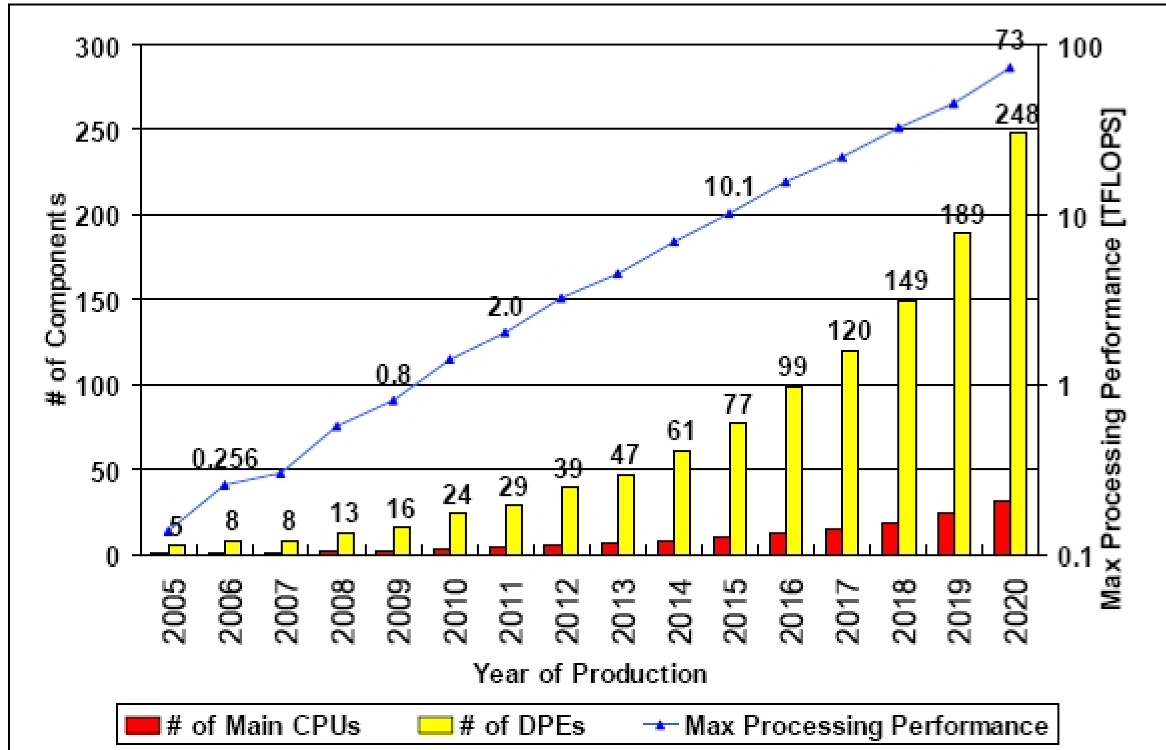


Fig. I.1 - Prédiction ITRS de la complexité des applications implémentées

Ces projections ne pourront se réaliser que si des méthodologies de conception et des outils logiciels permettent de générer de tels systèmes dans des délais et des coûts en accord avec les contraintes fortes de mise sur marché TTM (Time To Market) et de coûts propres à l'électronique grand public (consumer electronics). Les piliers de la stratégie adoptée pour atteindre cet objectif sont les suivants:

1. la réutilisation de composants,
2. l'utilisation de plateformes,
3. l'abstraction.

La réutilisation des composants a été encouragée par la création et la mise à disposition d'un nombre croissant, ces dernières années, de propriétés intellectuelles (IP[2]) de nature diverse (processeurs, accélérateurs matériels, bus, périphériques) respectant des standards d'interfaçage garantissant leur intégration rapide. Cette réutilisation a permis de constater que pour certains domaines, des composants étaient communément réutilisés donnant naissance au concept de plateforme. La plateforme représente un point de départ significatif en conception de systèmes permettant la focalisation sur la valeur ajoutée du système. Seules les IPs novatrices ou créant une valeur ajoutée certaine feront l'objet d'une conception dédiée. Les deux concepts précédents sont associés au troisième celui de l'abstraction permettant de

[1] ITRS : International Technology Roadmap for Semiconductors
[2] IP: Intellectual Property

réduire cette fois-ci le temps de conception par une réduction des détails. Il est ainsi possible d'utiliser des plateformes à différents niveaux d'abstraction basés sur des IPs d'abstraction mixte. L'ensemble de ces techniques permet de concevoir des systèmes complets pouvant répondre aux besoins d'applications complexes et déterministes. Cette situation change si les applications visées sont diverses dans leur comportement à l'exécution en termes d'utilisation de ressources et si de plus chaque application elle même présente un caractère variable à l'exécution. Cette variabilité de la charge de travail va à l'encontre des méthodologies actuelles qui considèrent que toutes les informations relatives aux applications sont connues de manière statique et que donc toutes les décisions de partitionnement logiciel-matériel, et d'allocation de ressources ainsi que d'ordonnancement le sont aussi. Ces hypothèses de conception ne peuvent pas s'appliquer à des applications de type multimédia portables comme la compression d'images dont les temps d'exécution et les ressources nécessaires dépendent de l'objet à compresser. Un des axes possibles et intuitifs est l'adaptation des ressources à cette variabilité par la reconfiguration dynamique. L'ITRS projette d'ici 2013 que 42% des fonctionnalités des SOC seront reconfigurables.

	Année de production	**2005**	**2006**	**2007**	**2008**	**2009**	**2010**	**2011**	**2012**	**2013**
	DRAM (nm)	80	70	65	57	50	45	40	36	32
	Design Reuse									
was	% surface total	32%	33%	35%	36%	38%	40%	41%	42%	44%
is	% surface total	32%	33%	35%	36%	38%	40%	41%	42%	44%
	Platform based design									
was	Plateformes	96%	88%	83%	83%	75%	67%	60%	55%	50%
is	Plateformes	100%	91%	87%	83%	75%	70%	60%	55%	52%
was	% de plateformes supportées par les outils	3%	6%	10%	25%	35%	50%	57%	64%	75%
is	% de plateformes supportées par les outils	3%	6%	10%	25%	35%	50%	57%	64%	75%
	Synthèse haut niveau									
	% estimation (surface, performance, ...) via valeurs réelles	53%	56%	60%	63%	66%	70%	73%	76%	80%
	reconfigurabilité									
was	% du SOC reconfigurable	23%	26%	28%	28%	30%	35%	38%	40%	42%
is	% du SOC reconfigurable	23%	26%	28%	28%	30%	35%	38%	40%	42%

Fig. I.2 - Prédiction ITRS en termes de méthodologies de conception

Cette reconfigurabilité doit être interprétée comme une propriété nécessaire des systèmes face à des charges variables. Cette distinction est essentielle pour séparer cette problématique des circuits reconfigurables initialement proposés pour le prototypage rapide ou de la reconfigurabilité exploitée pour faire face à des contraintes de surface toutes pouvant être traitées par des flots de conception classiques ou légèrement modifiés. Ici, le flot a pour objectif d'aider le concepteur à mettre dans son système les éléments matériels et logiciels pouvant répondre à ce besoin en adaptabilité.

L'objectif de cette thèse est de proposer un modèle de conception d'architectures reconfigurables dynamiquement, opérant en ligne au niveau système et permettant d'offrir

une meilleure densité de calcul par rapport aux solutions statiques existantes. De nombreuses questions sont soulevées dans ce cadre :

- **variabilité de charge:** comment définir la variabilité de charge ? comment identifier et analyser cette variabilité de charge et suivant quelles métriques ? où doit-on détecter cette variabilité de charge dans un système complexe? cette variabilité est elle éliminable par transformation de l'application ?
- **contrôle de la variabilité :** comment prédire cette variabilité ? comment traiter les variabilités multiples au sein d'un système et de quelle manière les corréler ? quel modèle de contrôle est applicable et pour quel type de système et pour quelle variabilité ?
- **implémentation du couple analyse de charge et contrôle de variabilité:** comment implémenter le couple analyse et contrôle de la variabilité et sous quelles contraintes? dans quelles conditions l'implémentation rend inefficace l'exploitation de cette variabilité ? quels sont les coûts en surface et consommation d'énergie de cette implémentation ?

La démarche scientifique adoptée dans cette thèse suit cette structuration même s'il est important de souligner que ces travaux trouvent leur origine dans un besoin issu suite à l'implémentation du codeur entropique de la chaîne de compression d'images fixes JPEG2000[3] et donc qui ont émergé de besoins réels.

Après une analyse de l'état de l'art sur les méthodologies de conception SOC, nous allons présenter quelques travaux de recherche visant l'introduction du caractère adaptatif dans un système sur puce et nous énumérons les limitations des méthodologies de reconfiguration dynamique présentées.

Dans le deuxième chapitre de cette thèse, nous allons nous focaliser sur les aspects variables figurant dans un système sur puce. En premier lieu, nous allons extraire les différentes caractéristiques qui favorisent l'introduction d'un aspect adaptatif dans l'architecture d'une application. En deuxième lieu, nous allons analyser les causes de variabilité en cours d'exécution des charges de travail de l'application. Nous exposerons les différents aspects reconfigurables d'une architecture embarquée. Finalement, nous allons présenter quelques travaux illustrant l'exploitation de la notion de variabilité des performances d'une application.

Dans le troisième chapitre de cette thèse, nous nous sommes intéressés à l'étude de la variabilité des performances des composants matériels et logiciels. Dans ce contexte, nous avons détecté les principales causes de variabilité. Des exemples d'applications à performances variables sont exposés. Le problème de la prédiction de grandes séries de temps résultant de la capture de la variabilité de charge est abordé et les outils pouvant être utilisés dans cet objectif sont cités. Nous présentons les réseaux de neurones récurrents connus comme des approximateurs universels capables de modéliser un phénomène dynamique non linéaire. Le choix des réseaux de neurones est justifié par leur robustesse en termes de modélisation des séries temporelles non linéaires.

Le quatrième chapitre de la thèse présente deux méthodologies d'exploration d'architectures de réseaux de neurones synthétisées à partir d'un haut niveau de description. La première méthodologie est indépendante de l'architecture du réseau de neurones. Elle effectue l'exploration des différentes architectures synthétisées à partir d'une description SystemC quelconque. La deuxième méthodologie explore l'effet de l'architecture d'un réseau de

[3] JPEG : Joint Photographic Experts Group

neurones sur les résultats d'implémentation. Un algorithme d'optimisation multi objectifs de type NSGAII[4] a été utilisé pour l'optimisation des architectures synthétisées.

Finalement, nous allons introduire le flot de conception de systèmes adaptatifs avec charge variable. La détection de besoins de reconfiguration est effectuée en ligne en se basant sur les estimations de performances effectuées par le réseau de neurones. Dans ce chapitre, nous allons présenter les résultats d'apprentissage d'un réseau de neurones récurrent appliqué à la prédiction des métriques décrivant les performances des différents composants appartenant à différents niveaux de granularité. L'environnement reconfigurable implémenté au niveau système associé aux différents résultats de reconfiguration sont aussi exposés.

[4] NSGA : Non dominating Sorting Genetic Algorithm

II. Méthodologies Statique et Dynamique de Conception et Synthèse de Systèmes sur Puces Reconfigurables

II.1. Introduction

Un système sur puce programmable (SOPC[5]) est défini dans [1] comme un système complexe et autonome sur une seule puce contenant aux moins un processeur. Plusieurs périphériques externes viennent compléter le fonctionnement du système. En d'autres termes, un SOPC est un système complet embarqué sur une puce, pouvant comprendre différents types de mémoires, un ou plusieurs processeurs ainsi que des périphériques d'interfaces. L'intégration de tous ces composants est rendue possible grâce à la miniaturisation croissante des composants. Un SOPC peut être vu selon plusieurs niveaux : le niveau système, qui décrit le fonctionnement global du SOPC au niveau d'abstraction le plus haut (niveau simulation), et le niveau implémentation, qui décrit le SOC sur une plateforme électronique. Les applications implémentées sur des SOPCs exécutent généralement des traitements temps réel pour lesquels le facteur temps représente une principale contrainte à respecter. Les traitements effectués par l'application doivent être juste et sans délai afin de respecter les contraintes de la qualité de service (QOS[6]) à assurer par l'application. Les applications temps réel interviennent dans plusieurs domaines notamment dans les télécommunications [2], la robotique et l'aéronautique.

Le flot de conception et d'implémentation d'un SOPC comporte plusieurs étapes ou niveaux d'abstraction tenant compte de la complexité croissante des applications à implémenter. Les travaux de recherche menés dans ce domaine se focalisent principalement sur deux axes :

- L'adaptation dynamique du design, qui représente l'architecture de l'application à un niveau haut ou bas de description, vis à vis de l'application à implémenter ce qui est connu dans la littérature par la reconfiguration dynamique des systèmes sur puces. En effet, l'apparition des composants ayant la logique dynamiquement reconfigurable tels que les composants FPGA (Field Programmable Gate Array) associée à la complexité des designs à implémenter sur des SOPC, tels que les applications multimédias et de télécommunications, imposent de nouveaux défis à la communauté qui vise l'introduction d'un caractère réactif et adaptatif dans le flot de conception d'un SOPC.
- L'automatisation du processus de conception tout en assurant l'adéquation entre la spécification et l'architecture générée de l'application représente le deuxième axe de recherche. Le passage entre algorithme et architecture est assuré par des algorithmes d'exploration d'un espace d'architectures et de synthèse automatique qui génère l'architecture adéquate du système. Cette étape de synthèse architecturale est composée de deux parties. Une première partie qui s'intéresse au développement de techniques d'exploration afin d'optimiser les performances du système implémenté. Les aspects architecturaux d'exploration analysent essentiellement le partitionnement logiciel/ matériel et l'ordonnancement des tâches de l'application. Dans la deuxième partie, les chercheurs focalisent sur la définition de méthodes et de métriques d'estimation de performances d'architectures décrites à différents niveaux d'abstraction.

Dans ce chapitre, nous allons exposer les travaux de recherche menés pour définir un cadre méthodologique pour la conception de systèmes sur puces reconfigurables. Nous allons commencer par la présentation des différentes étapes du flot de conception standard d'une architecture SOPC statique. Ensuite, nous allons donner un bref aperçu sur les caractéristiques des architectures dynamiquement reconfigurables. Dans la quatrième et la cinquième section,

[5] SOPC : System On Programmable Chip
[6] QOS : Quality Of Service

nous allons exposer quelques techniques de synthèse automatique d'architectures statiques traitant en particulier les problèmes de partitionnement logiciel/matériel et d'ordonnancement. Ensuite, nous allons présenter quelques méthodologies de reconfiguration dynamique. Pour conclure, nous allons énumérer les limites des méthodologies de conception de systèmes reconfigurables dynamiquement.

II.2. Flot de conception de systèmes sur puces

Un SOPC incorpore un ou plusieurs processeurs, de la mémoire sur puce et des blocs logiques qui représentent des accélérateurs matériels. L'approche de conception des systèmes embarqués hétérogène (logiciels/matériels), souvent appelée co-design, comporte des parties de conception matérielle et logicielle. La conception d'un SOPC combine alors des composants logiciels (SW) implémentés sur les processeurs de la plateforme et des composants matériels (HW) intégrés sur la partie logique. L'approche de développement SOPC sur FPGA présente deux avantages par rapport à l'approche SOC sur ASIC[7]. En premier lieu, une plateforme SOPC présente un développement associé à un prototypage rapide de l'application embarquée. En deuxième lieu, dans une plateforme SOPC l'application dispose de ressources reconfigurables au cours de l'exécution de l'application.

La conception d'un système sur puce exige généralement un important travail de conception et une grande expertise pour fixer l'architecture adéquate (au niveau des deux cotés matériel et logiciel), choisir les interfaces et gérer les périphériques de la plateforme. La diversité au niveau des choix de la conception du système complet offre un large espace d'architectures à explorer par le concepteur. Plusieurs travaux ont été ainsi proposés pour fixer un flot de conception standard pour l'implémentation d'un SOPC qui supporte les nouveaux challenges de conceptions notamment la reconfiguration dynamique [3].

II.2.1. Plateforme d'implémentation reconfigurable de type FPGA

L'apparition des premiers circuits reconfigurables remonte aux années 1980 (le premier FPGA le XC2000 est introduit par Xilinx [5] en 1984) sous la forme de circuits logiques programmables simples. Les circuits les plus récents de type FPGA offrent une surface d'implémentation matérielle importante pouvant s'étendre sur des millions de portes logiques programmables et s'exécutant à des fréquences de fonctionnement pouvant atteindre les 420MHZ. Une plateforme FPGA est constituée d'un ensemble de blocs logiques qui peuvent être interconnectés par programmation pour réaliser des circuits différents. Un bloc logique dans ce cas peut être aussi simple qu'un transistor ou aussi complexe qu'un microprocesseur. L'interconnexion des blocs logiques est assurée par une architecture de routage selon laquelle les segments de raccordement sont positionnés pour permettre le câblage des différentes unités logiques. On distingue plusieurs architectures de FPGA proposées par un nombre de fabricants de composants tels que Actel [4], Xilinx [5] et Altera [6]. Les composants FPGA comportent de plus en plus de portes logiques, de composants (blocs mémoire, processeurs) tout en optimisant la fréquence de fonctionnement. A titre d'exemple, le composant VirtexII [7] de Xilinx comporte 8M de portes logiques et pouvant atteindre une fréquence de fonctionnement de 420MHz, des blocs de mémoire à porte double qui varient entre 4 et 168 blocs.

II.2.2. Flot de conception classique de SOPCs

Un flot de conception d'un système sur puce regroupe plusieurs niveaux d'abstraction. Dans chaque niveau, le concepteur s'intéresse à la résolution d'un problème lié à la spécification de

[7] ASIC : Application Specific Integrated Circuit

l'application, la définition de l'architecture ou à l'implémentation du design sur circuit spécifique. Les méthodologies de conception SOPC peuvent être divisées en trois familles : conception dirigée par les contraintes temporelles (TDD[8]), la conception basée sur des blocs fonctionnels (BBD[9]), la conception basée sur des plateformes déjà implémentées (PBD[10]). La méthodologie TDD représente une méthodologie itérative d'optimisation de la performance du design. La méthodologie de conception BDD est la plus utilisée dans la littérature. Idéalement, en BBD la modélisation comportementale est réalisée au niveau système, où on effectue les compensations HW/SW et la Co-vérification fonctionnelle du système hétérogène en utilisant la simulation SW et/ou l'émulation du HW. Les nouveaux composants conçus sont alors partagés et implémentés sur les blocs fonctionnels RTL[11] spécifiés. Ces composants doivent respecter les contraintes de performances, de puissance, et de surface. Comme la BBD, la méthodologie de conception PBD est une méthodologie de conception hiérarchique qui commence au niveau système. Là où la PBD diffère de la BBD est qu'elle atteint le maximum de sa productivité à travers une extension planifiée du concept de la réutilisation de composants.

Le flot de conception SOPC classique représenté par la figure II.1, et qui représente une méthodologie de type BBD, admet deux entrées sous la forme d'une spécification fonctionnelle de l'application et d'une bibliothèque de propriétés intellectuelles (Intellectual Propriety (IP)).

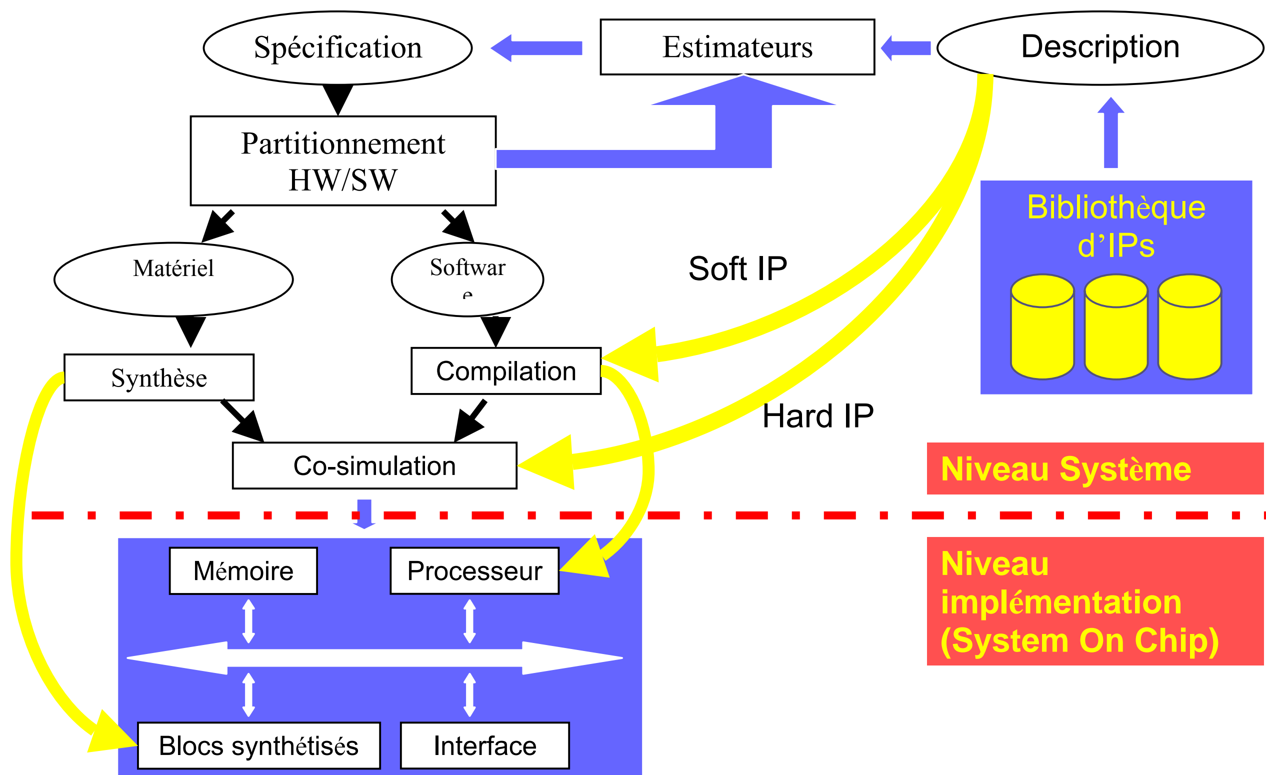


Fig. II. 1 : Flot de conception SOC

Le flot de conception des SOPCs de la figure II. 1 regroupe essentiellement trois niveaux d'abstraction :

[8] TDD : Time Driven Delais
[9] BBD : Blocs Based Design
[10] PBD : Platform-Based Design
[11] RTL : Register Transfer Level

1. En partant d'une spécification fonctionnelle complète du design et en disposant d'une bibliothèque d'IPs logicielles et matérielles, le concepteur partitionne l'application en une partie logicielle et une autre matérielle. L'optimisation des techniques de partitionnement constitue un sujet de recherche très sollicité par les scientifiques [13, 42, 43]. Ces activités de recherche s'intéressent surtout à l'automatisation [8] de cette étape.
2. Une fois le partitionnement effectué, on implémente la partie matérielle en utilisant un langage de description matériel tel que VHDL, VERILOG ou SystemC. La synthèse de la partie matérielle est réalisée par des outils de conception assistée par ordinateur (CAO[12]) tels que Design Compiler de Synopsys [9], Allegro de Cadence [10] et Agility Compiler de Celoxica [11]. La partie logicielle en revanche, sera implémentée en C/C++. La compilation de la partie logicielle doit tenir compte du type du processeur cible. Ensuite, on passe à la co-simulation du système complet en utilisant des composants (IP) matériels et logiciels déjà existants dans la bibliothèque d'IPs de l'outil de simulation. Les tâches déjà expliquées représentent le niveau système du flot de conception des SOPCs.
3. Dans la phase finale du flot, on injecte le SOPC sur une plateforme adaptée de type FPGA. Cette plateforme représente un environnement idéal pour l'implémentation des SOPCs contenant un ou plusieurs processeurs qui supportent la partie logicielle, une surface de portes logiques (pour supporter la partie matérielle et les bus de communication), des blocs mémoires et des interfaces de communication.

La complexité du processus de conception d'un système embarqué a nécessité la définition de plusieurs niveaux d'abstraction. Le concepteur s'intéresse à chaque niveau à définir et à vérifier un aspect fonctionnel ou architectural du design. On distingue le niveau comportemental et le niveau des registres (RTL). La conception au niveau comportemental s'intéresse à l'étude des fonctionnalités du système, la définition des tâches concurrentes et indépendantes de l'application et finalement le partitionnement HW/SW du design. La conception au niveau RTL consiste à modéliser l'architecture de l'application en tenant compte des contraintes d'implémentation sur le matériel. Dans les paragraphes qui suivent, nous allons détailler les différentes étapes du flot de conception comme présenté dans la figure II.1.

II.2.2.1. Spécification haut niveau

La spécification d'un système à implémenter au niveau fonctionnel fixe les fonctions principales et dérivées exigées par l'utilisateur. En spécifiant le système, le concepteur doit tenir compte des exigences de performances techniques (performance du traitement) et économiques (coût d'implémentation) du système sur puce à implémenter. L'objectif de l'étape de spécification est de fixer un modèle fonctionnel et de le tester tout en vérifiant les contraintes d'implémentation de l'application. La création d'un modèle fonctionnel s'intéresse à l'organisation ou la structuration du comportement de l'architecture au cours du temps. La spécification de l'application dépend des contraintes imposées par la plateforme d'implémentation.

Les concepteurs de systèmes embarqués disposent d'une large variété de modèles de spécification fonctionnelle tels que Statechart, les réseaux de pétri et UML (Unified Modeling Language). Ces modèles de spécification, qui sont capables de représenter les fonctionnalités d'un système, restent inadaptés pour exprimer les contraintes non fonctionnelles de l'application telles que les performances temporelles (durée, latence et débit) et les ressources

[12] CAO : Conception Assisté par Ordinateur

d'implémentation nécessaires (énergie, mémoire et surface d'implémentation). La vérification de l'intégrité fonctionnelle de l'application nécessite l'utilisation d'un langage de description haut niveau associé à un noyau de simulation comme SystemC [12].

II.2.2.2. Partitionnement logiciel / matériel

L'étape de partitionnement logiciel/matériel [13] détermine les tâches qui vont être implémentées sur un ou plusieurs processeurs et les tâches de traitement effectuées par des accélérateurs matériels. Durant cette étape de partitionnement, le concepteur fixe également les interfaces entre la partie HW et SW. L'étape de partitionnement est une étape déterminante dans le processus de conception d'un SOPC car elle fixe l'architecture finale du système ainsi que ses performances. Plusieurs algorithmes de partitionnement ont été proposés pour automatiser cette étape. Le défit majeur des activités de recherche est de proposer une méthodologie standard de conception SOPC conjointe, automatique et flexible. Dans le flot présenté dans la figure II.1, l'étape de partitionnement se base sur des estimations de performances de chaque partie du design. L'étape d'estimation déploie des blocs fonctionnels d'IPs complexes pouvant être réutilisés dans plusieurs designs. Ces blocs représentent des IPs matériels qui sont des implémentations physiques dépendant d'une technologie très optimisée et des IP logiciels décrits en langage de description haut niveau comme VHDL [13] ou SystemC souvent paramétrables et synthétisable. L'étape de partitionnement est effectuée généralement dans un environnement de développement de co-design, co-simulation et co-vérification adapté.

L'étape de partitionnement automatique d'une spécification est un problème complexe (un problème NP-complet). Afin de décomposer ce problème, on peut subdiviser ce processus en trois parties principales :

- Une partie qui effectue l'allocation en fixant le type et le nombre nécessaire de ressources matérielles et logicielles. Dans le cas d'un processus d'allocation statique, le concepteur dimensionne à un niveau d'abstraction très haut l'architecture globale de l'application embarquée. Le concepteur fixe ainsi la limite de performance du design qui dépend des composants déployés dans l'architecture.
- Une partie qui effectue le partitionnement spatial ou temporel en affectant les tâches qui constituent l'application sur la partie matérielle ou logicielle. Cette partie se limite souvent à un problème de partitionnement spatial dans lequel on affecte les différentes tâches aux différents composants fonctionnels. Cependant, dans le cas des architectures reconfigurables, une nouvelle notion s'intercale : c'est la notion temps. Pour résoudre le problème de partitionnement dynamique, le concepteur doit effectuer un partitionnement temporel en agençant les tâches matérielles reconfigurables dans des segments temporels différents. Dans cette étape, le concepteur définit les délais d'occupation de chaque composant et favorise par la suite le principe de réutilisation des composants.
- Une partie qui effectue l'ordonnancement de l'exécution et de la reconfiguration des différentes tâches ainsi que la communication entre elles. A ce niveau, le concepteur doit explorer l'espace de solution afin de sélectionner une architecture qui respecte les contraintes de traitement temps réel imposées. La complexité de la tâche d'ordonnancement augmente en tenant compte de l'aspect de reconfiguration dynamique du design.

[13] VHDL : Very High Speed Integrated Circuit Hardware Description Language

Généralement, l'étape de partitionnement logiciel / matériel est effectuée manuellement en utilisant un outil adapté comme l'environnement proposé par Coware [14] ou d'une manière automatique et statique. L'aspect de partitionnement automatique et dynamique est rarement traité. Dans le paragraphe II. 3, nous allons détailler les méthodologies de partitionnement automatique et statique proposées dans la littérature. Les algorithmes de partitionnement, définis dans la littérature, produisent automatiquement la partie logicielle ainsi que les accélérateurs matériels associés.

II.2.2.3. Synthèse de la partie matérielle

La complexité croissante de l'étape de synthèse des IPs matériels favorise la mise en œuvre d'un chemin plus directe entre la spécification et l'implémentation de l'application. En effet, plusieurs initiatives se sont intéressées à l'automatisation du passage de la spécification vers l'implémentation en proposant des flots de conception adaptés et des outils de CAO associés. Ces outils aident le concepteur à maîtriser la complexité de l'application ce qui implique une réduction du temps de conception. Ceci nous ramène à résoudre un problème d'adéquation algorithme – architecture qui se résume à trouver la meilleure correspondance entre une description haut niveau et une architecture à implémenter en satisfaisant les contraintes de traitement temps réel, de surface et de faible consommation d'énergie.

L'une des initiatives les plus marquantes pour la description matérielle des systèmes embarqués est SystemC [15]. SystemC est un langage de description d'architecture matérielle comme VHDL ou Verilog. Il est souvent classé comme un langage de description système puisqu'il est généralement utilisé pour décrire le comportement du design. SystemC est une bibliothèque de composants codée en C++ permettant la description matérielle d'une application. La figure II.2 décrit les différents composants de la bibliothèque SystemC [12]. La bibliothèque SystemC regroupe un ensemble de classes et un noyau de simulation permettant la conception et la simulation de la partie matérielle d'un système embarqué. Ces classes permettent la modélisation en respectant plusieurs contraintes telles que la notion temps, la notion de concurrence entre les tâches avec la possibilité de fixer des structures hiérarchiques décrivant l'architecture du design. SystemC propose aussi un ensemble de types de données (bits et vecteurs de bits) et des structures de communication (port, signaux) permettant la description d'architectures matérielles.

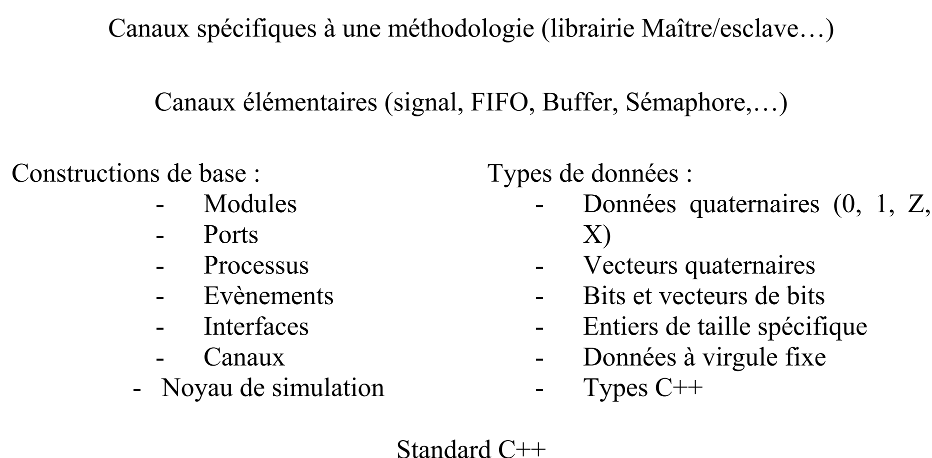


Fig. II. 2 : Architecture et composants de la bibliothèque SystemC

Les concepteurs de systèmes sur puces utilisent SystemC pour décrire les architectures matérielles au niveau système ou RTL (Fig. II.3). La contribution majeure de l'approche

SystemC est la possibilité de concevoir, synthétiser et vérifier des architectures et les interfaces de communication matérielles à partir d'un niveau haut de description [16-18].

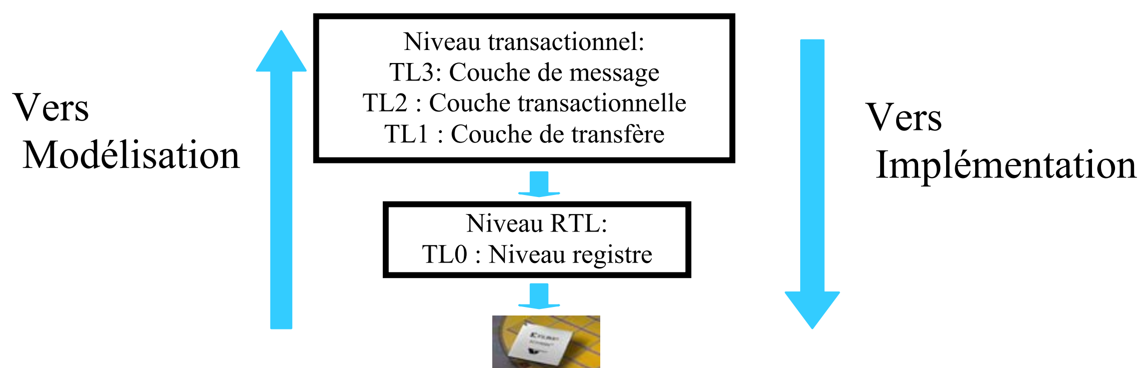


Fig. II. 3 : Les niveaux d'abstraction du flot d'implémentation

Dans la figure II.3, on distingue deux niveaux d'implémentation qui sont : le niveau transactionnel ou système et le niveau RTL qui est le niveau de départ d'un flot d'implémentation matérielle. Dans le niveau système, on distingue trois sous niveaux d'abstraction :

- Le niveau 'Couche de message' représente un modèle fonctionnel du design. A ce niveau d'implémentation, on analyse la fonctionnalité du système en faisant abstraction de la communication telle que les bus et les mémoires partagées, des contraintes matérielles d'implémentation et de la notion de temps.
- Le niveau 'Couche transactionnelle' appelé aussi niveau TLM[14] [19]. A ce niveau, on s'intéresse principalement à la modélisation de la communication entre les différents modules du design à l'aide d'appels de fonctions qui caractérisent les transactions. L'avantage de la modélisation au niveau TLM est la rapidité de simulation des architectures à ce niveau système.
- Le niveau 'Couche de transfère' est un niveau qui respecte la notion de temps (Cycle accurate). Dans ce niveau, on peut analyser les détails d'implémentation avant de passer au niveau RTL. Le partitionnement HW/SW est permis à partir de ce niveau. Plusieurs outils industriels de CAO ont été développés pour la conception et le test de systèmes sur puce : PEK [20] d'IBM et Cocentric System Studio [21] de Synopsys. Ce dernier propose une grande base d'IPs matériels.

L'apparition de SystemC a permis la description des designs dans des différents niveaux système en gardant la possibilité de décrire des systèmes HW/SW. SystemC a favorisé aussi l'introduction des méthodologies de conception logicielles, notamment la conception orientée objet, dans le flot de conception des SOPCs [22], [23], [24]. D'autres travaux ont exploité la rapidité de simulation SystemC au niveau TLM pour proposer des méthodologies et des environnements d'exploration d'architectures au niveau système [25], [26], [27]. Les méthodologies appliquées sur des architectures SystemC au niveau TLM ont essentiellement traité le problème de partitionnement HW/SW [28], d'estimation de la consommation d'énergie [29], de la surface d'implémentation [30] et de la détection de parallélisme [31] au niveau système. L'utilisation de SystemC ne se limite pas au niveau système, il existe aussi des travaux qui démarrent d'une description SystemC pour implémenter des applications matérielles [32], [33]. L'apparition de SystemC offre aussi de grandes possibilités pour concevoir et tester des systèmes reconfigurables dynamiquement [34].

[14] TLM : Transaction Level Modeling

II.2.2.4. Partie software et processeurs embarqués

La synthèse de la partie software destinée à être implémentée sur un processeur embarqué est une étape compliquée notamment en tenant compte de la complexité de l'application et de l'aspect reconfigurable du design. L'implémentation d'un tel système reconfigurable peut varier, selon le profil de l'application, d'une description matérielle complète vers une implémentation entièrement logicielle qui s'exécute sur un processeur embarqué [35], [36] dans la plateforme. La partie logicielle contrôle, en général, les blocs d'IPs implémentés dans la partie matérielle. Dans le cas d'un système complexe, un système sur puce multiprocesseur peut exécuter un ensemble de programmes simultanément. Ces programmes doivent gérer les conflits d'accès à des ressources partagées représentées par les blocs d'IPs matériels, les périphériques de communication, d'interface et de stockage.

La synthèse automatique des blocs matériels d'IPs, des parties logicielles adéquates et des interfaces de communication entre les différents composants est effectuée au niveau de l'étape de partitionnement. Les algorithmes de partitionnement seront détaillés dans II.3.

II.3. Systèmes sur puces reconfigurables dynamiquement

L'évolution rapide des composants reconfigurables tels que les composants FPGA associée à la complexité des designs tels que les applications multimédia imposent de nouveaux défis à la communauté qui s'intéresse à la conception des SOPCs. En effet, c'est en constatant ce déséquilibre de plus en plus grand entre les besoins des applications et ce qu'offrent les systèmes qu'est né le concept des systèmes dynamiquement reconfigurables. La fixation d'un cadre méthodologique pour la conception d'un système sur puce dynamiquement reconfigurable qui génère une distribution flexible et adaptée de ses composants tout en vérifiant son intégrité et en favorisant la réutilisation de ses composants est l'un des plus importants problèmes à résoudre. L'implémentation d'un SOPC dynamiquement reconfigurable consiste à lui attribuer une architecture réactive et flexible qui s'adapte pour satisfaire des contraintes de coût, performances ou de ressources. Une architecture reconfigurable comporte des composants de contrôle, des unités de traitement, de mémorisation et des interfaces de communication. L'approche générale de reconfiguration dynamique consiste à fractionner l'exécution séquentielle d'un algorithme en reconfigurant en ligne des parties matérielles et logicielles de l'application. La version précédente de l'approche dynamique de reconfiguration est la synthèse architecturale. La synthèse d'architecture se base sur la génération automatique mais statique d'une architecture du design qui favorise la réutilisation des composants.

II.3.1. Evaluation d'une architecture reconfigurable

Dans ce paragraphe, nous allons élaborer les critères d'évaluation d'un système reconfigurable qui sont la flexibilité, les performances, la consommation d'énergie, le degré de parallélisme et le coût d'implémentation du design.

1. La flexibilité d'une application consiste à caractériser le degré avec lequel on peut adapter l'architecture du système en utilisant un processus déterministe.
2. Les performances d'un tel système sont liées aux nombres d'opérations de calcul élémentaire qu'il peut réaliser et au débit de données qu'il peut échanger avec son environnement. L'amélioration des performances du système est, généralement, couplée à une augmentation de la consommation d'énergie du système. La réduction de la consommation d'énergie nécessite un flot de conception adapté capable de vérifier la contrainte de faible consommation à chaque niveau d'abstraction.

3. Le degré de parallélisme détecté au niveau de l'application, est un paramètre déterminant pour comparer les architectures reconfigurables synthétisées.
4. Le dernier critère de comparaison entre les différentes architectures reconfigurables est le critère coût. Ce critère regroupe plusieurs paramètres reliés au temps, aux outils de conception et au prix du prototypage et de l'implémentation finale. Les critères énumérés dans ce paragraphe permettent l'évaluation des performances des architectures reconfigurables.

II.3.2. Caractérisation d'une architecture reconfigurable

Les critères énumérés dans le paragraphe précédent permettent l'évaluation des performances des architectures reconfigurables. Pour caractériser ces architectures, on distingue quatre critères qui permettent la classification des architectures reconfigurables.

II.3.2.1. La granularité des ressources de traitement

Le niveau de granularité qui caractérise une architecture ou une plateforme reconfigurable varie entre une reconfiguration grain fin et gros grain. En effet, la reconfiguration dynamique peut adapter des ressources logiques élémentaires de la plateforme (grain fin), comme elle peut reconfigurer des composants représentés, à titre d'exemple, par des coeurs de processeurs (gros grain). Dans le cas d'une architecture hétérogène, le processus de reconfiguration peut agir sur différent niveaux de granularité ce qui évoque la notion de reconfiguration multi-grain. Les plateformes FPGA ont une faible granularité puisqu'elles sont composées par des unités fonctionnelles à quatre entrées. Les composants FPGA sont configurés en chargeant une séquence de bits qui représente la reconfiguration ou le contexte de l'application. La reconfiguration dynamique de la plateforme consiste à recharger, selon l'évolution du système, une nouvelle séquence de bits décrivant la nouvelle reconfiguration. Le délai de la reconfiguration dépend de la taille de la séquence de bits à charger sur la plateforme reconfigurable. Les architectures reconfigurables au niveau gros-grain nécessitent des séquences plus courtes que celles des architectures à grain fin, ce qui entraîne un délai important pour reconfigurer des architectures à grain fin. Une faible granularité provoque une plus grande flexibilité de reconfiguration du matériel mais elle entraîne un délai d'adaptation important.

II.3.2.2. La communication processeur - accélérateur

Une architecture reconfigurable est généralement implémentée autour d'un processeur hôte qui gère la reconfiguration de la partie matérielle (les interfaces et les IPs) et qui assure le transfert de données entre les différents composants. Le type de la liaison processeurs–accélérateurs a une grande influence sur les performances de la reconfiguration. On distingue trois types de liaison processeurs–accélérateurs:

- Liaison en mode périphérique : l'échange de données est assuré à travers des interfaces d'entrées/sorties. Dans ce cas, le processeur communique avec l'accélérateur matériel comme avec n'importe quel autre périphérique ce qui génère un temps de latence important. La reconfiguration reste stable pendant un délai de traitement réalisé sur les accélérateurs.
- Liaison en mode co-processeur : l'échange de données est assuré à travers le bus local du processeur hôte. Le processeur communique avec l'accélérateur matériel (un co-processeur dans ce cas) en utilisant le même protocole de communication. Dans ce cas de liaison, la reconfiguration de l'architecture peut changer au cours du traitement à la

suite d'une interruption ou de l'exécution d'une application particulière sur le processeur.

- Liaison directe : Dans ce cas l'accélérateur représente une unité de traitement spécifique pour le processeur qui est donc directement implémentée sur le chemin de données interne du processeur ce qui minimise au maximum les délais de communication.

II.3.2.3. La reconfigurabilité

Le processus de la reconfiguration peut agir sur plusieurs composants de l'architecture notamment les ressources de traitement et les réseaux de communication. L'intervention du processus de reconfiguration peut être d'une manière statique ou dynamique, partielle ou totale :

- Reconfiguration statique : ce mode de reconfiguration consiste à fixer une configuration unique du système. Pour reconfigurer un système, l'utilisateur doit interrompre le fonctionnement de l'application pour recharger une nouvelle configuration.
- Reconfiguration dynamique : il s'agit de reconfigurer l'architecture du système d'une façon dynamique, automatique, partielle ou totale. On distingue deux approches de reconfiguration dynamique qui sont la reconfiguration à contexte unique et celle multi contextes. La reconfiguration à contexte unique ou pseudo-dynamique se base sur un seul plan d'adaptation qui supporte une seule reconfiguration possible. Tout changement de contexte provoque la reconfiguration totale de la plateforme. Contrairement à la reconfiguration à contexte unique, la reconfiguration multi contextes gère plusieurs plans d'adaptation possible. Un plan de reconfiguration unique peut être actif à un moment donné. L'architecture de l'application évolue en cours de traitement.
- Reconfiguration partielle : la reconfiguration partielle affecte une zone précise de la plateforme alors que le reste du système reste inactif tout en gardant ses informations de configuration.

II.3.2.4. L'organisation mémoire

Les calculs exécutés sur les composants reconfigurables s'effectuent sur des données stockées en mémoire. Dès que le nombre de données traitées est important, les traitements génèrent des résultats intermédiaires qui doivent être aussi sauvegardés avant le passage à la nouvelle reconfiguration. L'organisation de la mémoire sur la plateforme reconfigurable a une grande influence sur la rapidité d'accès aux données.

II.3.3. Reconfiguration au niveau système

Concevoir une architecture reconfigurable au niveau système consiste à connecter plusieurs processeurs programmables auxquels sont associées des ressources de calculs spécifiques. Les architectures reconfigurables au niveau système telles que le Lx de HP [15] et STMicroelectronics [39] (Fig. II.4) décomposent l'exécution de l'application sur plusieurs processeurs spécialisés composés par quatre unités d'exécution VLIW [16]. Chaque unité comporte quatre unités arithmétiques et logiques (UAL) sur 32 bits, deux multiplieurs, une unité de chargement et stockage (load/store) et une unité de branchement.

[15] HP : Hewlett Packard
[16] VLIW : Very Long Instruction Word

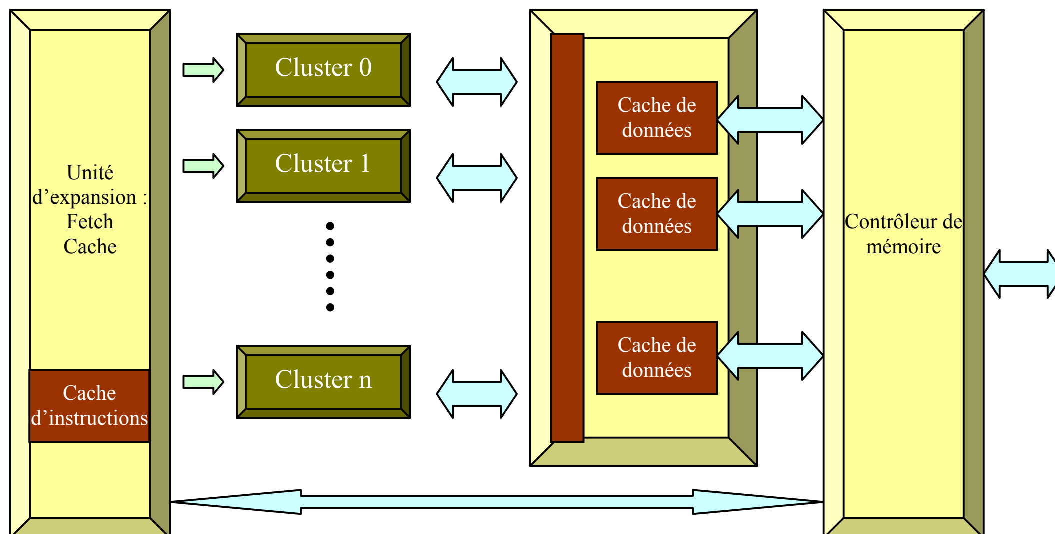


Fig. II. 4 : Structure de l'architecture LX [39]

En plus de la possibilité de la reconfiguration au niveau des paramètres des unités de traitement spécifique tels que le jeu d'instruction et la grandeur des données traitées, l'approche de la reconfiguration dynamique peut être introduite en ajoutant un cœur adaptable implémenté sur une plateforme reconfigurable de type FPGA par exemple. L'utilisation de cette approche est particulièrement intéressante en cas d'implémentation d'une application faisant un usage intensif d'un algorithme particulier. En adoptant cette architecture, le concepteur limite l'application des aspects reconfigurables à des fonctionnalités spécifiques de l'application à implémenter.

La dynamique de l'application est contrôlée selon la charge de travail fournie par chaque composant de l'architecture. Au niveau système, la performance de chaque composant est caractérisée par le nombre de cycles nécessaires pour effectuer un traitement spécifique. L'utilisation d'architectures reconfigurables au niveau système aide le concepteur à analyser la flexibilité de l'application.

II.3.4. Reconfiguration au niveau fonctionnel

Les architectures reconfigurables au niveau fonctionnel comportent des blocs de traitement dédiés et programmables à gros grain reliés par un réseau d'interconnexion reconfigurable. En plus de la reconfigurabilité des blocs de traitement, les concepteurs s'intéressent dans ce niveau à la reconfiguration des interconnexions tout en optimisant les échanges. En fonction du nombre d'unités fonctionnelles, le concepteur opte pour une solution statique ou dynamique. Dans la suite, nous allons présenter quelques architectures reconfigurables au niveau fonctionnel.

- L'architecture MorphoSys [40] a été développée dans un cadre universitaire. Cette architecture est constituée d'une matrice 8x8 de cellules reconfigurables, une mémoire de contexte, un cœur de processeur, un buffer de données et un contrôleur d'accès mémoire dynamique. La cellule reconfigurable est l'élément de traitement le plus important dans l'architecture. Chaque cellule est composée de quatre éléments de base : une unité arithmétique et logique pour le traitement, une mémoire de données, un module d'entrée/sortie pour assurer l'interconnexion avec les autres unités reconfigurables et finalement un bloc de composants logiques reconfigurables à grain fin. Le processeur contrôle les opérations de base de la matrice à travers des

instructions spéciales ajoutées à son jeu d'instructions. La mémoire sert à stocker les différentes configurations possibles de l'architecture. Cette architecture est considérée reconfigurable à gros grain puisqu'elle opère sur des données de largeur de 8 à 16 bits bien que chaque cellule intègre une unité de composants reconfigurable à grain fin opérant au niveau bit.

- L'architecture Pleiades [63] a été développée à Berkeley et a été conçue pour respecter la contrainte de faible consommation. L'architecture Pleiades est composée d'un processeur hôte, plusieurs unités d'exécution appartenant à des différents niveaux de granularité, de la mémoire et un réseau d'interconnexion qui relie les composants de l'architecture. Suivant le domaine de l'application, une nouvelle instance d'architectures peut être créée en adaptant le type ou le nombre des unités d'exécution. La méthode de partitionnement associée à cette architecture est purement manuelle. L'implémentation démarre d'une description purement logicielle exécutée sur le processeur hôte. Si les contraintes de temps et de consommation ne sont pas respectées, les tâches sont migrées manuellement vers les unités d'exécution.
- L'architecture aSoC[17] [64] qui a été développée à l'université du Massachusetts. Cette architecture est composée d'un réseau maillé d'interconnexions pipelinées de cœurs de traitement hétérogènes. A partir d'une description C de l'application, les concepteurs traduisent la spécification en une représentation intermédiaire sous la forme d'un graphe de contrôle de flots de données. Les concepteurs entament une étape d'exploration architecturale en estimant les performances de chaque cœur de traitement. l'allocation des cœurs de traitement se fait en utilisant un algorithme glouton qui se base sur les estimations de performances.
- L'architecture DART[18] [65] qui a été développée à l'université de Rennes 1 est une structure dynamiquement reconfigurable. L'adaptation de la plateforme est effectuée d'une manière partielle, en ligne et dynamique. Cette architecture est composée d'un contrôleur de tâches, de ressources de mémorisation et de ressources de calcul appelées clusters. Le contrôleur de tâches affecte les différents traitements à exécuter aux différents clusters. Chaque cluster peut représenter des différents niveaux de granularité puisqu'il est constitué à base de FPGA opérant sur des données de largeur 8 à 16 bits.

II.3.5. Reconfiguration au niveau portes logiques

Les architectures reconfigurables au niveau portes logiques s'intéressent à l'adaptation du design implémenté sur une plateforme reconfigurable généralement de type FPGA. La technologie de reconfiguration la plus utilisée se base sur des mémoires de type SRAM[19] qui permettent le stockage des différentes configurations. L'architecture de l'application peut être indéfiniment reconfigurée.

II.4. Méthodologies statiques de partitionnement logiciel/matériel

Plusieurs méthodologies de conception de systèmes sur puce matériel/logiciel ont été proposées dans la littérature. Ces méthodologies, représentées essentiellement par des algorithmes d'exploration, s'intéressent essentiellement à la phase de partitionnement logiciel/matériel au niveau système. Le problème de partitionnement peut se décrire comme la

[17] aSoc : adaptive System on Chip
[18] DART : Dynamically Reconfigurable Architecture dealing with next Generation Telecommunications Constraints
[19] SRAM : Static Random Access Memory

recherche d'une répartition des fonctionnalités d'une spécification sur une architecture hétérogène cible, cette répartition étant déterminée de manière à garantir certaines performances et d'optimiser un ou plusieurs critères. Les méthodologies de partitionnement sont basées sur plusieurs heuristiques et techniques d'optimisation. Ils se différencient par différents objectifs à optimiser notamment les performances, l'utilisation des ressources et la consommation en énergie du design. Le tableau II.1 regroupe plusieurs algorithmes de partitionnement automatique ainsi que leurs caractéristiques.

Table. II. 1 : Algorithmes de partitionnement logiciel/ matériel

	LANGAGE DE SPECIFICATION	TECHNIQUE DE DECOUPAGE	ARCHITECTURE	VALIDATION LOGICIELLE/MATERIELLE
Ptolemy	FDS (Flot de Données Synchrone)	Découpage automatique orienté logiciel ; optimisation du temps global d'exécution	Mono processeur	Processus légers communicant à travers des portes (sockets) ; communication synchrone
Cosyma	Cx (C parallèle)	Découpage automatique orienté logiciel, basé sur des estimations dynamiques de performance	Mono processeur	Processus communicant à travers le modèle CSP ; utilisation de portes (Socket)
SpecSyn	SpecChart	Découpage automatique basé sur la technique de groupement	Multi processeur	Processus communicant à travers le modèle CSP
Co-Saw	CSP	Découpage interactif basé sur la technique de groupement	Mono processeur	Sockets du système UNIX
Corba	VHDL	Découpage non automatique orienté matériel	Mono processeur	Simulation dirigée au VHDL. Le logiciel est décrit en langage assembleur
Vulcan II	HardwareC	Découpage automatique orienté matériel	Mono processeur	Co-simulation au niveau assembleur ; communication dirigée par des évènements discrets
Rassp	VHDL	Découpage non automatique orienté matériel	Mono processeur	Co-simulation à base de modèles détaillés des processeurs et ASIC au niveau des portes
Lycos	VHDL ou C	Découpage manuel	Mono processeur	Processus communicant
Tosca	SpeedChart	Découpage manuel	Mono processeur	Validation par prototypage physique
Codes	SDL, StateChart	Découpage manuel	Mono processeur	Validation par prototypage physique
MCSE	Formalisme MSCE	Découpage interactif base sur des estimations de performance	Multi processeur	Co-simulation au niveau modèle suivie par simulation VHDL et C++
CoWare	POPE/SystemC	Synthèse des interfaces et raffinement des processeurs	Multi processeur	Validation par prototypage physique

Nous allons détailler dans les sections suivantes quelques algorithmes de partitionnement élaborés dans la littérature.

II.4.1. Partitionnement manuel

Les premiers travaux de partitionnement proposent des environnements manuels de conception utilisés généralement pour la co-simulation. On peut citer par exemple l'environnement Coware [14] qui propose une solution de partitionnement représentée par la figure II.5.

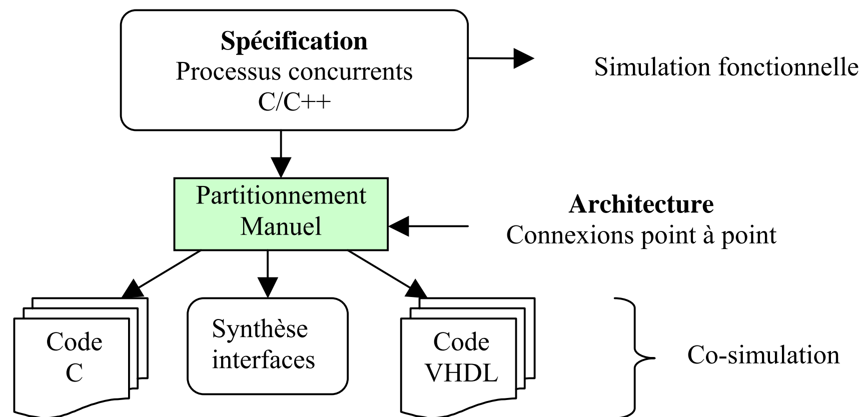


Fig. II. 5 : Partitionnement manuel : Coware

Le flot de la figure II.5 expose la méthodologie de partitionnement de Coware. Cette méthodologie est basée sur la spécification des processus concurrents codés en C/C++ et sur une architecture fixée par le concepteur. Le flot génère la partie logicielle codée en C, la partie matérielle en VHDL et les interfaces entre les deux parties. On distingue plusieurs autres flots de partitionnement automatique appliqués sur des applications orientées contrôle/commande.

II.4.2. L'algorithme de partitionnement GCLP

Un algorithme de partitionnement classique commence généralement par une étape de translation qui traduit la description fonctionnelle de l'application vers une description formelle sous la forme d'un graphe acyclique de tâches appelé DAG[20]. Un DAG est composé d'un ensemble de nœuds qui caractérisent un traitement spécifique. L'interconnexion entre les nœuds du DAG est assurée par des arcs qui représentent la dépendance de données existant entre les différents nœuds. L'application d'un algorithme de partitionnement sur ce graphe de tâches consiste à affecter l'exécution de chaque tâche à un accélérateur matériel ou à un processeur. L'algorithme de partitionnement doit respecter quelques critères liés à la performance et au coût d'implémentation du design.

L'algorithme GCLP[21] [41] optimise deux fonctions objectives contradictoires. Le GCLP optimise la performance du design tout en minimisant l'allocation de ressources. Pour une telle optimisation, l'algorithme définit deux métriques qui sont le GC (Global Criticality), qui mesure une estimation de la valeur de la criticité globale du temps par rapport à la phase locale LP (Local Phase) de l'exécution de l'application. A chaque étape, le GCLP choisit son objectif adéquat en fonction de GC et LP. La valeur de GC est comparée à un seuil pour dire si le temps est critique ou non. La figure II.6 représente le fonctionnement global du GCLP.

[20] DAG : Directed Acyclic Graph
[21] GCLP : Global Criticality / Local Phase

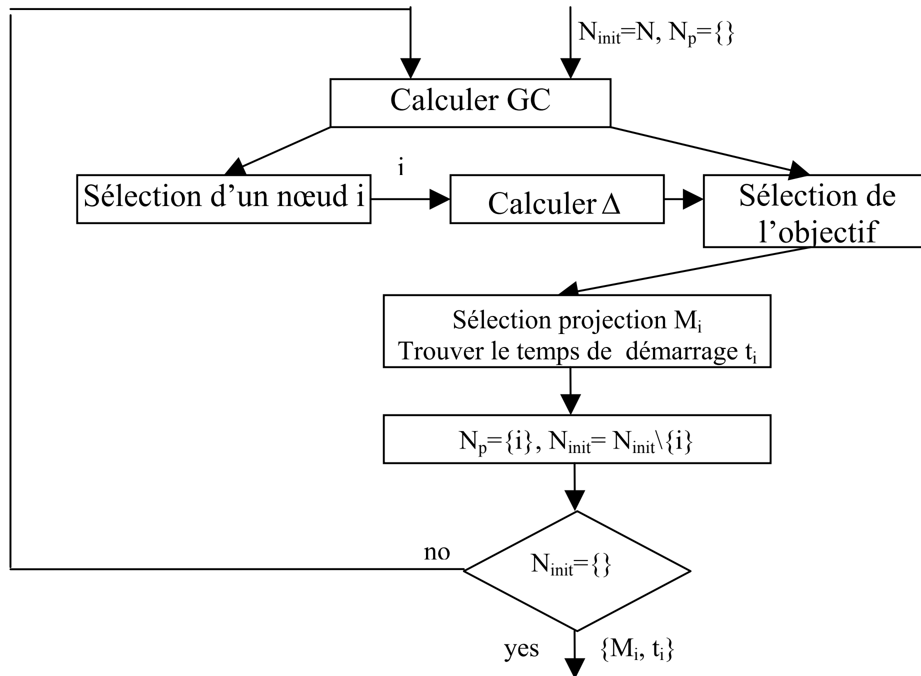


Fig. II. 6 : Organigramme de l'algorithme GCLP

Initialement, tous les nœuds (N_{init}) sont affectés à la partie software. A chaque période, l'ordonnancement des différentes tâches est connu. Si on ne peut pas affecter toutes les tâches à la partie software (dépassement de temps d'exécution pour cette phase locale), quelques ensembles de traitement désignés par des nœuds différents migrent vers la partie matérielle (N_p). Le nombre de nœuds qui ont migrés de la partie logicielle vers la partie matérielle détermine la métrique GC. Une importante valeur de GC implique que cette période d'exécution de l'application est particulièrement critique.

II.4.3. L'algorithme de partitionnement MAGELLAN

L'entrée de l'algorithme de MAGELLAN [42] est une spécification de l'application sous la forme de graphes hiérarchiques de contrôle du flot de données et de tâches. Le graphe de tâches est constitué d'entités ports, de canaux de communication et de tâches. MAGELLAN supporte 4 types différents de tâches qui sont 'leaf task', 'case task', 'loop task' et 'hierarchical task'. L'algorithme utilise d'une manière itérative un répartiteur pour le partitionnement et un ordonnanceur pour l'ordonnancement des tâches afin de déterminer l'architecture finale du design. A partir de la spécification de l'application, MAGELLAN génère en premier lieu, une solution initiale d'ordonnancement des différentes tâches. En deuxième lieu, l'algorithme optimise la solution en utilisant différentes techniques de compilation comme le 'loop unrolling' (parallélisme au niveau des boucles) et l'ordonnancement spéculatif (dupliquer les blocs conditionnels). Le répartiteur tente ensuite d'améliorer cette solution en changeant la composition de l'architecture initiale en effectuant des permutations (appelées aussi des mouvements). Le répartiteur évalue l'architecture modifiée en invoquant l'ordonnanceur. L'exécution de MAGELLAN se termine lorsque le répartiteur est incapable d'effectuer un nouveau mouvement. L'organigramme de l'algorithme de MAGELLAN est présenté par la figure II.7.

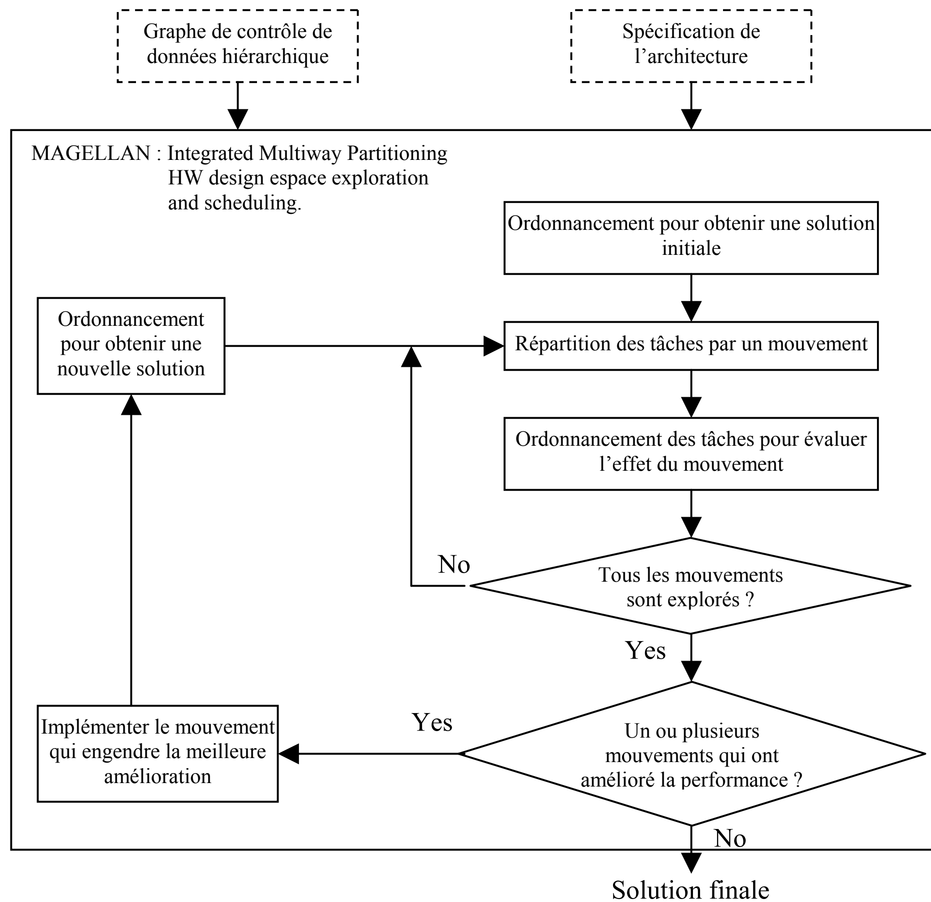


Fig. II. 7 : Organigramme de l'algorithme MAGELLAN

II.4.4. L'algorithme de partitionnement COSYN

L'algorithme COSYN [43] était le premier qui a pris en compte l'optimisation de la consommation d'énergie. Cet algorithme commence par effectuer une analyse du graphe de tâches et des contraintes système/tâche. La spécification d'entrée décrite en C est ensuite décomposée en blocs de traitement de base. Cette représentation est régénérée en utilisant un réseau de pétri coloré. En démarrant d'une description totalement software, COSYN déplace itérativement des blocs de traitement vers la partie matérielle en tenant compte de leurs performances et de la dépendance entre eux. Les performances de traitement sur la partie matérielle ou logicielle sont évaluées grâce à des estimateurs. La figure II.8 représente l'organigramme de fonctionnement de l'algorithme COSYN.

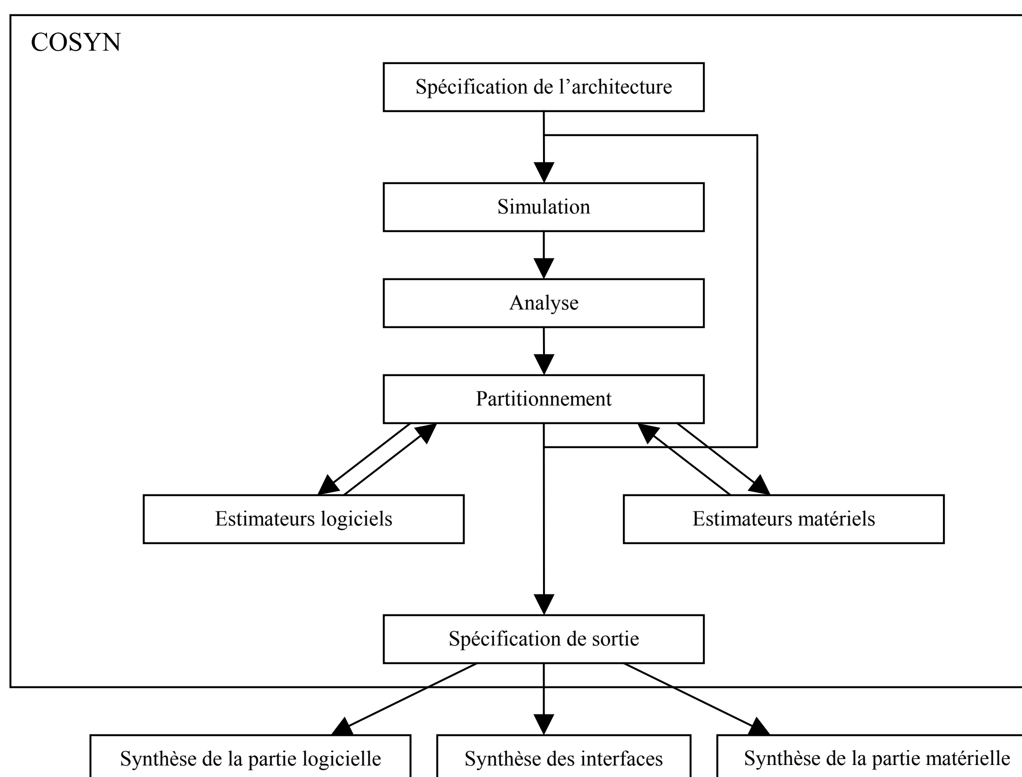


Fig. II. 8 : Organigramme de l'algorithme COSYN

II.5. Méthodologies statiques de synthèse d'architectures de systèmes sur puce

La synthèse automatique d'architectures embarquées intervient à plusieurs niveaux d'abstraction [52] dans le flot de conception de ces architectures. A chaque niveau et en fonction de l'état de l'application à implémenter, le concepteur est invité à sélectionner une architecture parmi plusieurs implémentations possibles. Plusieurs méthodologies ont été proposées dans la littérature permettant l'exploration et l'évaluation d'un espace d'architectures reconfigurables. La synthèse d'architectures reconfigurables au niveau système est une tâche très complexe. Sa complexité est due à la difficulté d'évaluer, à ce niveau d'abstraction, les performances et les ressources d'implémentation des architectures explorées. Les trois aspects suivants sont particulièrement intéressants pour la caractérisation d'un flot de synthèse automatique d'architectures embarquées :

II.5.1. Construction de l'architecture

Un problème de synthèse automatique d'architectures embarquées est défini comme un problème d'exploration. Etant donné un graphe de contrôle de flot de données (CDFG[22]) par exemple, un ensemble de ressources architecturales associé à un ensemble de tâches est alloué. Le flot de synthèse définit une interconnexion entre les ressources en minimisant un critère relatif à l'implémentation de cette architecture. Le problème de la synthèse architecturale peut être décomposé en deux sous problèmes : un problème d'ordonnancement et un autre d'affectation des ressources allouées.

[22] CDFG : Control Data-Flow Graph

II.5.1.1. Ordonnancement

L'étape d'ordonnancement détermine l'ordre d'exécution des différentes tâches de l'application. Etant donné un ensemble partiellement organisé de tâches associées à des délais d'exécution, l'algorithme d'ordonnancement détermine le temps de démarrage de chaque tâche tout en vérifiant les contraintes de dépendance spécifiées dans le CDFG. Des restrictions supplémentaires liées à l'implémentation de l'architecture comme la surface et la consommation d'énergie peuvent être prises en compte à ce niveau. Dans la littérature, on distingue deux algorithmes populaires d'ordonnancement qui sont le ASAP[23] et le ALAP[24]. L'algorithme ASAP effectue une organisation topologique des tâches ce qui simplifie la vérification des contraintes de dépendance. Le ASAP détermine d'une façon optimale le temps de démarrage de chaque tâche tandis que le ALAP détermine le temps maximum pour exécuter une tâche. Le délai possible d'exécution d'une tâche est déterminé en calculant la différence entre le temps limite d'exécution et le temps de démarrage de l'exécution de l'opération. Le problème d'ordonnancement devient un problème très complexe si on tient compte des différentes contraintes d'implémentation matérielle. La méthodologie d'ordonnancement peut être optimisée pour satisfaire plusieurs contraintes. On peut, par exemple, organiser les tâches pour optimiser la performance du design tout en minimisant les ressources. L'ordonnancement dynamique consiste à appliquer en ligne un algorithme pour adapter l'ordre des tâches d'une application qui s'exécute sur une plateforme dynamiquement reconfigurable.

II.5.1.2. Affectation des ressources

Le problème d'affectation des ressources consiste à associer un composant matériel à une ou plusieurs tâches. Le but de cette affectation est de minimiser le nombre de ressources allouées en favorisant la réutilisation des composants. L'opération d'affectation peut être réalisée avant ou après l'étape d'ordonnancement. Dans le cas d'une procédure de synthèse architecturale qui traite le problème d'ordonnancement suivi par l'affectation des ressources, les résultats d'ordonnancement représentent une contrainte pour l'affectation des ressources. Par exemple, deux tâches exécutées pendant le même intervalle de temps ne peuvent pas partager la même ressource. Le processus d'affectation des ressources affecte énormément les résultats d'implémentation de l'architecture notamment la surface, les performances et l'utilisation mémoire.

II.5.2. Techniques d'exploration et d'évaluation

Plusieurs techniques ont été adoptées dans la littérature pour simplifier l'exploration de l'espace d'architectures. La sélection d'une architecture performante est basée sur les estimations des performances détectées par implémentation directe ou par des techniques d'estimation. L'étape de reconfiguration peut agir sur plusieurs aspects paramétrables de l'architecture. L'introduction de la notion de reconfiguration dynamique a amplifié la complexité de l'exploration puisqu'elle introduit la notion de temps dans ce processus. En effet une architecture performante à un instant peut ne pas l'être après un certain délai.

Les techniques d'exploration les plus utilisées dans la littérature sont les méthodes d'exploration exhaustives qui évaluent toutes les solutions possibles dans l'espace de conception. Cette stratégie, non guidée par le concepteur, nécessite un temps d'exploration beaucoup plus important que les autres techniques d'exploration. Les techniques exhaustives sont inapplicables pour la synthèse de systèmes complexes. On distingue aussi les méthodes

[23] ASAP : As Soon As Possible ASAP
[24] ALAP : As Late As Possible ALAP

stochastiques qui mettent en œuvre des modèles probabilistes et aléatoires pour tester un nombre limité de solutions. En troisième lieu, on distingue les méthodes heuristiques qui utilisent des connaissances antérieures d'exploration pour limiter l'espace d'exploration. Ces méthodes ne garantissent une solution optimale mais elles assurent une convergence rapide. Dans la suite, on liste quelques travaux d'exploration spécifiques au problème de partitionnement.

Dans [49], Gupta propose un flot d'exploration qui optimise le partitionnement de l'architecture. Il commence par une description entièrement hardware puis il explore itérativement les résultats de partitionnement en déplaçant à chaque fois une tâche de la partie hardware vers la partie software. Contrairement à cette approche orientée hardware, les approches proposées dans le paragraphe II.3 démarrent d'une solution entièrement software. Catiana [50] utilise la logique floue pour automatiser l'étape d'exploration architecturale. D'autres techniques comme les algorithmes génétiques ont été adoptées pour proposer une méthodologie d'exploration des résultats de partitionnement logiciel/matériel [51].

La durée de l'exploration est directement liée à la dimension de l'espace d'exploration. Il est donc intéressant de chercher à diminuer le temps d'exploration. Différentes méthodes peuvent être mises en œuvre pour réduire l'espace d'exploration comme :

- Méthode de réduction par exploration hiérarchique : Cette méthode consiste à utiliser différentes stratégies d'exploration opérant chacune à un niveau d'abstraction différent. Des explorations initiales basées sur des estimations d'abord grossières puis fines permettent de réduire l'espace d'exploration.
- Méthode de découpage en petits éléments : Cette méthode consiste à diviser l'espace d'exploration en sous-espaces de tailles inférieures. La mise en concurrence des solutions finales relatives à chaque sous-espace permet de déterminer la solution optimale pour l'espace complet d'exploration.
- Méthodes de découpage en parties indépendantes : Cette méthode consiste à découper les tâches d'exploration dans le cas où on essaie d'optimiser des objectifs indépendants. Le but de cette méthodologie est de découper l'espace d'exploration en plusieurs sous-espaces indépendants.

II.5.3. Objectif d'optimisation

La sélection de l'objectif à optimiser par le flot de synthèse architecturale est une étape déterminante dans le processus d'exploration. En effet, les paramètres à optimiser pour l'implémentation d'un système embarqué ne se limitent pas à la surface d'implémentation, la performance d'exécution et la consommation d'énergie. On peut définir un espace d'objectifs contenant plusieurs paramètres liés aux contraintes et performances du système à développer. Les principaux objectifs d'exploration sont :

- La performance de traitement du système : cette performance qui illustre la vitesse de fonctionnement du système, peut être représentée en utilisant plusieurs métriques comme la fréquence maximale de fonctionnement du système, le débit de communications, le nombre d'opérations par cycle.
- La consommation d'énergie : cet objectif devient de plus en plus l'objectif principal d'optimisation. Elle rentre en compte pour les systèmes embarqués à contraintes liées à la durée de vie et l'encombrement (taille des batteries) des composants.
- Le coût d'implémentation : l'objectif dans ce cas est de réduire le coût d'implémentation du système sur puce en privilégiant les solutions les moins

coûteuses. Cela peut être le résultat de la réduction de la surface de silicium consommée ou, dans le cas d'une implémentation sur FPGA, par le choix d'un taux d'utilisation maximum des ressources. Le coût en temps de développement peut entrer aussi en compte.

- La flexibilité : ce critère est particulièrement étudié dans le cas des architectures reconfigurables. Le dynamisme de la reconfiguration des caractéristiques de l'architecture peut être considéré dans les objectifs.
- La communication : l'étape d'exploration peut optimiser l'architecture en réduisant le taux de communication entre les différents composants de l'architecture. Plusieurs stratégies de communication peuvent être implémentées.
- On distingue aussi dans la littérature l'intervention de quelques objectifs complémentaires nécessaires pour la représentation d'un compromis entre des objectifs différents. Parmi ces objectifs complémentaires nous pouvons citer la fiabilité du système, le taux d'utilisation des ressources dans le temps et la testabilité du système.

On remarque bien ces aspects dépendants qui relient les différents objectifs à optimiser. L'optimisation d'un objectif produit impérativement la dégradation des autres objectifs. Cette dépendance impose l'utilisation de techniques d'exploration et d'estimation qui évaluent les performances de la solution à chaque génération d'architectures. Enfin, chaque domaine applicatif spécifique peut apporter de nouveaux objectifs qui n'ont d'intérêt que dans un cadre précis.

II.6. Méthodologies dynamiques de synthèse de systèmes sur puce dynamiquement reconfigurables

La synthèse automatique d'architectures dynamiquement reconfigurables est associée à une étape d'exploration de méthodes systématiques qui génèrent les contextes ou les reconfigurations spécifiques à l'application. En plus des problèmes cités dans le paragraphe II. 4 suite à une opération de synthèse d'architectures statiques comme les problèmes de partitionnement et d'ordonnancement, l'introduction de l'aspect réactif dans l'architecture d'une application reconfigurable amplifie la complexité de la synthèse de cette architecture. En effet, les méthodologies de synthèse d'une architecture dynamiquement reconfigurable doivent adapter la construction de l'architecture à chaque changement de contexte de l'application.

La définition d'une méthodologie de reconfiguration dynamique d'architectures consiste à préciser l'intervalle d'intervention de la notion de reconfiguration dynamique. En effet, plusieurs aspects liés à la conception de l'architecture peuvent être gérés dynamiquement notamment l'allocation et l'affectation des ressources, l'ordonnancement des tâches et des reconfigurations, l'interconnexion des composants et même le partitionnement HW/SW de l'application. L'introduction de l'aspect de reconfiguration dynamique a contribué à l'apparition de nouveaux aspects architecturaux reliés à la gestion, l'organisation et le contrôle des différentes reconfigurations.

La construction d'une architecture reconfigurable passe par plusieurs étapes. Dès qu'on vise l'automatisation de la génération d'architectures dynamiquement reconfigurables, le concepteur doit extraire les tâches communes et complexes de l'application. En effet, une fonction complexe appartenant à une application relative à un contexte particulier représente des ressources gaspillées dans le cas où d'autres applications appartenant à d'autres contextes sont exécutées sur la même architecture. Une fois déterminé, le compilateur développe

l'architecture de la partie reconfigurable (construction de l'architecture) tout en spécifiant les mécanismes de détection de changement de contextes et de reconfiguration.

II.6.1. Construction d'une architecture reconfigurable

La tâche de construction de l'architecture pour un système sur puce dynamiquement reconfigurable est une étape de décision qui fixe le nombre, le placement et la stratégie d'ordonnancement et de partitionnement dynamique des éléments de traitement reconfigurables et des blocs mémoires de l'architecture. A partir d'une spécification détaillée de l'application à implémenter, l'outil responsable de la construction de l'architecture ou le compilateur analyse les performances et les contraintes d'implémentation de l'application et génère l'architecture adéquate. Parmi ces compilateurs, on distingue celui développé à l'université de Stanford SUIF [25] [53] qui représente un environnement idéal pour l'optimisation de l'architecture d'un système. L'étape de synthèse d'architectures dynamiquement reconfigurables traite des problèmes d'ordonnancement [54] et de partitionnement dynamique [55]. L'ordonnancement est dit statique dans le cas où le système exécute les tâches dans un ordre fixé hors ligne. L'ordonnancement dynamique décide en ligne et autorise le changement de l'ordre d'exécution des tâches [44].

Yang [44] propose un algorithme d'ordonnancement conservatif qui utilise des informations prédites sur les performances des ressources caractérisées dans ce travail par le nombre d'exécution d'une opération de chargement de données (CPU Load). L'étape de prédiction est assurée par un prédicteur statistique à un pas (single-step-ahead predictor). La technique de prédiction utilise la tendance de l'historique de la série temporelle pour prédire la prochaine valeur de la série. L'algorithme d'ordonnancement optimise le processus de décision et d'affectation des tâches qui compose l'application exécutée sur une plateforme dynamique composée par des ressources de calcul hétérogène.

Karam [45] propose une autre méthode pour optimiser les performances d'un système sur puce en améliorant le partitionnement et l'ordonnancement des tâches de l'application. Cette méthode est appliquée à des applications à caractère transformatif (applications qui possèdent un comportement itératif tel que les applications multimédia). L'outil développé dans [45] optimise l'implémentation du design, généralement en pipeline, en adaptant son partitionnement et l'ordonnancement de ces tâches. L'outil optimise itérativement l'intervalle d'initialisation de l'application, le nombre d'étages de pipeline et l'espace mémoire utilisé par l'application.

Chen [46] développe un processus d'ordonnancement sensible au besoin de reconfiguration dynamique de la partie hardware de l'application. L'objectif de ce travail est de minimiser le nombre de reconfigurations de la partie hardware du système sur puce implémenté. D'autres objectifs peuvent être optimisés en adaptant dynamiquement l'ordonnancement des tâches de l'application comme la durée de transfert de données [47]. Dans [48], une méthodologie d'ordonnancement dynamique des fréquences possibles de la plateforme selon la charge de travail d'un processeur est présentée. L'objectif de cette méthodologie est de réduire la consommation d'énergie d'une plateforme multiprocesseur en adaptant la fréquence de chaque processeur au besoin de l'application. D'autres travaux s'intéressent à définir une méthodologie dynamique de partitionnement logiciel/matériel. Dave [57] présente un outil pour la co-synthèse hors ligne d'architectures HW/SW dynamiquement reconfigurables. L'outil présenté admet un graphe acyclique de tâches comme entrée et génère une architecture distribuée, hétérogène et dynamiquement reconfigurable tout en minimisant le coût de l'implémentation matérielle de l'application. Noguera [55] présente un algorithme constructif

[25] SUIF : Stanford University Intermediate Format

de partitionnement HW/SW pour implémenter des architectures dynamiquement reconfigurables. L'algorithme démarre son fonctionnement en fixant une solution initiale de partitionnement. Ensuite, il optimise cette solution en explorant l'espace de solutions possibles. L'algorithme de partitionnement prend en considération l'aspect adaptatif de l'application en introduisant des techniques de prefetching des reconfigurations au niveau de l'étape de partitionnement.

La notion d'implémentation d'une architecture reconfigurable est notamment adoptée dans le cas des applications multimédia. Russell [67] présente une implémentation reconfigurable d'un décodeur de Viterbi. La reconfiguration de l'architecture du décodeur adapte le nombre de canaux de communication aux besoins de l'opération de décodage. Ce besoin est détecté en se basant sur la valeur de bruit qui caractérise ces canaux de communication. Tandis que dans [68], Bouchoux propose une implémentation reconfigurable dynamiquement du codeur arithmétique du standard de compression d'images JPEG2000.

Dans cette section, nous avons détaillé quelques méthodologies de construction d'architectures reconfigurables. Ces méthodologies adaptent essentiellement l'allocation, l'ordonnancement et le partitionnement des différentes tâches qui constituent l'application. Ces méthodologies d'adaptation sont appliquées en ligne ou hors ligne. La réalisation de ces méthodologies exige implicitement, notamment dans le cas d'une application reconfigurable en ligne, la présence d'un contrôleur afin d'assurer la gestion des différentes reconfigurations. Dans la section suivante, nous introduisons quelques méthodologies de gestion des reconfigurations durant le cycle de vie d'un système.

II.6.2. Méthodologies de gestion des configurations de l'architecture

La gestion dynamique et en ligne des différentes reconfigurations durant l'exécution de l'application sur une plateforme matérielle/logicielle est une tâche complexe qui comporte deux étapes. La première étape consiste à détecter le besoin d'adaptation de l'architecture de la plateforme à l'application exécutée. La deuxième étape s'intéresse à l'ordonnancement, le prefetching et la réalisation de la reconfiguration. Plusieurs travaux qui proposent des flots de conception et de synthèse d'architectures reconfigurables en ligne ont été développés dans la littérature.

Ces travaux représentent, généralement, l'application en utilisant un modèle formel qui décrit le chemin de données de l'application. Lee [56] utilise une représentation en machine à état fini hiérarchique (HFSM[26]) associée à un modèle de flot de données synchrone (SDF[27]) pour modéliser, respectivement, les états et les traitements de l'application. La méthode proposée dans [56] définit une méthodologie d'ordonnancement en ligne des différentes reconfigurations du système. Une autre méthodologie d'ordonnancement en ligne est présentée dans [63] permettant la génération automatique et en ligne des reconfigurations du système.

Koch [58] propose une méthodologie générique de traitement des différentes reconfigurations partielles du système basée sur une analyse des données du fichier décrivant la nouvelle configuration. Cette analyse détermine les parties reconfigurables de l'architecture. Le traitement effectué consiste à extraire et compresser les données des sous-modules reconfigurables.

Resano [66] propose un système de contrôle de reconfiguration qui effectue, essentiellement, en ligne l'ordonnancement des différentes reconfigurations souhaitées. L'objectif de ce travail

[26] HFSM : Hierarchical Finite State Machine
[27] SDF : Synchronous Data Flow

est de minimiser le délai d'implémentation des différentes reconfigurations tout en minimisant le niveau de consommation d'énergie du design. La méthode d'ordonnancement utilisée dans ce travail se base sur deux phases. Une première phase d'exploration hors ligne qui consiste à tester l'effet de l'ordonnancement de plusieurs tâches sur la consommation d'énergie du design. La deuxième phase, exécutée en ligne, consiste a sélectionné l'ordonnancement adéquat parmi ceux qui ont été testé. Le même principe est utilisé dans [62] pour reconfigurer l'architecture d'une application embarquée. Le processeur de la plateforme alerte le contrôleur de reconfiguration de la nécessité d'accélérer le traitement. Le contrôleur de reconfiguration sélectionne l'architecture adéquate. Des informations de performances sont associées à chaque architecture.

Les travaux cités dans les paragraphes précédents proposent des méthodologies de gestion et de réalisation d'architectures reconfigurables au niveau implémentation (réalisation directe sur plateforme). Nous pouvons citer des travaux qui ont traité ce problème au niveau simulation [69-71]. Vasilko [69] propose une architecture basée sur l'implémentation de plusieurs chemins de données parallèles qui représentent les différentes reconfigurations. Le passage d'une reconfiguration à une autre est effectué d'une façon dynamique et en ligne. Noguera [70] estime la variation de la performance et de la consommation d'énergie dans une architecture reconfigurable implémentée au niveau système. Finalement, Boschetti [71], propose une méthodologie pour l'optimisation et le contrôle de la synthèse d'architectures reconfigurables implémentant une application de traitement d'images à partir d'un niveau de description système.

D'autres travaux s'intéressent à la réalisation d'un système d'exploitation générique pour gérer une architecture reconfigurable [59]. Ce système d'exploitation assure l'allocation, le partitionnement, le placement et le routage des différentes applications qui s'exécutent sur la plateforme reconfigurable. D'autre part, la focalisation sur la vérification de l'intégrité d'un système reconfigurable durant l'exécution de l'application a fait l'objet d'autres activités de recherche. Ces travaux proposent des flots de conception et de vérification de systèmes sur puces reconfigurables [60, 61], capables d'implémenter des systèmes sur puces reconfigurables et fiables.

II.7. Limitations des méthodologies statiques/dynamiques de conception de systèmes sur puce

La définition d'un flot standard pour la synthèse, la conception et la vérification d'un système sur puce dynamiquement reconfigurable est un sujet de recherche récent. De ce fait, les travaux de recherche, traitant cette discipline, visent la résolution de plusieurs problèmes. On distingue plusieurs problèmes liés à la synthèse architecturale, la conception selon plusieurs niveaux d'abstraction et l'implémentation sur une plateforme reconfigurable des architectures de systèmes dynamiquement adaptables.

Les limitations de la synthèse architecturale des systèmes reconfigurables sont surtout des problèmes d'exploration au niveau de la composition d'architectures en blocs de traitement, d'interfaces de communication et de périphériques de stockage. Ces problèmes sont de plus en plus complexes dans le cas d'une architecture hétérogène ou en pipeline. Une étape d'exploration est composée d'une étape de décision et une autre d'évaluation de l'architecture. La limitation au niveau de l'étape de décision consiste à trouver le bon mécanisme pour détecter les besoins de l'application, sélectionner le nouveau contexte et reconfigurer l'architecture. La limitation au niveau de l'étape d'évaluation consiste à déterminer les métriques nécessaires relatives à chaque niveau d'abstraction pour évaluer une architecture.

La limitation au niveau de la conception de systèmes sur puces reconfigurables dynamiquement est l'absence d'une méthodologie standard associée à des outils de CAO efficaces pour la conception dynamique multi niveaux de systèmes embarqués. Ces méthodologies nécessitent aussi la définition d'une méthode pour implémenter un contrôle efficace, intelligent et en ligne des changements de contexte durant l'exécution de l'application. Les outils futures de CAO supportant un flot de conception d'architectures reconfigurables dynamiquement doivent répondre à plusieurs contraintes telles que :

- Adaptabilité : ces outils de CAO doivent avoir la capacité d'adapter le fonctionnement du design d'une façon rapide en gardant l'intégrité fonctionnelle de l'application. Ces outils doivent aussi avoir la capacité de contrôler le dynamisme de l'application et de définir un contrôleur de reconfiguration et de l'introduire dans le design reconfigurable. Le contrôleur de reconfiguration est l'entité responsable de la gestion des reconfigurations de la plateforme en communicant des ordres d'adaptation aux différentes ressources de l'architecture.
- Extensibilité : les outils de CAO doivent être extensibles pour supporter des nouvelles technologies dans une architecture déjà implémentée. Ceci nous ramène au caractère plug-and-play du design.
- Interopérabilité : les applications d'aujourd'hui fonctionnent dans des environnements distribués et hétérogènes. Les outils de CAO appliqués à la conception d'architectures reconfigurables doivent implémenter une plateforme de communication regroupant tous les composants (HW/SW) d'un système tout en respectant les contraintes strictes de bande passante et de temps de réponse.
- Hétérogénéité : les mécanismes de contrôle implémentés dans ces outils de CAO doivent supportés l'interchangeabilité du fonctionnement de plusieurs applications appartenant à plusieurs domaines. Ceci nécessite une transition vers des mécanismes de contrôle plus flexible, moins restrictif et intelligent.

Les plateformes reconfigurables sont particulièrement conçues pour supporter l'implémentation d'applications complexes qui nécessitent une certaine flexibilité au niveau de l'architecture déployée par le design. Les contraintes principales qui défavorisent l'implémentation de systèmes reconfigurables sont la latence de reconfiguration au niveau de la plateforme et le surcoût au niveau de la consommation d'énergie. La reconfiguration partielle [37] associée à un modèle de prefetching des configurations futures [38] a été proposée pour résoudre les problèmes de délais de reconfiguration. Dans cette thèse nous essayons de proposer un flot de conception d'architectures dynamique au niveau système. Le flot proposé intègre un mécanisme de contrôle basé sur les réseaux de neurones. Ce mécanisme de contrôle agit sur les parties hardware et software du design. Les problèmes de communication ont été éliminés en appliquant notre méthodologie à un niveau TLM qui représente la communication inter composants à travers des transactions. Notre méthodologie introduit un flot générique de conception d'architectures reconfigurables applicable surtout sur des applications à caractère transformatif tels que les applications multimédia.

II.8. Conclusion

Dans ce chapitre nous avons détaillé le flot de conception des SOPCs statique ainsi que les méthodologies de synthèse d'architectures fixes. En deuxième lieu, nous avons présenté des aspects généraux de caractérisation et de classification d'architectures dynamiquement reconfigurables. Nous avons également exposé les différentes tendances de recherche relatives à la conception et l'automatisation de la synthèse d'architectures dynamiques.

Il apparaît que la conception d'une architecture dynamiquement reconfigurable est une nécessité imposée par la complexité des applications à implémenter. Cette complexité impose le recours à l'optimisation en ligne de l'architecture initiale afin de mieux gérer les ressources de traitement et d'énergie de la plateforme. La tâche d'optimisation agit sur les paramètres de l'architecture afin de les adapter aux besoins en unités de traitement variable de l'application. Cette variabilité dépend du changement au cours du temps du comportement de l'application ou de la nature des données traitées.

Il est clair aussi que les aspects statiques qui apparaissent dans les méthodologies de conception et de synthèse architecturale classiques ne satisfont pas les aspects variables et réactifs d'une architecture dynamique. L'absence d'un flot de conception standard et d'outils de CAO adaptés pour la conception multi-niveaux de systèmes dynamiquement reconfigurables nuit à l'évolution des systèmes réactifs.

La reconfiguration dynamique des systèmes sur puce est basée sur la variabilité de la charge de travail de l'application à implémenter. Le flot de conception des systèmes dynamiquement reconfigurables doit détecter et exploiter cette variabilité pour contrôler le processus de reconfiguration. Dans le prochain chapitre, nous allons exposer les métriques significatives illustrant la variabilité de la charge de travail d'une application. Nous allons également discuter les différentes causes de cette variabilité.

II.9. Références

[1] Henry. C, Larry. C, Merrill. H, Grant. M, Andrew. M, Lee. T, 'Surviving the SOC Revolution: A Guide to Platform-Based Design', Kluwer Academic Publishers, ISBN 0-7923-8679-5, 1999.

[2] David. R, Chillet. D, Pillement. S, Sentieys. O, 'Mapping Future Generation Mobile Telecommunication Applications on a Dynamically Reconfigurable Architecture', Proceedings of the IEEE International Conference of Acoustics, Speech and Signal Processing, Vol. 4, 2002, pp. 4194.

[3] Nasi. K, Karoubalis. T, Danek. M, Zdenek. P, 'FIGARO – An Automatic Tool Flow for Designs with Dynamic Reconfiguration', Proceedings of the 13th International Symposium on Field-Programmable Gate Arrsays, 2005, pp. 262-266.

[4] Actel: www.actel.com

[5] Xilinx: www.xilinx.com

[6] Altera : www.altera.com

[5] VirtexII: www.xilinx.com/products/silicon_solutions/fpgas/virtex_ii_pro_fpgas/index.htm

[8] Kalavade. A and Subramanyam. A, 'Hardware/ Software Partitioning for Multi-function Systems', Proceedings of the IEEE/ACM International Conference on Computer Aided Design, 1997, pp. 516-521.

[9] Synopsys: www.synopsys.com

[10] Cadence: www.cadence.com

[11] Celoxica : www.celoxica.com

[12] SystemC : www.systemc.org

[13] Arato. P, Mann. A. Orban. A, 'Algorithmic Aspects of Hardware-Software partitioning', ACM Transactions on Design Automation of Electronic System, Vol. 10, No. 1, pp. 136-156.

[14] Coware: www.coware.com

[15] G. Martin, 'SystemC and the Future of Design Languages: opportunities for Users and Research', Proceedings of the 16th Symposium on Integrated Circuits and Systems Design, 2003, pp. 69-78.

[16] Habibi. A, Tahar. S, 'Design For Verification of SystemC Transaction Level Models', Proceedings of the Design, Automation and Test in Europe Conference, DATE'05, 2005.

[17] L.Cai, P.Kritzinger, M. Oliveras, D. Gajaski, 'Top-Down System Level Design Methodology Using SpecC, VCC and SystemC', Proceedings of the Design, Automation and Test in Europe Conference, DATE'02, 2002.

[18] Savard.K, Ben-Fredj.N, Nicolescu. G and Jerraya. A, 'A Higher Level System Communication Model for Object-Oriented Specification and Design of Embedded Systems', 2001, pp. 69-78.

[19] Cai. L and Gajski, 'Transaction Level Modelling: An Overview', Proceedings of ACM international Conference on Hardware/Software codesign and system synthesis, 2003, pp. 19-24.

[20] Pek : http://www128.ibm.com/developerworks/power/pek/

[21] Cocentric System Studio: http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html

[22] Schulz-Key.C, Winterholer.M, 'Object-Oriented Modeling and Synthesis of SystemC Specifications', Proceedings of IEEE Asia South Pacific Design Automation Conference, 2004, pp. 238-243.

[23] Riccobene. E, Scandurra.P, Rosti.A, 'A UML 2.0 Profile for SystemC: Toward High-level SOC Design', Proceedings ACM international conference on Embedded software, 2005, pp. 138-141.

[24] Riccobene. E, Scandurra.P, Rosti.A, Bocchio. S, 'A SOC Design Methodology Involving a UML 2.0 Profile for SystemC', Proceedings of the Design, automation and Test in Europe conference, 2005, pp. 704-709.

[25] A. Wellig, 'Framed Complexity Analysis in SystemC for Multi-Level Design Space Exploration', Proceedings of Euromicro Symposium on Digital System Design, 2003, pp. 416-423.

[26] Jang Hye-On, Kang. M, Lee. M, 'High-level System Modelling and Architecture Exploration with SystemC on a Network SoC: S3C2510 Case study', Proceedings of the Design, automation and Test in Europe conference, 2004, pp. 538-543.

[27] T.Kogel, M.Doerper, A.Wiefernik, R.Leupers, G.Ascheid, H.Meyr, S.Goossens, 'A Modular Simulation Framework for Architectural exploration of on-chip Interconnection Networks', Proceedings of ACM international Conference on Hardware/Software codesign and system synthesis, 2003, pp. 7-12.

[28] Economakos. G, Oikonomakos, Panagopoulos. I, 'Behavioral Synthesis with SystemC', Proceedings of the Design, automation and Test in Europe conference, DATE'01, 2001.

[29] Lettnin. D, Braun. A, Bodgan. M, 'Synthesis of Embedded SystemC Design: A case Study of Digital Neural Networks', Proceedings of the Design, automation and Test in Europe conference, 2004, pp. 248-253.

[30] Rissa. T, Donlin. A, Luk.W, 'Evaluation of SystemC Modelling of Reconfigurable Embedded Systems', Proceedings of the Design, automation and Test in Europe conference, 2005, pp. 253-258.

[31] Baleani.M, Gennari.F, Jiang.Y, Patel.Y, K.Brayton, 'HW/SW Partitioning and Code Generation of Embedded Control Applications on Reconfigurable Architecture Platform', Proceedings of ACM international Conference on Hardware/Software Codesign and System Synthesis, 2002, pp. 151-156.

[32] Dhanwada. N, Lin.Ing-Chao, Narayanan. V, 'A Power Estimation Methodology for SystemC Transaction Level Models', Proceedings of ACM international Conference on Hardware/Software Codesign and System Synthesis, 2005, pp. 142-147.

[33] Brandolese. C, Fornaciari. W, Salice. F, 'An Area Estimation Methodology for FPGA Based Designs at SystemC-Level', Proceedings Design Automation Conference, 2004, pp. 129-132.

[34] Pazos. N, Maxiaguine. A, Ienne. P, 'Parallel Modelling Paradigm in multimedia applications: Mapping and scheduling onto a Multi-processor system-on-chip', Proceedings of the International Global Signal Processing Conference, 2004.

[35] www.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC.

[36] www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp

[37] Robertson. I, Irvine. J, 'A design flow for partially Reconfigurable Hardware', in the ACM Transactions on Embedded Computing Systems, Vol.3, No.2, May 2004, pp. 257-283.

[38] Li. Zhiyuan, Hauck. S, 'configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation', Proceedings ACM International Sympsium on Field Programmable Gate Array, 2002, pp. 187-195.

[39] Faraboschi. P, Brown. G, Fisher. A, Desoli. G, Homewood. F, 'Lx: A technology platform for customizable VLIW embedded processing', Proceedings ACM International Symposium on Computer Architecture, 2000, pp. 203-213.

[40] Parizi. H, Niktash. A, Bagherezadeh. N and Kurdahi. F, 'MorphoSys: A coarse grain reconfigurable architecture for multimedia applications', Euro-par 2002, LNCS 2004, pp. 844-848.

[41] Kalavade. A, Subrahmanyam. P.A, 'Hardware/Software Partitioning for Multi-function Systems', International Conference on Computer Aided Design, Proceedings IEEE/ACM International Conference on Computer-Aided Design, 1997, pp. 516 – 521.

[42] Catha. K, Vemuri. R, 'Magellan: multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs', Proceedings of ACM international Conference on Hardware/Software Codesign and System Synthesis, 2001, pp. 42-47.

[43] Hendry. D.C, Sananilkone. D.S, 'Hardware/ software partitioning of embedded systems with multiple hardware processes', IEE Proceedings on Computers and digital techniques, No. 5, Vol. 144, 1997, pp. 285-294.

[44] Yang. L, Schopf. J, Foster. I, 'Conservative Scheduling: using predicted variance to improve scheduling decisions in dynamic environments', Proceedings of ACM/IEEE Conference on Supercomputing, 2003, pp. 31-46.

[45] Karam. S and Vemuri. R, 'Hardware-software partitioning and pipelined scheduling of transformative applications', IEEE Transactions on Very Large Scale Integration systems, Vol. 10, No. 3, 2002, pp. 193-208.

[46] Chen. G, Kandemir. M, Sezer. U, 'Configuration-sensitive process scheduling for FPGA-based computing platforms', Proceedings of the Design, automation and test in Europe, 2004, pp. 16-20.

[47] Ito. K, 'A scheduling and allocation method to reduce data transfer time by dynamic reconfiguration', Proceedings Asia and South Pacific Design Automation Conference, 2000, pp. 323-328.

[48] Sinha. A, Chandrakasan. P, 'Dynamic voltage scheduling using adaptive filtering of workload traces', Proceedings International Conference on VLSI Design, 2001, pp. 221-226.

[49] Gupta. R.K, Micheli. G, 'System-level synthesis using re-programmable components', Proceedings of European Design Automation Conference, 1992, pp.2-7.

[50] Catania. V, Malgeri. M, Russo. M, 'Applying fuzzy logic to codesign partitioning', Proceedings IEEE Micro, 1997, pp. 62-70.

[51] Ben chida. K, 'Méthodologie de partitionnement logiciel/matériel pour plateformes reconfigurables dynamiquement', thèse soutenu en 2004, France.

[52] Muraoka. M, Nishi. H, Morizawa. K, Yokota. H, Onishi. Y, 'SOC architecture synthesis based on high-level IPs', IEICE Transaction fundamentals, Vol. E85, No. 12, 2004.

[53] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennesy, 'SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers', ACM SIGPLAN Symposium on Principles and practice of parallel pro programming, Vol. 29, No. 12, 1994, pp. 37-48.

[54] Nouguera. J, Badia. R, 'Dynamic run-time scheduling techniques for reconfigurable architecture', Proceedings International Conference on Hardware Software Codesign, 2002, pp. 205-210.

[55] Nouguera. J, Badia. R, 'A HW/SW partitioning algorithm for dynamically reconfigurable architecture', Proceedings of the Design Automation and Test in Europe, 2001, pp. 729-734.

[56] Lee. S, Yoo. S, Choi. K, 'Reconfigurable SOC design with hierarchical FSM and synchronous dataflow model', Proceedings International Symposium on Hardware/Software Codesign, 2002, pp. 199-204.

[57] Dave. P, "CRUSADE: Hardware/software co-synthesis of dynamically reconfigurable heterogeneous real-time distributed embedded systems", Proceedings Design, Automation and Test in Europe and Exhibition, 1999, pp. 97-104.

[58] Koch. D, Teich. J, "Platform-independent methodology for partial reconfiguration", Proceedings of ACM/CF, 2004, pp. 398-403.

[59] Wigley. G, Kearney. D, "The first real operating system for reconfigurable computers", Australasian Computer Systems Architecture Conference (ACSAC 2001), 29-30 January 2001, pp. 130-137.

[60] Borgatti. M and all, 'An Integrated Design and Verification Methodology for Reconfigurable Multimedia Systems', Proceedings Design, Automation and Test in Europe Conference and Exhibition, 2005, pp. 266-271.

[61] Hemmati. H, Niamanesh. M, Jalili. R, 'A Framework to support run-time assured dynamic reconfiguration for pervasive computing environments', International Symposium on Wireless Pervasive Computing, 2006.

[62] Benoit. P, Torres. L, Sassatelli. G, Robert. M, Becker. J, 'Dynamic hardware multiplexing: improving adaptability with a run time reconfiguration manager', Proceedings IEEE emerging VLSI technologies and architectures, 2006, pp. 251.

[63] Abnous. A, Rabaey. J, 'Ultra-low-power domain-specific multimedia processors', Proceedings IEEE Workshop on VLSI Signal Processing, 1996, pp. 461-470.

[64] Bousset. L, Burelson. W, Gogniat. G, Anand. V, Laffely. A, 'Targeting tiled architectures in design exploration', Proceedings International Parallel and Distributed Processing Symposium, 2003, pp. 8.

[65] David. R, Chillet. D, Pillement. S, Sentieys. D, 'DART: A Dynamically Reconfigurable Architecture dealing with next Generation Telecommunications Constraints', in Reconfigurable Architecture Workshop, 2002.

[66] Javier. R, Mazos. D, Catthor, F, Diederik. V, 'A reconfiguration Manager for Dynamically Reconfigurable Hardware', Transactions on IEEE Design & Test Computers, 2005, pp. 452-460.

[67] Russel. T, Sriram. S, Ramaswamy. R, Dennis. G, Wayne. B, ‘A Reconfigurable, Power-Efficient Adaptive Viterbi Decoder’, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 13, No. 4, 2005, pp. 484-488.

[68] Bouchoux. S, Bourennane. E, Paindavoine. M, ‘Implementation of JPEG2000 Arithmetic Decoder Using Dynamic Reconfiguration of FPGA’, Proceedings International Conference on Image Processing, 2004, pp. 2841-2844.

[69] Vasilko. M, Cabanis. D, ‘Improving Simulation accuracy in Design Methodologies for Dynamically Reconfigurable Logic Systems’,

[70] Noguera. J, Badia. R, ‘Power-Performance Trade-offs for Reconfigurable Computing’, Proceedings CODES+ISSS, 2004, pp. 116-121.

[71] Boschetti. M, Adario. A, Silva. I, Bampi. S, ‘Techniques and Mechanisms for dynamic Reconfiguration in an Image Processor’, Proceedings of Symposium on Integrated Circuits and Systems Designs, 2002, pp.

III. Application Adaptative et Variabilité des Charges de Travail

III.1. Introduction

La tendance actuelle au niveau de la conception de systèmes sur puce est de générer des architectures réactives qui s'adaptent en ligne pour mieux satisfaire les besoins de l'application. Les besoins de l'application qui s'expriment en termes de performance, de qualité de services ou de ressources de traitement sont décidés suite à la détermination de la charge de travail à effectuer par l'application. Dans ce chapitre, nous allons identifier les origines de la variabilité de la performance d'un système sur puce pouvant regrouper plusieurs applications. Nous citons, également, une partie des travaux de la littérature qui se sont focalisés sur les méthodes de représentation de cette variabilité.

La charge de travail (workload) de l'exécution d'une tâche sur un processeur est une fonction multi-niveaux à p dimensions regroupant p paramètres ou métriques qui caractérisent, au cours du temps, la performance d'exécution, le nombre de ressources et la consommation d'énergie de cette tâche. Nous citons à titre d'exemple le nombre de cycles par instruction ou par traitement particulier, le nombre d'accès mémoire en lecture ou en écriture ou aussi le nombre de traitement flottant par période de temps. La charge de travail est une fonction variable qui dépend de plusieurs paramètres internes (tel que la nature des traitements effectués par l'application) ou externes (tels que les préférences de l'utilisateur) de l'application. La charge de travail imminente de l'application détermine en grande partie la performance d'exécution de l'application.

Des techniques de détection et de représentation de la variabilité de la charge de travail d'une application ont été élaborées au niveau logiciel ou matériel. La nécessité de l'implémentation d'un système adaptatif au niveau logiciel est illustrée par l'exemple d'un système de gestion des requêtes de connexion à la demande des utilisateurs à un serveur Internet. L'adaptation dans ce cas consiste à gérer les ressources du serveur en associant chaque opération de traitement d'une requête à une ressource de traitement. La reconfiguration dans ce cas touche à la stratégie de gestion des connexions. Un deuxième exemple d'application concerne le codeur MPEG[28] qui représente un standard de compression vidéo. La variabilité de la charge de travail de cette application est une conséquence directe du comportement de l'algorithme de l'application en vue du changement de la nature ou de la grandeur des données à traiter. En effet, la performance d'exécution de l'application MPEG dépend du type du segment d'image (frame) traité.

Les exemples traités montrent la diversité des causes et des domaines d'application de la notion de reconfiguration dynamique. Cette diversité influence l'étape de conception et d'implémentation de l'architecture d'une application. Le besoin d'adapter le comportement d'un système est une conséquence de la variabilité des charges de travail de quelques modules ou de la totalité de l'application en cours de traitement.

Dans ce chapitre, nous allons présenter, en premier lieu, le domaine d'application de l'aspect adaptatif dans la description d'un système. Nous allons extraire les différentes caractéristiques qui favorisent l'introduction d'un aspect adaptatif dans l'architecture d'une application. En deuxième lieu, nous allons analyser les causes de variabilité en cours d'exécution des charges de travail de l'application. Ensuite, nous allons exposer les différents aspects reconfigurables d'une architecture embarquée. Finalement, nous allons présenter un exemple d'exploitation de la notion de variabilité des charges de travail d'une application afin de détecter les différentes phases d'exécution de l'application. Nous allons également exposer quelques travaux qui utilisent les résultats de cette analyse de variabilité pour définir une stratégie

[28] MPEG : Motion Photographic Experts Group

d'adaptation de l'architecture d'une application basée sur la prédiction des phases de l'application.

III.2. Domaine d'application

La tendance au niveau de la modélisation logicielle, matérielle ou mixte de systèmes à traitements spécifiques est de définir une méthodologie dynamique de conception et d'implémentation d'architectures sensible au comportement variable de l'application [1]. Les applications considérées dans un contexte d'architecture reconfigurable disposent d'un ensemble de caractéristiques qui favorisent l'introduction de cet aspect dynamique et réactif dans l'architecture de l'application. Ces caractéristiques sont essentiellement :

- Applications présentant des contraintes de temps d'exécution : cette contrainte temporelle est spécifiée dès le démarrage du processus de spécification de l'application. Elle détermine le délai limite possible pour produire un résultat. Les résultats à produire peuvent servir à d'autres tâches, à un ou plusieurs utilisateurs du système. Les exemples les plus concrets d'applications qui exécutent sous contraintes temporelles sont essentiellement les systèmes sur puces fonctionnant toujours sous des contraintes de traitement temps réel.
- Applications dissociables en sous tâches avec délai: l'adaptation influence généralement la décomposition en tâches de l'application. L'attribution d'un délai d'exécution à une sous tâche vient du fait que l'exécution complète d'une tâche doit être achevée pendant un délai pour servir d'autres tâches. La dépendance entre les sous tâches associées aux contraintes de traitement temps réel d'un système imposent cette décomposition sous contraintes temporelles.
- Applications présentant un comportement variables : une application regroupe des variantes qui représentent la source de variabilité du comportement de l'application. Ces paramètres seront distingués en spécifiant pour chaque variante un besoin particulier de ressource. A titre d'exemple, ces variantes peuvent être représentées par un niveau de qualité des résultats défini par l'utilisateur ou par un degré de contribution d'une tâche adaptable dans le processus de traitement.
- Application présentant un besoin important de ressources de traitement : on suppose dans ce cas que les ressources allouées par l'application pour opérer d'une façon correcte et dans les délais dépassent les ressources fournies par le plateforme d'exécution. Dans ce cas, le processus d'adaptation doit prendre en compte la disponibilité de la ressource avant de l'allouer. Le flot d'adaptation doit optimiser les performances de l'application tout en améliorant le taux de réutilisation des ressources du système.

Une application qui satisfait les caractéristiques citées ci-dessus possède un caractère adaptatif et appartient au domaine d'application des systèmes reconfigurables. Pour définir une méthodologie d'adaptation dynamique, le concepteur doit évaluer des critères reliés au comportement de l'application à implémenter. Ces critères sont généralement le degré d'adaptabilité de l'architecture, le degré de complexité d'une étape de traitement caractérisé par le type de traitement effectué et le nombre d'accès à la mémoire et le nombre de répétitions d'exécution de cette étape pour achever le traitement global désiré par l'application. Le degré d'adaptabilité est dirigé par l'évolution dynamique du comportement de l'application.

Le comportement variable d'une application est généré par la variation de sa charge de travail au cours du traitement. La charge de travail d'un système ou d'une tâche influence la

mesure de sa performance d'exécution. La performance d'un système est représentée par plusieurs métriques ou paramètres d'exécution. En effet, l'évaluation de la performance d'une application est basée sur l'estimation de plusieurs paramètres clés particulièrement importants dans des contextes différents. Le changement de ces paramètres de performance signale un changement de contexte de l'application. Par exemple, dans le cas d'une implémentation micro architecturale, les métriques significatives à évaluer sont [2]:

- Instructions déployées : ce paramètre mesure le nombre des différents types d'instructions exécutées. On distingue par exemple le pourcentage des instructions de calcul arithmétique entier, le pourcentage des opérations logiques, le pourcentage des opérations de décalage et de manipulation d'octets, le pourcentage d'opérations load/store, le pourcentage des opérations de contrôle et le nombre d'instructions par cycle (IPC[29]) .
- Le taux de réussite de la prédiction de l'adresse de branchement (branch prediction accuracy) : évaluer les performances de prédiction d'adresses de branchement en utilisant différentes techniques de prédiction.
- Le flot de contrôle : ce paramètre est caractérisé par le nombre d'instructions entre deux opérations de branchement.
- Comportement du flux de données : ce paramètre est caractérisé par la mesure du taux de défaut de cache (cache miss rate) en utilisant différentes capacités de mémoires cache pour stocker le flux de données à traiter.
- Comportement du flux d'instructions : ce paramètre est caractérisé par la mesure du taux de défaut de cache en utilisant différentes capacités de mémoires cache pour stocker le flux d'instructions.
- Parallélisme au niveau instruction (ILP[30]) : ce paramètre représente le degré de parallélisme qui relie les opérations ou les instructions d'un programme. Il représente un indice inhérent à tous les algorithmes. L'exploitation de ce type de parallélisme est relativement simple bien que son implémentation sera contrainte par les dépendances entre les données. Pour exploiter ce type de parallélisme, le contrôleur de l'architecture doit être capable de spécifier concurremment à plusieurs opérateurs, la fonction devant être implémentée. On peut distinguer d'autres critères qui s'intéressent aux aspects parallèles d'une architecture comme le TLP[31] qui décrit le nombre de traitements ou de tâches qui peuvent être exécutés en parallèle. Dans le cas d'une plateforme qui exécute plusieurs applications en parallèle, on peut distinguer un troisième niveau de parallélisme au niveau des applications.

On peut décomposer ces métriques en deux groupes : des métriques dépendant ou non de l'architecture. Les métriques de prédiction de branchement, de défauts de cache et d'IPC dépendent du choix de l'architecture. En revanche, le changement de l'architecture de l'application n'affecte pas les autres métriques.

Les métriques détaillées ci-dessus caractérisent les performances d'une application exécutée sur un processeur. La tendance actuelle est de développer des SOPCs pour résoudre les problèmes d'implémentation de plusieurs applications complexes sur la même plateforme. Les SOPCs regroupent un ou plusieurs processeurs, un réseau de communication, des mémoires et des IPs matériels spécifiques. L'introduction d'un caractère adaptatif qui gère les

[29] IPC : Instructions Per Cycle
[30] ILP : Instruction Level Parallelism
[31] TLP : Thread Level Parallelism

composants d'un SOPC est une nécessité pour atteindre les performances et les propriétés envisagées par le concepteur d'une application complexe [65, 66]. L'adaptation d'un SOPC passe obligatoirement par une étape d'estimation des performances des différents composants matériels et logiciels du système. L'évaluation des performances d'un SOPC au niveau système est une tâche complexe. En effet, cette évaluation de performances impose la caractérisation d'un ensemble d'événements architecturaux difficile à prévoir au niveau système. On distingue plusieurs outils de co-simulation au niveau cycles, détaillés dans le chapitre II, capables de vérifier et d'estimer les performances des architectures implémentées. Evaluer les performances d'un SOPC au niveau système implique la co-simulation des composants de la plateforme et la détection du nombre de cycles nécessaires pour réaliser une tâche particulière. La complexité et la diversité des applications implémentées sur des plateformes SOPC engendrent une variabilité au niveau des performances de l'architecture. Dans le paragraphe suivant on va analyser les différentes causes de variabilité de la performance d'une application.

III.3. Aspects variables des charges de travail

Dans cette section, nous allons détailler les causes de la variabilité des performances d'une application. Cette variabilité dépend de différentes contraintes internes ou externes du système. Les contraintes internes naissent de la description algorithmique qui définit un comportement variable de l'application. Un exemple concret de variabilité au niveau de l'algorithme consiste à implémenter un algorithme de compression d'images à qualité variable. La qualité et le taux de compression, dans ce cas, changent au cours du traitement et par la suite les paramètres et les traitements de l'application implémentée changent aussi. On peut distinguer, parmi les causes internes de variabilité, les paramètres liés à des aspects indéterminés d'exécution comme le temps d'exécution aléatoire alloué par une interruption générée par le système et exécutée par le processeur ou le temps d'accès mémoire variable géré par un DMA[32].

Les conditions les plus déterminantes qui précisent le degré de variabilité d'une application sont celles imposées par l'environnement extérieur de l'application [3]. On note, parmi les exemples d'application à charge de travail variable, le cas d'une application de traitement multimédia qui manipule un flux de données changeant au cours du temps. La variabilité d'un système peut être aussi le résultat d'un changement brutal du taux d'utilisation d'un système multi utilisateurs [67]. Cette variabilité peut être générée également par la migration de la fonctionnalité d'un système vers une autre application totalement différente. Le cas concret d'implémentation d'un tel système est représenté par un système sur puce qui co-intègre plusieurs applications sur la même plateforme. Les conditions externes d'exécution d'une application influencent fortement la charge de travail du système. La tendance au niveau des méthodologies de conception est de déterminer une stratégie d'implémentation adaptative et sensible à la variabilité de la charge de travail [68].

III.3.1. Variabilité dirigée par la nature des données

La variabilité des performances de traitement associée à une application dépend largement de la nature du flux de données traitées [4]. Une architecture embarquée qui implémente une application traitant des données multimédias, comme les applications de vision artificielle, d'imagerie 3D et de codage audio, représente un exemple significatif qui illustre cette variabilité temporelle. Dans le cas des applications multimédias, l'origine principale de cette variabilité de performance est l'instabilité au niveau des propriétés des données traitées

[32] DMA : Direct Memory Access

(images ou séquence vidéo) [5]. En effet, le traitement d'un flux de données multimédia, représenté à titre d'exemple par un ensemble de segments d'une image, est effectué pendant un nombre variable de cycles par segment. Dans le cas d'une application multimédia, le taux obtenu en divisant le nombre maximum par la moyenne d'opérations de chargement de données par le processeur peut aller jusqu'à 10 [6]. Le temps de codage des différents segments d'une image varie selon et le contenu de chaque segment. Une autre forme d'instabilité du flux multimédia consiste à coder un signal à nombre d'entrées variable. Par exemple, la procédure de décodage à longueur variable défini dans le standard MPEG (Moving Picture Experts Group) [19] consomme un nombre de bits variable pour produire un seul bloc décodé. Ce comportement génère une variabilité au niveau de la charge de travail du décodeur [18].

Une architecture système typique qui traite des données multimédias comporte des accélérateurs matériels spécifiques, un ou plusieurs composants programmables (processeurs, DSP ou contrôleurs) et de la mémoire distribuée. Le déploiement d'un large espace de stockage dans un système signifie sa pénalisation au niveau de la consommation d'énergie et de la surface d'implémentation. D'une autre part, une opération de transfert de données ou d'accès mémoire consomme typiquement plus d'énergie qu'une opération de traitement élémentaire. A titre d'exemple, une opération de recherche d'une donnée à partir d'un banc de mémoire consomme 33 fois plus d'énergie qu'une opération d'addition. Une opération de transfert de données à partir d'une mémoire consomme entre 4 et 10 fois plus d'énergie qu'une opération d'addition [16]. De ce fait, l'implémentation optimale d'une architecture d'un système à faible consommation d'énergie passe obligatoirement par le contrôle du mécanisme de stockage et des transferts de données effectués entre les composants du système [20]. Ce contrôle représente une tâche assez complexe quand il s'agit d'un système traitant un flux instable de données tel que les flux multimédias. L'instabilité au niveau des données traitées contraint le concepteur du système sur puce à insérer des registres additionnels entre les modules de traitement et l'interface d'acquisition [7]. Le concepteur est invité alors à déployer une stratégie d'ordonnancement adaptée [8] à la variation du flux multimédia traité. L'ajout de tels composants, effectuant le contrôle de l'ordonnancement dynamique, amplifie le coût et la consommation d'énergie du design. De ce fait, les concepteurs sont encouragés à caractériser la variabilité du flux de données à traiter par l'application afin de mieux contrôler la composition ou les paramètres (fréquence ou voltage [21]) de l'architecture implémentée. Les travaux présentés dans [17] proposent une méthode en ligne pour évaluer exactement l'espace de stockage nécessaire pour implémenter une application temps réel de traitement multimédia. Un autre travail présenté dans [69] s'est focalisé sur le développement d'une nouvelle approche pour sélectionner dynamiquement la taille de la fenêtre de l'historique d'un prédicteur des adresses de branchement. En revanche, dans [70], les auteurs proposent une méthodologie d'extraction des profiles des instructions d'un programme. L'extraction du profile d'un programme participe à la caractérisation de son comportement afin de sélectionner l'architecture du processeur adéquate à l'application.

III.3.2. Variabilité dirigée par l'environnement d'exécution du système

Dans ce cas, la variabilité de la performance provient de la fluctuation temporelle du taux d'utilisation du système. Cette notion est apparue suite à l'évolution de la complexité des applications implémentées sur des systèmes de plus en plus distribués accessibles pour différents utilisateurs. En effet, l'apparition de nouvelles gammes de services accessibles à distance à travers un réseau Internet ou un réseau mobile amplifie la complexité des tâches qui visent le contrôle de tels systèmes. Dans la littérature, on distingue deux types de systèmes qui illustrent bien ce phénomène de variabilité.

- Cas des applications pervasives [9] : l'implémentation des systèmes pervasifs est imposée par l'introduction d'un aspect de mobilité dans des systèmes complexes. Ces systèmes effectuent généralement un ensemble de traitements distribués tel qu'un système de traitement multimédia distribué et accessible via une connexion sans fils. Le challenge imposé par la mobilité de ces systèmes est d'introduire un caractère adaptatif qui répond aux préférences des utilisateurs, aux limites de la capacité du système et aux contraintes de la communication via un réseau sans fils. Un système pervasif dépend largement des conditions extérieures de l'application ce qui engendre un certain niveau de variabilité de sa charge de travail.
- Transmission d'un média continu via Internet : ce système peut être vu comme un cas particulier des systèmes pervasifs. Un serveur de transmission vidéo gère à chaque instant un nombre variable de connexions Internet. La variabilité de la charge de travail du serveur de connexion n'est pas générée par la mobilité du système mais par la variation du nombre de requêtes traitées par le serveur. Ce problème est de plus en plus posé avec l'émergence des sites web qui fournissent l'accès à des séquences vidéo ou audio telles que les services de transmission sur Internet de programmes de télévision ou radio [22]. En comparant la charge de travail associée à un flux multimédia avec celui d'une transmission Internet ordinaire, le flux multimédia utilise nettement plus d'espace de stockage et plus de ressources de traitement et de transmission. La caractérisation et l'analyse de la charge de travail d'un flux de transmission multimédia traitée dans [10] et [11] représentent la première étape dans le processus d'implémentation des composants d'un réseau (serveurs, routeurs) adaptatif.

Les deux exemples présentés ci-dessus illustrant la présence de la notion de variabilité générée par les nouveaux cas d'utilisation des systèmes complexes implémentés pour un usage accessible par plusieurs utilisateurs. L'aspect multi utilisateurs des systèmes partagés favorise dans ce cas la notion de variabilité. Plusieurs techniques, comme le 'caching' [23] et le 'prefetching' [24] des séquences multimédia à transmettre, ont été élaborées pour réduire la complexité des systèmes de diffusion vidéo. Les techniques proposées dans la littérature s'intéressent à distribuer la charge de travail sur les différents composants du système et de rapprocher le média à diffuser vers le site utilisateur. Le travail élaboré dans [25] propose un algorithme d'ordonnancement qui contrôle l'allocation d'un ensemble de supports de transmission en se basant sur des estimations de la charge de travail. Cette technique est applicable sur des systèmes de diffusion de média avec une variation très aléatoire de la charge de travail. Cette variation peut être causée par un événement répétitif pour des utilisateurs différents.

Une notion plus évoluée consiste à implémenter des systèmes, tels que les plateformes SOPC, qui regroupent plusieurs applications exécutées sur la même plateforme. Les systèmes implémentés seront alors multi applications et multi utilisateurs ce qui entraîne une variation de la charge de travail encore plus marquante.

III.3.3. Variabilité dirigée par la diversité des applications

Le progrès des systèmes informatiques mobiles et embarqués comme les applications implémentées sur les téléphones mobiles et les PDAs qui intègrent plusieurs applications multimédias complexes [12] sur la même plateforme est la première source de variabilité de la performance de tels systèmes. En effet, la diversité des applications exécutées sur la même plateforme produit une variabilité au niveau de la charge de travail du système complet. La caractérisation de l'évolution instantanée de la charge de travail des différentes applications est une étape nécessaire pour estimer les performances d'exécution du système implémenté

sur la plateforme. En conséquence, le concepteur peut envisager la mise en œuvre d'une stratégie d'ordonnancement dynamique implémentée sur cette plateforme multi applications capable d'organiser l'ordre d'exécution des différents processus [13]. Ces estimations sont aussi nécessaires pour adapter les caractéristiques de la plateforme, la fréquence d'exécution par exemple, à l'application en exécution [14].

Les capacités limitées des plateformes multi applications comme les terminaux mobiles ne permettent pas aux utilisateurs d'installer ou d'implémenter toutes les applications dont ils font usage. Ils ont donc besoin d'un mécanisme leur permettant de déployer provisoirement une application et la remplacer par une autre lorsqu'ils en n'ont plus besoin. Ce mécanisme automatique est implémenté à l'aide de la notion de la reconfiguration dynamique. Un système reconfigurable dynamiquement, implémentant plusieurs applications, modifie et gère [15] complètement ou partiellement le comportement et essentiellement la composition de la plateforme. L'étape de reconfiguration au niveau des applications comporte toutes les activités qui rendent une application directement utilisable par un utilisateur à un moment précis. Ces activités incluent l'installation dans le cas d'une implémentation logicielle ou la configuration dans le cas d'une implémentation matérielle reconfigurable, l'activation, la mise à jour et aussi la désinstallation et la reconfiguration d'une application. L'adaptation au contexte joue un rôle important dans le déploiement dynamique des applications puisqu'elle permet d'activer une application sur le site utilisateur qui s'adapte à ses besoins et à ses contraintes environnementales.

III.4. Origines de la variabilité des microarchitectures des processeurs embarqués

On s'intéresse dans cette section à la partie logicielle d'un système embarqué. Les programmes qui représentent cette partie seront exécutés sur des processeurs dédiés à être implémentés sur un système embarqué. L'ensemble logiciel et matériel intégré dans une plateforme unique constitue un SOPC. Un exemple de processeur embarqué est le PowerPC [26] d'IBM représenté par la figure III.1.

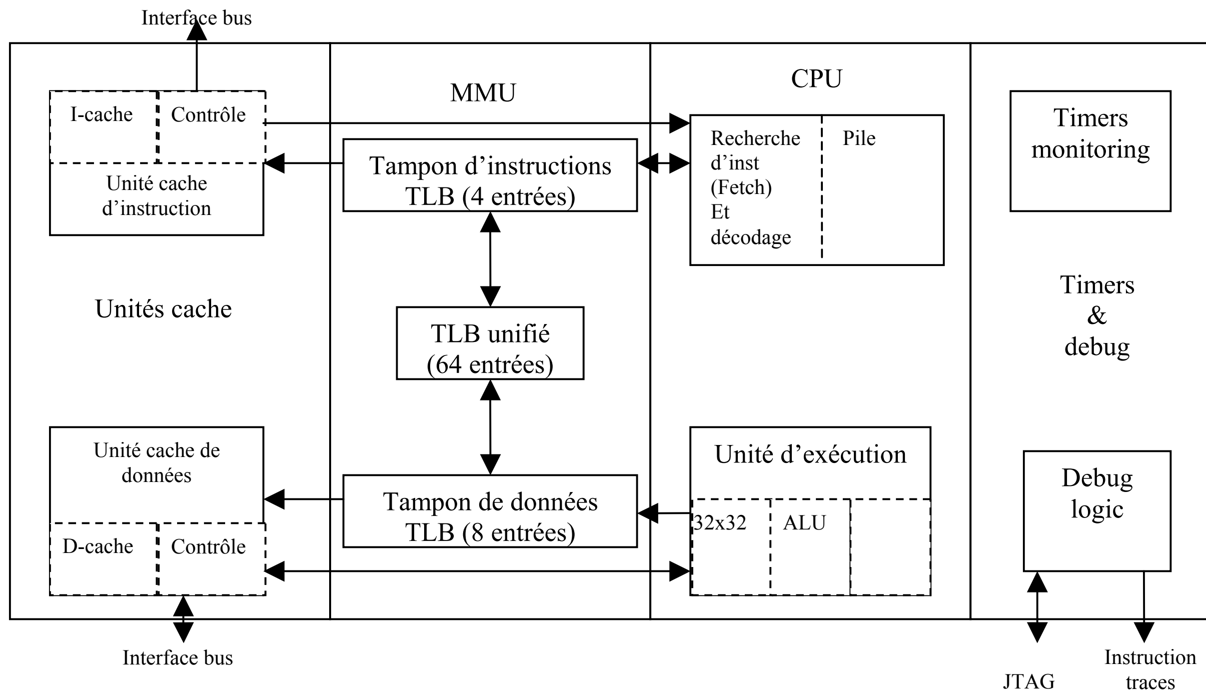


Fig. III. 1 : Architecture d'un processeur PowerPC

On distingue dans cette architecture la présence de deux types de mémoires caches. Une mémoire I-cache (Instruction Cache Unit) destinée à stocker les instructions qui vont être exécutées sur le processeur. Une deuxième mémoire D-cache (Data Cache Unit) destinée à stocker les données qui vont être traitées. On distingue dans la littérature plusieurs initiatives qui visent la définition d'un processeur embarqué reconfigurable [43-45]. Le processus de reconfiguration des processeurs adapte essentiellement l'interaction entre la mémoire cache (données ou instructions) et le processeur. L'accès à la mémoire cache est particulièrement reconnu comme une opération qui consomme énormément d'énergie [46]. Dans ce qui suit, nous allons détailler les aspects micro architecturaux paramétrables dynamiquement de la mémoire cache.

III.4.1. Mémoire cache

Le processeur essaie d'accéder un mot dans le cache avant de passer à sa recherche dans la mémoire principale. En cas d'échec (miss cache), le mot est gardé dans le cache pour un accès futur. En cas de succès (hit cache), la mémoire principale n'est pas accédée par le processeur. La fréquence des succès, relatif au chargement d'un mot à partir de la mémoire cache, ne dépend pas uniquement de la taille du cache mais aussi de l'algorithme exécuté par le contrôleur de cache.

Dans les processeurs embarqués, on distingue deux types de mémoires caches : les mémoires caches de données et celles d'instructions. Les mémoires caches de données regroupent les données traitées par l'application tandis que les mémoires caches d'instructions regroupent les instructions à exécuter par le processeur. L'adressage des mots à l'intérieur d'une mémoire cache suit des propriétés de localité. La localité spatiale [31] est représentée par une répétitivité au niveau d'une séquence de code d'un programme qui s'exécute toujours à l'intérieur de petites zones répétées de mémoire. Le code des blocs répétés correspond généralement à des boucles ou des sous programmes. D'une autre part, on distingue un autre type de localité qui est la localité temporelle caractérisée par des blocs qui s'exécutent en

séquences très proches. Dans ce cas de localité, il y a plus de chance d'accéder à une position de mémoire utilisée récemment qu'à une autre utilisée depuis long temps.

L'opération d'accès mémoire est très pénalisante en termes de performance d'exécution du programme. De ce fait, les performances et la consommation d'énergie d'un processeur dépendent largement du nombre d'échecs de chargement d'un mot à partir du cache (miss cache). L'importance de cette action d'accès mémoire associée à la variabilité de la charge de travail des applications déjà vues dans la section III.3 fait de la capacité [27], de la topologie [28] et des algorithmes de contrôle du cache des applications [29] un paramètre variable qui dépend des besoins des architectures mono ou multi processeurs [30]. La reconfiguration adaptative de la stratégie d'utilisation des mémoires caches est implémentée essentiellement dans des applications caractérisées par un traitement intensif de données comme les applications multimédia [32], [33].

III.4.2. Branchement

Le branchement est une technique de programmation très utilisée pour déterminer conditionnellement le chemin d'exécution d'un programme. Si une condition de branchement n'est pas validée, alors le processeur sélectionne la prochaine instruction dans la mémoire pour être exécutée. Si la condition de branchement est valide, la prochaine instruction à exécuter par le processeur est localisée dans un emplacement mémoire quelconque du cache ou de la mémoire principale. L'instruction de branchement précise l'adresse mémoire de l'instruction à exécuter dans ce cas. Cette instruction fait le choix entre un chemin d'exécution valide et non valide, la direction correcte est déterminée en ligne.

Les problèmes au niveau de l'exécution d'une opération de branchement suite à un défaut de cache pénalisent le système au niveau de la performance d'exécution [39] et de la consommation d'énergie. Afin d'éviter un problème de chargement d'instructions ou de données à partir de la mémoire cache (défaut de cache), des techniques de recherche en ligne de l'instruction à exécuter suite à un branchement ou d'une donnée à traiter appelées 'prefetching' sont établies [34]. Le 'prefetching' est une technique qui consiste à prédire des adresses d'instructions ou de données avant que l'application n'en ait besoin. Ces informations sont stockées dans des registres spécifiques ou dans un niveau élevé du cache. Les techniques de branchement sont classifiées en deux catégories : la prédiction statique et la prédiction dynamique [38]. L'implémentation du prefetching peut être introduite dans le code du programme, il s'agit dans ce cas d'un prefetching logiciel. Dans le cas du prefetching matériel [40], le programme de prédiction est exécuté par un composant du processeur.

- Le prefetching au niveau logiciel consiste à introduire des instructions spécifiques dans le code du programme qui permettent de déterminer les informations à précharger. Un compilateur logiciel peut placer des instructions de prefetching entre les instructions d'un programme, il peut les éliminer en cas de redondance. Dans le cas de l'utilisation de techniques de prefetching logiciel, le système déploie un petit support matériel associé à un grand effort de compilation.
- Le prefetching au niveau matériel est assuré par le microprocesseur. Les performances du prefetching matériel sont nettement meilleures que celles du logiciel. Les performances du prefetching matériel dépendent de l'organisation et du comportement des adresses mémoire à gérer. L'implémentation d'une opération de prefetching matérielle n'impose pas un traitement spécifique au niveau de la compilation.

Dans le cas du prefetching au niveau instruction, le processeur prédit les adresses de branchement en analysant le comportement de l'historique de cette opération. En cas d'erreur

de prédiction, le système exécute un mécanisme de recouvrement pour recommencer les recherches d'instructions de branchement à partir d'un chemin correct. L'opération de recouvrement pénalise d'avantage les performances du système. Ceci nous ramène à introduire des techniques nouvelles de prefetching [35] appliquées surtout pour des applications à traitement compliqué telles que les applications multimédia [36], [37].

Les techniques de prefetching sont appliquées aussi pour prédire le comportement et la localité des données traitées par l'application. Un exemple d'utilisation de ces techniques de prédiction est proposé dans [41] qui présente un système de stockage adaptatif basé sur la prédiction d'accès. Ce système est caractérisé, par rapport à un système de stockage ordinaire, par une faible consommation d'énergie tout en améliorant le temps d'accès à une donnée. D'une autre part, ce système démontre l'avantage d'implémentation d'un système adaptatif sensible à la variation de la charge de travail de l'application. En effet, la variation de la charge de travail à exécuter par une application peut influencer le nombre d'accès mémoire en chargement ou en stockage [47]. Le système de stockage proposé dans [41] assure aussi la validité des données face à une variation de la charge de travail.

Pour le cas d'un prefetching logiciel, la compilation statique hors ligne ne présente pas des résultats efficaces de prédiction. Cette inefficacité est causée par deux critères :

- L'efficacité des résultats de la prédiction dépend largement du comportement du programme exécuté ainsi que de l'organisation des données traitées dans la mémoire cache.
- L'efficacité des résultats de la prédiction dépend largement du nombre d'échecs de prédiction par exécution. L'échec de prédiction dépend lui même des données traitées par l'application.

L'approche du prefetching dynamique a été introduite dans le but de remédier aux deux limitations du prefetching présentées ci-dessus. Dans ce cas, le processus de prefetching insère dynamiquement les instructions de branchement pour corriger le chemin où un échec de prédiction est estimé. Le prefetching dynamique adapte en ligne le comportement de l'exécution d'un programme. Pour effectuer cette tâche d'insertion dynamique, le délai d'adaptation ne doit pas dépasser le temps de changement de phases [42] au cours de l'exécution du programme. Le problème de prefetching demeure plus complexe s'il est implémenté sur une plateforme multi processeurs [48].

III.4.3. Impact sur le CPI

Dans cette section on s'intéresse au prefetching au niveau du cache instruction qui influence énormément les performances du processeur. L'impact du prefetching au niveau instruction est sensé être plus influant sur les performances du processeur que celui au niveau des données compte tenu de deux raisons :

- Les traces des adresses des instructions sont plus faciles à manipuler que les adresses de données. En effet, les adresses instructions sont constituées d'un seul flux de données tandis que les adresses mémoires de données sont composées de deux types d'accès : accès en écriture et en lecture
- La localité spatiale est plus présente pour l'allocation des adresses instructions. Cette localité facilite la tâche de prédiction effectuée par le système de prefetching. De ce fait, les résultats de prefetching pour les adresses des instructions sont plus rigoureux que les résultats de prefetching des données.

Le nombre d'instruction par cycle CPI (Cycles Per Instruction) est la métrique standard pour l'évaluation de la performance d'exécution d'une application sur un processeur. Le CPI est

largement influencé par un échec (cache miss) au niveau du chargement d'une instruction. On distingue d'autres métriques qui aident à évaluer l'influence d'un défaut de cache sur la performance comme le MPI[33] qui représente le nombre d'échecs lors de l'exécution d'une seule instruction et le CPM[34] qui représente le temps perdu suite à un cache miss. On distingue aussi par le CPI_{miss} la moyenne de cycles perdus suite à un cache miss. Dans ce cas, CPI_{miss} est calculé en utilisant l'équation : $CPI_{miss} = CPM * MPI$.

L'application d'une stratégie de prefetching des instructions réduit le MPI contribuant ainsi à la réduction du temps d'exécution non exploité à la suite d'un cache miss. Ceci assure une performance d'exécution plus élaborée en cas d'utilisation d'une stratégie de prédiction au niveau des instructions. Le système de prefetching doit définir une fenêtre de prédiction assez large pour pouvoir éviter un échec de chargement d'une instruction.

III.5. Analyse de la variabilité des charges de travail

La caractérisation de la charge de travail [49], [50] d'un système de traitement informatique est un domaine de recherche qui consiste à analyser le comportement du système durant l'exécution d'une tâche de traitement. Cette analyse permet de prédire les performances du système face à des changements de contexte provoqués par les causes détaillées dans la section III.3. Cette analyse intervient aussi dans la simulation d'un point de vu statistique des processeurs [51] afin de caractériser automatiquement le flux de code qui représente le profile de l'application à implémenter. Une étape de caractérisation de la charge de travail est considérée également nécessaire pour implémenter un système d'échantillonnage efficace des données de test afin de réduire le temps de simulation de l'application [60].

La caractérisation de la charge de travail d'une application est une étape très importante dans le processus de conception d'une architecture efficace implémentant cette application. Cette étape de caractérisation permet de prévoir une implémentation d'architecture reconfigurable. Le flot de conception d'architectures SOPC reconfigurables au niveau système proposé dans [70] est représenté par la figure III.2.

[33] MPI : Misses Per Instruction
[34] CPM : Cycles Per Miss

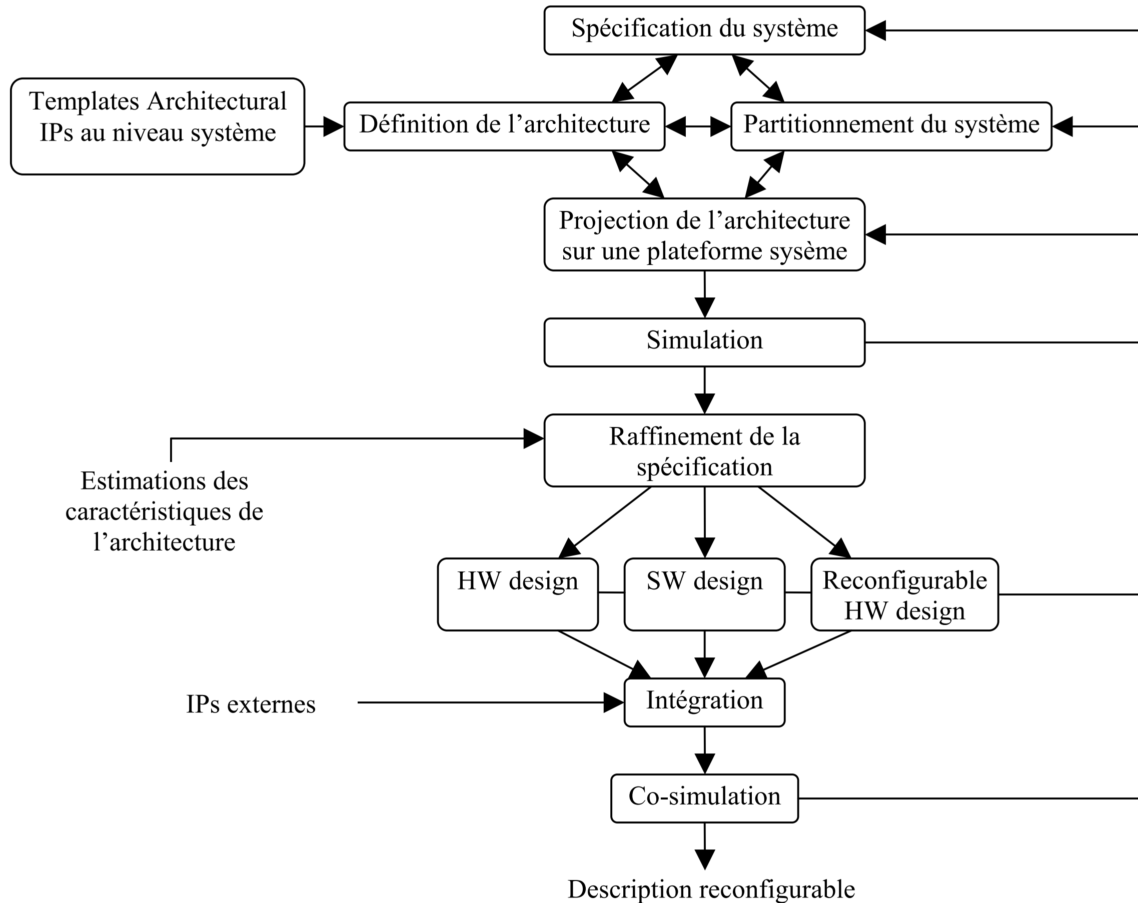


Fig. III. 2 : Flot de conception au niveau système d'architectures reconfigurables [70]

Dans le flot de la figure III.2, le concepteur est invité à caractériser le profile de l'application afin de déterminer les parties reconfigurables du design. La caractérisation du profile de l'application est estimé à travers ses traitements. Une partie matériel reconfigurable est représentée par des accélérateurs qui ne sont pas utilisés à chaque instant jusqu'au maximum de leurs capacités. Elle peut être représentée par un bloc de traitement qui modifie périodiquement sa spécification ou par un module cible d'innovation dans des générations futures. Le flot présenté dans [70] propose un flot de conception itératif d'architectures reconfigurables. Ce flot s'intéresse uniquement à la détection des aspects reconfigurables de l'application.

Les travaux développés dans [71] touchent plus à la reconfiguration dynamique des systèmes sur puces en se basant sur les besoins de ces systèmes. Le flot de la figure III.3 propose l'environnement d'estimation de performances d'architectures reconfigurables.

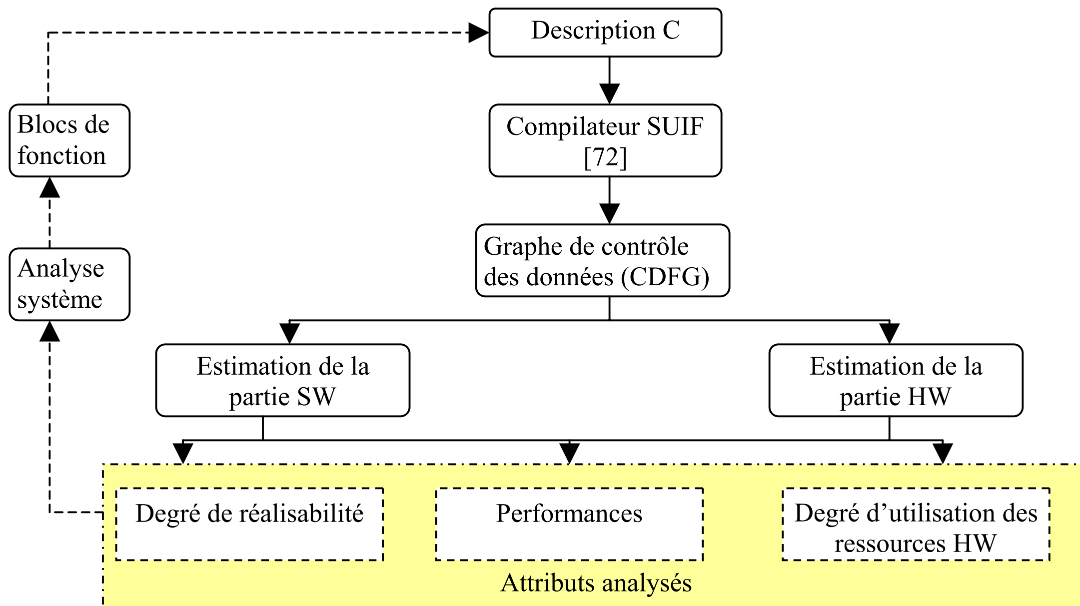


Fig. III. 3 : Environnement d'analyse appliqué pour le développement et l'évaluation d'architectures reconfigurables

Le flot de la figure III.3 génère un ensemble de blocs fonctionnels à partir d'une description C. Un bloc fonctionnel peut être implémenté en SW ou HW. La reconfiguration dynamique de l'architecture est basée sur un système d'ordonnancement dynamique. L'ordonnancement est effectué dans ce cas en se basant sur les estimations de performances des blocs fonctionnels.

III.5.1. Détection et prédiction de phases

La majorité des programmes informatiques comportent des traitements répétitifs. Les recherches menées dans ce contexte affirment qu'une application multimédia par exemple passe 90% du temps de traitement pour exécuter d'une façon répétitive 10% du code. Les architectures avancées proposent de regrouper automatiquement les instructions du code dans des intervalles de temps, qui représentent des périodes d'exécution continue et similaire appelées phases [58]. Une phase est caractérisée par un comportement homogène de ses instructions regroupées dans des ensembles de travail appelés 'working sets'. La détection des différentes phases d'un programme est effectuée en examinant la performance des différentes instructions [51]. La caractérisation du comportement dynamique d'une application intervient afin d'optimiser l'implémentation d'une architecture par l'implémentation d'un processus de reconfiguration dynamique [52], par une étape de compilation guidée [53] ou par équilibrage de la charge de travail dans une plateforme multi processeurs [54].

Pour détecter le changement de phase au cours de l'exécution d'une application, plusieurs métriques ont été utilisées. Ces métriques, évaluées au cours de l'exécution du programme, estiment les performances de l'architecture implémentée. Une variation brusque de l'une de ces métriques indique un changement de phase dans l'exécution du programme. Un système de détection de changement de phases doit opérer en ligne en évaluant plusieurs métriques d'exécution. Il doit également utiliser un espace de stockage limité tout en effectuant des opérations ciblées, rapides et synchronisées avec la rapidité d'exécution du programme. Afin de détecter un changement de phase, on évalue deux types de métriques :

- Métriques dépendant de l'architecture : comme le IPC et le MPI détaillées dans la section III.4.3. On distingue aussi le LSQ[35] qui représente le nombre d'instructions de chargement (Load) et de stockage (Store) insérées dans la pile d'attente durant un intervalle de temps. Les valeurs des métriques dépendant de l'architecture indiquent le degré d'adéquation entre l'architecture implémentée et la charge de travail de l'application durant la phase actuelle. Si les paramètres de l'architecture du système changent, les performances d'exécution de l'application changent aussi. Les métriques dépendant de l'architecture s'intéressent plus à l'évaluation des opérations d'accès mémoire comme le taux de défaut de cache, d'utilisation du bus de cache, de cohérence du cache dans un système multi processeurs et finalement le taux d'efficacité des opérations de prefetching.
- Métriques non dépendant de l'architecture : comme le IPB[36] qui représente le nombre d'instructions exécutées par une opération de branchement. Cette métrique ne dépend pas de l'architecture implémentée. Ce type de métrique est affecté par les techniques de compilation utilisées comme l'exécution spéculative [55]. La majorité des métriques indépendantes de l'architecture sont évaluées directement à partir du code source de l'application. Ces métriques donnent une idée sur la nature, le flot de contrôle, de traitement et sur la prédictibilité des branchements de l'application. A titre d'exemple, la redondance de l'instruction conditionnelle (IF ELSE) dans un code source indique l'irrégularité du flot de contrôle de l'application. Par contre les instructions de boucles indiquent l'implémentation d'un flot de contrôle structuré.

Les systèmes d'évaluation et de détection de phases au cours de l'exécution d'une application sont intégrés, généralement, dans des systèmes reconfigurables dynamiquement afin de contrôler le processus d'adaptation dynamique de l'architecture implémentée. Dans certains cas d'applications qui changent fréquemment de phase, le processus de reconfiguration peut freiner la performance d'exécution d'une application. Pour remédier à ce problème, le processus de reconfiguration adapte le système après l'exécution d'un nombre déterminé τ de phases [56]. La valeur de τ dépend essentiellement du comportement de la charge de travail associée à l'application. La tendance est d'augmenter le nombre de phases exécutées par une seule reconfiguration afin de diminuer le nombre d'adaptations effectuées lors de l'exécution de l'application. Ceci implique une diminution du temps global consacré à l'exécution multiple du processus de la reconfiguration de l'application. Une solution proposée dans [57] présente une méthodologie de classification des différentes phases d'un programme. Cette solution permet de regrouper les différentes phases afin de les exécuter par la même configuration. Les techniques d'analyse de la variabilité au cours de l'exécution d'une application seront détaillées dans le prochain paragraphe.

III.5.2. Analyse de la variabilité

L'analyse des caractéristiques de la charge de travail de l'application est une tâche extrêmement importante dans le processus de conception des architectures embarquées reconfigurables. La caractérisation des applications implémentées sur des plateformes reconfigurables permet de faire varier les caractéristiques des processeurs de la plateforme telles que les paramètres de la micro architecture et la hiérarchie des mémoires de la plateforme. La caractérisation de la charge de travail de l'application participe aussi à l'évaluation efficace des performances de l'architecture implémentée. En pratique, la caractérisation du comportement de l'application est effectuée en évaluant quelques propriétés ou métriques de l'architecture comme le comportement d'accès mémoire, la localité des

[35] LSQ : Load Store Queue
[36] IPB : Instruction Per Branch

données ou des instructions, le comportement du flot de contrôle dans l'application et le niveau de parallélisme entre les instructions (ILP). L'évaluation de ces métriques est effectuée soit hors-puce à travers des analyseurs ou des simulateurs de code ou sur-puce à travers des moniteurs.

Après la détection des métriques représentatives du fonctionnement de l'application. Les concepteurs passent à l'étape d'analyse de ces métriques. Plusieurs caractéristiques sont à analyser telles que la localité des adresses mémoires, l'extraction des traitements redondants et la prédictibilité du comportement de l'application :

- La localité des adresses mémoires : la localité des références traitées par l'application est une propriété qui influence la performance de l'application. En effet, la distribution des données affecte la hiérarchie de la mémoire du système d'où la performance du système. Cette localité affecte aussi la prédictibilité des données à extraire de la mémoire cache [59]. Dans la littérature, on ne trouve pas un modèle générique pour représenter la distribution des données. Idéalement, on définit un index ou une fonction de localité pour chaque application. Cette fonction aide à prédire la performance de la mémoire cache sans accéder à une simulation détaillée.
- L 'extraction des traitements redondants : l'extraction de redondance au niveau des traitements effectués par une application entre dans le cadre de la détection de phases d'exécution d'un programme détaillée dans la section III.5.1. L'extraction du comportement de l'application est effectuée en-ligne, pour les systèmes temps réel, en utilisant des moniteurs de performance. Ces moniteurs fixent une période d'échantillonnage. Au bout de chaque période, le moniteur réalise une extraction de performance. L'exécution multiple de la même application, en utilisant différents exemples d'application, génère une différence au niveau de la trace de performances. Cette différence est causée par les effets variables d'entrée/sortie de la mémoire. La caractérisation des phases de plusieurs exécutions d'une application doit mener à la même distribution de phases [58]. On distingue deux méthodologies d'extraction de phases. La première méthode effectue une analyse des performances extraites par le moniteur afin de détecter, à travers plusieurs techniques de classification, les similarités entre les différentes traces d'exécution. La deuxième méthodologie consiste à détecter uniquement les transitions entre les différentes phases d'exécution. Contrairement à la première méthode, cette méthodologie permet d'éviter l'utilisation d'algorithmes complexes de classification appliqués sur des traces de performances de grande taille.
- La prédictibilité du comportement de l'application : la reconfiguration dynamique efficace d'architectures complexes impose la mise en œuvre d'un système d'analyse et de prédiction du comportement de l'application. Pour mieux représenter son comportement, un système est approché par un modèle non linéaire tel que les réseaux de neurones [60], [61]. La prédictibilité de la trace de performances d'exécution de l'application est déterminée par la taille de cette trace [62] ou par le niveau de corrélation entre les différentes phases d'exécution. Plusieurs techniques ont été utilisées dans la littérature pour réduire la complexité des traces d'exécution [63], [64].

III.6. Conclusion

La complexité des applications à implémenter sur des systèmes sur puces associée aux contraintes strictes de traitement temps réel, de faible consommation d'énergie et de maximum de fonctionnalité incite l'apparition de nouvelles tendances de conception. L'initiative la plus marquante est l'introduction de la notion d'architectures adaptatives implémentées sur des plateformes reconfigurables. L'adaptation de ces architectures passe obligatoirement par une étape de caractérisation de la charge de travail au cours de l'exécution de l'application. Dans ce chapitre, nous avons exposé cette approche en se focalisant sur les causes de la variabilité de la charge de travail ainsi que sur les métriques qui caractérisent cette charge. Nous avons également présenté quelques travaux exposant des méthodologies d'évaluation et d'exploitation des caractéristiques d'une application.

Dans la première section de ce chapitre, nous avons présenté les caractéristiques qui favorisent l'implémentation d'une application sur une architecture reconfigurable. Une architecture est fortement reconfigurable si elle présente une variabilité considérable et assez fréquente de sa charge de travail durant l'exécution de l'application. Dans la deuxième section du chapitre, nous avons détaillé les causes de cette variabilité. La variabilité de la charge de travail est dirigée par trois facteurs qui sont la variation de la nature et de la grandeur des données traitées, la variation de l'environnement d'exécution du système et par la diversité des applications exécutées sur la même plateforme reconfigurable. Ensuite, nous avons exposé, dans la troisième section de ce chapitre, les différents aspects reconfigurables de la micro architecture d'un processeur. Dans cette partie nous avons détaillé les aspects architecturaux reconfigurables au niveau de l'organisation et du contrôle de la mémoire cache. Finalement nous avons présenté quelques travaux d'analyse de la variabilité de la charge de travail telle que la détection et la prédiction de phases d'exécution d'une application.

Il est clair que les applications implémentées dans les systèmes sur puce d'aujourd'hui présentent de plus en plus de comportement variable. Ceci du fait de l'émergence des applications multimédia accessibles à distance. Les aspects reconfigurables au niveau de la micro architecture sont ajustés pour adapter les performances de l'architecture aux besoins de l'application. L'adaptation de ces paramètres optimise les performances d'une application purement logicielle mais pas un système mixte qui contient des composants logiciels et d'autres matériels.

L'introduction d'un caractère adaptatif dans la composition de l'architecture d'un système embarqué suppose l'insertion d'un contrôleur de reconfiguration dans la plateforme. Ce contrôleur aide à caractériser les performances de l'architecture et de décider de la nouvelle reconfiguration. Dans le prochain chapitre, nous allons étudier des exemples concrets de variabilité. Ensuite, nous allons discuter l'application d'un modèle neuronal pour décrire cette variabilité.

III.7. Références

[1] Liang. J, Laffely. A, Srinivasan. S and Tessier. R, 'An architecture and compiler for scalable on-chip communication', IEEE Transactions on very large scale integration (VLSI) systems, Vol.12, No. 7, 2004, pp. 711- 726.

[2] Eeckhout. L, Sundareswara. R, Yi. J, Lilja. D and Schrater, 'Accurate statistical approaches for generating representative workload compositions', IEEE, 2005, pp. 56-66.

[3] Maxiaguine. A, Liu. Y, Chakraborty. S, Ooi. W, 'Identifying "representative" workloads in designing MpSoC Platforms for media processing', IEEE, 2004, pp. 41-46.

[4] Rutten. M et al., 'A Heterogeneous Multiprocessor Architecture for Flexible Media processing', IEEE Transactions on Design and Test of Computers, Vol. 19, No. 4, 2002, pp. 39-50.

[5] Varatkar .G.V and Marculescu. R, 'On-chip traffic modelling and synthesis for MPEG2 video applications', IEEE Transactions on Very Large Scale Integration (VLSI) systems, 2004, pp. 108-119.

[6] Rutten. M. J, Eijndhoven. J, Pol. E, 'Robust media processing in a flexible and cost-effective network of multi-tasking coprocessors', Proceedings Euromicro Conference on Real Time Systems, 2002, pp. 223-230.

[7] Lu. Y, Benini. L, Micheli. G.D, 'Dynamic frequency scaling with buffer insertion for mixed workloads', IEEE Transactions on Computer Aided Designs of Integrated Circuits and Systems, Vol. 21, No. 11, 2002, pp. 1284-1305.

[8] Im. C, Ha. S, 'Dynamic Voltage Scaling for Real-time multi-task Scheduling using buffers', Proceedings conference on Languages, Compilers, and Tools for Embedded Systems, 2004, pp. 88-94.

[9] Stayanarayanan, 'Pervasive Computing: Vision and Challenges', IEEE PCM, 2001, pp. 10-17.

[10] Chesire. M, Wolman. A, Volker. G, Levy. H, 'Measurement and Analysis of a Streaming-Media Workload', Proceedings USENIX Symposium Internet Technology Systems, 2001, pp. 1-12.

[11] Velso. E, Almeida. V, Meira. W, Bestavors. A, Jin. S, 'A Hierarchical Characterization of a Live Streaming Media Workload', IEEE/ACM Transactions on networking, Vol. 14, No. 1, 2006, pp. 133-147.

[12] Raman. B, Chakraborty. S, Ooi. W, 'Meeting CPU constraints by delaying playout of multimedia tasks', Proceedings ACM International Workshop on Network and Operating System Support for Digital Audio and Video, 2005, pp. 165-170.

[13] Chen. G, Kandemir. M, Sezer. U, 'Configuration-Sensitive Process Scheduling for FPGA-based Computing Platforms', Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2004.

[14] Liu. Y, maxianguine. A, Chakraborty. S, Ooi. W, 'Processor Frequency Selection for SOC platforms for multimedia applications', Proceedings IEEE International Real Time System Symposium, 2004, pp. 336-345.

[15] Vuletic. M, Pozzi. L, Ienne. P, 'Virtual Memory Window for Application-specific Reconfigurable Coprocessors', Proceedings IEEE Design Automation Conference, 2004, pp. 948-953.

[16] Catthoor. F, Wuytack. S, De Greef. E, Balasa. F, Nachtergaele. L, Vandescappelle. A, 'Custom Memory Management Methodology: Exploration of Memory organisation for Embedded Multimedia System Design', Kluwer Academic Publishers, 1998.

[17] Zhu. H, Luican. I, Balasa. F, 'Memory Size Computation for Multimedia Processing Applications", Proceedings Asia and South Pacific Design Automation Conference ASP-DAC'06, 2006, pp. 802-807.

[18] Ying. T, Malani. P, Qiu. Q, Wu. Q, 'Workload Prediction and Dynamic Voltage Scaling for MPEG Decoding', Proceedings Asia and South Pacific Design Automation Conference ASP-DAC'06, 2006, pp. 911-916.

[19] Salembier. P, Smith. J, 'MPEG-7 Multimedia Description Schemes', IEEE Transactions on Circuits and Systems for Video Technology, Vol. 11, No. 6, 2001, pp. 748-759.

[20] Im. C, Ha. S, Kim. H, 'Dynamic voltage scheduling with buffers in low-power multimedia applications', ACM Transactions on Embedded Computing Systems, Vol. 3, 2004, pp. 686-705.

[21] Choi. K, Dantu. K, Cheng. W, Pedram. M, 'Frame-based dynamic voltage scaling for a MPEG decoder', Proceedings on Computer Aided Design, 2002, pp. 732-737.

[22] MacCreary. S, Claffy. K, 'Trends in Wide area IP traffic patterns, a view from Ames internet exchange', ITC specialist seminar on IP Traffic Modeling, Measurment and Management, 2000.

[23] Chan. S, Tobagi. F, 'Caching Schemes for distributed video services', Proceedings IEEE International Conference on Communications, 1999, pp. 994-999.

[24] Reisslein. M, Ross. K, 'High-Performance prefetching protocols for VBR prerecorded video', IEEE Transactions on Network, Vol. 12, 1998, pp. 46-55.

[25] Almeroth. K, 'Adaptive workload-dependent scheduling for large-scale content delivery systems', IEEE Transactions on Circuits and Systems For Video Technology, Vol. 11, No. 3, 2001, pp. 426-439.

[26] www-03.ibm.com/chips/power/powerpc.

[27] Cucchiara. R, Piccardi. M, Prati. A, 'Neighbor Cache Prefetching for Multimedia Image and Video Processing', IEEE Transactions on Multimedia, 2004, pp. 539-552.

[28] Peir. J, Lee. Y, Hsu. W, 'Capturing dynamic memory reference behaviour with adaptive cache topology', Proceedings International Conference on Architectural support for programming language and operating systems, 1998, pp. 240-250.

[29] Petrov. P, Orailoglu. A, 'Towards effective embedded processors in codesigns: customisable partitioned caches', Proceedings ACM/CODES, 2001, pp. 79-84.

[30] Carballeira. F, Carretro. J, Calderon. A, Perez. J, Garcia. D, 'An Adaptive Cache Coherence Protocol Specification for Parallel Input/Output Systems', IEEE Transactions on parallel and distributed Systems, Vol. 15, No.6, 2004, pp. 533-545.

[31] Johnson. T, Hwu. W, 'Run time adaptive cache hierarchy management via reference analysis', Proceedings of International Symposium on Computer Architecture, 1997, pp. 315-326.

[32] Ranganathan. P, Adve. S, Jouppi. N, 'Reconfigurable caches and their application to media processing', Proceedings International Symposium on Computer Architecture, 2000, pp. 214-224.

[33] Albonesi. D, ‘Selective cache ways: On-demand cache resource allocation’, Proceedings MICRO, 1999, pp. 248-259.

[34] Wiel. V, Lilja. J, ‘When caches aren’t enough: data prefetching techniques’, IEEE Transactions on computer, Vol. 30, No. 7, 1997, pp. 23-30.

[35] Lee. J, Jeong. S, Kim. S, ‘An intelligent cache system with hardware prefetching for hihg performance’, IEEE Transactions on Computers, Vol. 52, No. 5, 2003, pp. 607-616.

[36] Piccardi. R, Prati. M, ‘Neighboor cache prefetching for multimedia image and video processing’, IEEE Transactions on Multimedia, Vol. 6, No. 4, 2004, pp. 539-552.

[37] Lee. J, Park. C, Ha. S, ‘Memory access pattern analysis and stream cache design for multimedia applications’, Proceedings Asia and South Pacific Design automation conference, 2003, pp. 22-27.

[38] Olivier. R, Teller. J, ‘Dynamic and adaptive cache prefetch policies’, Proceedings International Conference on Performance, Computing and Communications, 2000, pp. 509-515.

[39] Lau. J, Sampson. J, Perelman. E, Harnerly. G, Calder. B, ‘The strong correlation between code signatures and performances’, Proceedings IEEE International Symposium on Performance Analysis of Systems and Software, 2005, pp. 236-247.

[40] Zucker. D, Flynn. M, Lee. R, ‘A comparison of hardware prefetching techniques for multimedia benchmarks’, Proceedings International Conference on Multimedia Computing and Systems, 1996, pp. 236-244.

[41] Rybczynski. J, Long. D, Amer. A, ‘Expecting the unexpected: Adaptation for predictive energy conservation’, Proceedings StorageSS, 2005, pp. 130-134.

[42] Lau. J, Schoenmackers. S, Calder. B, ‘Transition phase classification and prediction’, Proceedings International Symposium on High-Performance Computer Architecture, 2005, pp. 278 – 289.

[43] Petrov. P, Orailoglu. A, ‘Customizable embedded processor architectures’, Proceedings Euromicro Symposium on Digital System Design, 2003, pp. 468-475.

[44] Ghosh. A, Givargis. T, ‘Cache optimisation for embedded processor cores: an analytical approach’, Proceedings ACM/ International Conference on Computer Aided Design, 2003, pp. 342-347.

[45] Ramaswamy. S, Yalamanchili. S, ‘Customizable fault tolerant caches for embedded processors’, Proceedings IEEE International Conference on Computer Design, 2006.

[46] Inoue. K, Ishihara. T, Murkami. K, ‘Way-predicting set-associative cache for high-performance and low energy consumption’, Proceedings ISLPED, 1999, pp. 273-275.

[47] Cook. J, Olivier. R, Johnson. E, ‘Examining performance differences in workload execution phases’, Proceedings International Workshop on Workload Charecterization, 2001, pp. 82-90.

[48] Ganusov. I, Burtscher. M, ‘Future execution: A hardware prefetching technique for chip multiprocessors’, International Conference on Parallel Architectures and Compilation Techniques, 2005, pp. 350-360.

[49] John. L, Vasudevan. P, Sabarinathan. J, ‘Workload characterization: Motivation, Goals and Methodology’, Proceedings Workload Characterization: Methodology and Case Studies, 1998, pp. 3-14.

[50] Bradfort. J, Fortes. J, 'Performance and memory-access characterization of data mining applications', Proceedings Workload Characterization: Methodology and case study, 1998, pp. 49.

[51] Sherwood. T, Sair. S, Calder. B, 'Phase tracking and prediction', Proceedings International Symposium on Computer Architecture, 2003, pp. 336-347.

[52] Dhodapkar. A, Smith. J, 'Managing multi-configuration hardware via dynamic working set analysis', Proceedings International Symposium on Computer Architecture, 2002, pp. 233-244.

[53] Merten. M, Trick. A, Barnes. R, Nystrom. E, George. C, Gyllenhaal. J, Hwu. W, 'An architectural framework for run time optimization', IEEE Transactions on Computers, Vol. 50, No. 6, 2001, pp. 567-589.

[54] Paulo. J, Almeida. A, Van sinderen. M, Pires. L, Wegdam. M, 'Platform-independent dynamic reconfiguration of distributed applications', Proceedings International Workshop on Future Trends of Distributed Computing Systems, 2004, pp. 286-291.

[55] Gupta. S, Saviou. N, Kim. S, Dutt. N, Gupta. R, Nicoulau. A, 'Speculative techniques for high level synthesis of control intensive designs', Proceedings Design Automation Conference, 2001, pp. 269-272.

[56] Li. K, 'Optimal period of workload redistribution for dynamic bulk synchronous computations in heterogeneous computing systems', Proceedings International Parallel and distributed Processing Symposium, 2004, pp. 242.

[57] Lau. J, Schoenmackers. S, Calder. B, 'Transition phase classification and prediction', Proceedings International Symposium on High-Performance Computer architecture, 2005, pp. 278-289.

[58] Isci. C, Martonosi. M, 'detecting recurrent phase behaviour under real-system variability', Proceedings IEEE International Workload Characterization Conference, 2005, pp. 13-23.

[59] Sanchez. F, Gonzalez. A, 'Data locality analysis of the SPEC95', Workshop on Digest of Performance Analysis and its Impact on Design, 1998, pp. 78-84.

[60] Yoo. R, Lee. M, Chow. K, Lee. H, 'Constructing a non-linear model with neural networks for workload characterization' Proceedings International Symposium on Workload Characterization, 2006, pp. 150-159.

[61] Owens. A.J, 'Empirical Modeling of Very Large Data Sets Using Neural Network', Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, Vol. 6, 2000, pp. 302 - 307.

[62] Todi. R, 'Speclite: Using representative samples to reduce spec cpu2000 workload', Proceedings IEEE International Workshop on Workload Characterization, 2001, pp. 15-23.

[63] Sang-woon. K, Oommen. B, 'Enhancing prototype Reduction schemes with recursion: A Method Applicable for Large Data Sets', IEEE Transactions on Systems, Man and cybernetics, Vol. 34, No. 3, 2004, pp. 1384-1397.

[64] Zhong. S, Khoshgoftaar. T, Seliya. N, 'Analyzing Software Measurement Data with Clustring Techniques', IEEE Transactions on Intelligent Systems, 2004, pp. 20-27.

[65] Prudencio. R, Indrusiak. L, Glesner. M, 'An efficient Hardware Implementation of a Self-Adaptable Equalizer for WCDMA Downlink UMTS Standard', Proceedings Emerging VLSI Technologies and Architectures, 2006, pp. 5.

[66] Kowk. T, Kowk. Y, 'On the Design of a Self-Reconfigurable SoPC Based Cryptographic Engine', Proceedings International Conference on Distributed Computing Systems Workshops, 2004, pp. 876-881.

[67] Hung. M, Chen. Y, Cheng. J, Ho. R, Chen. G, 'Development Scheme of SoPC-Based Reconfigurable Controllers', Proceedings Euromicro Conference on Digital System Design, 2005, pp. 364-371.

[68] Sedcole. P, Cheung. P, 'parametric yield in FPGAs due to within-die delay variations: a quantitative analysis', Proceedings ACM International Symposium on Field Programmable Gate Arrays, 2007, pp. 178-187.

[69] Hallscmid. P, Saleh. R, 'Fast Configuration of an Energy-Efficient Branch Predictor', Proceedings Emerging VLSI Technologies and Architectures, 2006, pp. 289.

[70] Pelkonen. A, Masselos. K, Cupak. M, 'System-Level Modeling of Dynamically Hardware with SystemC', Proceedings of the International Parallel and Distributed Processing Symposium, 2003, pp. 8.

[71] Qu. Y, Tiensyrja. K, Soininen. J, 'SystemC-based Design Methodology for Reconfigurable System on Chip', Proceedings of the Euromicro Conference on Digital System Design, 2005, pp. 364-371.

[72] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennesy, 'SUIF: An infrastructure for research on parallelizing and optimizing compilers', ACM SIGPLAN Notices, Vol. 29, No. 12, 1994, pp. 31-37.

IV. Méthodologie d'implémentation de systèmes sur puce reconfigurables à base d'estimateurs de performances

IV.1. Introduction

Un SOPC est un système embarqué complexe qui comporte plusieurs composants intégrés sur la même plateforme. Les composants logiciels et matériels, qui constituent ce système embarqué, appartiennent à plusieurs niveaux de granularité (ou résolution). Dans ce contexte, un niveau de granularité est défini comme un palier qui regroupe plusieurs composants ayant des caractéristiques en commun. Dans le cadre des travaux élaborés dans cette thèse, nous allons aborder le cas des systèmes sur puce reconfigurables, au niveau système, à grain large, c'est à dire, les systèmes qui ne font pas appel à une configuration de la logique des composants. L'étape d'adaptation d'un tel système est effectuée en ligne en reconfigurant les différents composants du système selon plusieurs niveaux de résolution logicielle (reconfiguration sensible au comportement des adresses instructions) et matérielle (reconfiguration sensible à la performance des modules et des applications matérielles).

Le comportement dynamique d'une trace d'exécution, décrivant la performance d'un composant, dépend largement de la nature et du niveau de granularité du composant. La composition d'un SOPC, décrit au niveau système, nous a permis de détecter trois niveaux de granularité qui représentent les applications, les modules et les instructions (Fig. IV.1). De ce fait, l'étape d'analyse et de caractérisation de la charge de travail d'une application en cours d'exécution doit adopter des techniques différentes compatibles avec les différents niveaux de granularité. Cette étape d'analyse de la charge de travail du système doit nous permettre de définir l'architecture du SOPC reconfigurable.

Dans le premier chapitre de cette thèse, nous avons présenté les étapes de conception de systèmes sur puces en s'intéressant aux méthodologies d'implémentation de systèmes adaptés en ligne selon l'évolution de la charge de travail de l'application. Dans le deuxième chapitre de cette thèse, nous avons exposé les différentes raisons de variabilité de la charge de travail de l'application. Nous avons également exposé les différents travaux qui ont exploité cet aspect variable. Ce chapitre introduit la méthodologie proposée pour l'implémentation de systèmes reconfigurables multi résolutions c'est à dire qui agit sur des composants appartenant à différents niveaux de granularité. Cette méthodologie est applicable à un niveau haut de description qui est le niveau système. La méthodologie de reconfiguration présentée se base sur des estimations de performances prédites par un réseau de neurones récurrent appliqué à la prédiction des séries temporelles.

Ce chapitre est organisé comme suit. En premier lieu, nous allons présenter la décomposition adoptée d'un système embarqué selon plusieurs niveaux de résolutions à grains larges. Ensuite nous allons détailler la méthodologie adoptée pour implémenter un système sur puce reconfigurable basé sur la prédiction de la charge de travail multi résolutions c'est à dire selon plusieurs niveaux de granularité. Dans la troisième partie, nous allons présenter quelques techniques de prédiction de séries temporelles notamment à base de réseaux de neurones récurrents. Nous allons exposer également quelques techniques de prétraitement appliquées pour simplifier les dynamiques des séries à prédire.

IV.2. Décomposition multi résolutions et multi composants d'un SOPC reconfigurable

La synthèse automatique d'architectures embarquées réactives est un sujet très traité par les travaux de recherche récents [29-32]. Dans cette optique, les organismes de recherche ainsi que les industriels se concentrent sur les techniques l'exploration d'un espace d'architectures reconfigurables très vaste [33]. L'objectif de cette exploration est de proposer des méthodologies de conception automatique d'applications implémentées sur les parties matérielle, logicielle ou sur des plateformes mixtes associant flexibilité et hautes performances. Le terme architecture reconfigurable englobe un espace de solutions de taille importante délimité par les processeurs généralistes et les FPGAs.

L'objectif de cette thèse est de proposer un modèle de conception d'architectures reconfigurables dynamiquement au niveau système. Ce modèle offre une meilleure densité de calcul par rapport aux architectures statiques existantes tout en optimisant les coûts de réalisation de cette architecture reconfigurable. Il s'agit d'exploiter la flexibilité de l'application à implémenter afin de contrôler ses ressources tout en garantissant un certain niveau de qualité de service QOS (Quality Of Service) par le design.

IV.2.1. Architecture d'un SOPC reconfigurable au niveau système

Le schéma général d'une telle architecture au niveau système peut être décrit par la figure IV.1.

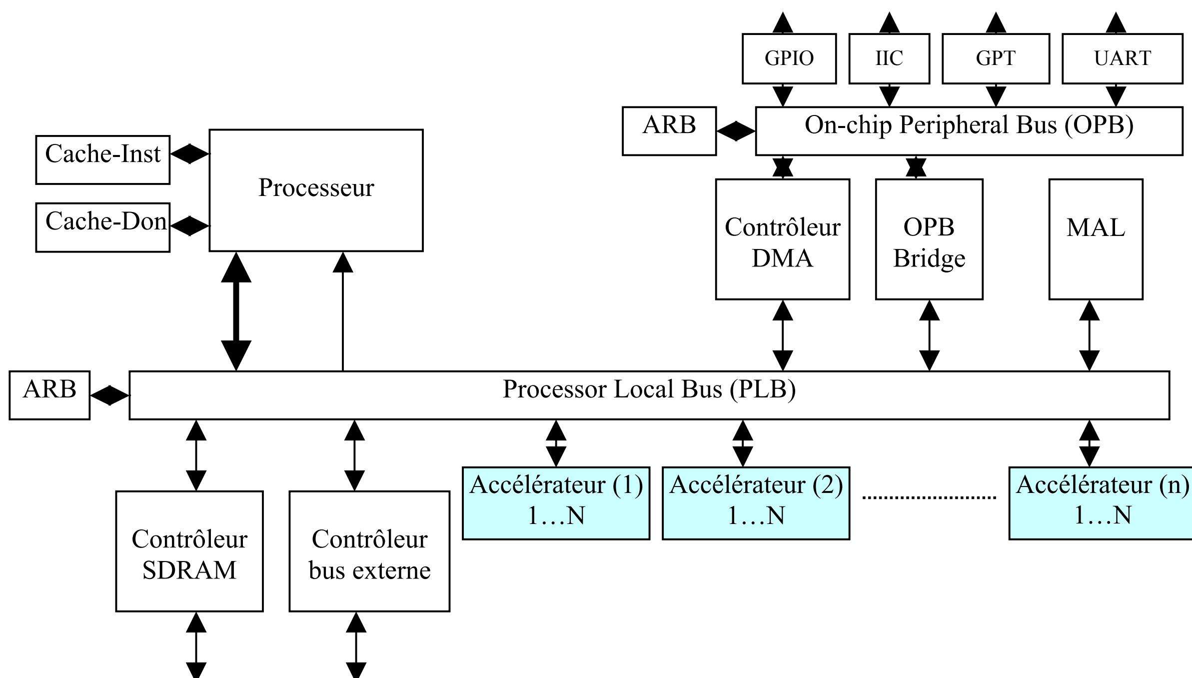


Fig. IV. 1: Architecture standard d'une description haut niveau d'un système sur puce reconfigurable

Dans cette figure, nous présentons une plateforme SOPC comprenant des composants de base tels que les bus de communication et les cœurs de processeur. A titre d'exemple, nous avons présenté, dans le schéma de la figure IV.1, les bus PLB[37] et OPB[38] définis dans la bibliothèque de CoreConnect d'IBM [1]. La partie logicielle du système sera exécutée sur un ou plusieurs cœurs de processeur embarqués sur la plateforme. Plusieurs descriptions d'accélérateurs matériels peuvent être implémentées sur la plateforme. La communication

[37] PLB : Processor Local Bus
[38] OPB : On-chip Peripheral Bus

entre les deux parties logicielle et matérielle est assurée à travers d'un ou plusieurs bus de communication. La reconfiguration dynamique du système embarqué affecte tous les composants de l'architecture. Afin de parvenir à définir un système sur puce réactif, beaucoup de problèmes sont soulevés.

- Quels sont les composants flexibles du système qui assurent un gain au niveau du coût d'implémentation de l'architecture reconfigurable? En effet en analysant un système sur puce, nous remarquons la présence de plusieurs niveaux de granularité regroupant plusieurs composants logiciels et matériels dépendants.
- Que doit-on reconfigurer dans un design ? doit-on agir sur les bus de communication, sur la décomposition de l'architecture ou sur la présence, la duplication ou l'élimination de certaines applications qui sont utilisées, très utilisées ou qui ne sont pas utilisées à un instant précis ?
- Comment assure-t-on un contrôle efficace du design malgré la diversité de la nature des composants implémentés? Quelle est la technique qu'on doit implémenter pour assurer le contrôle de la reconfiguration du design?

IV.2.2. Décomposition multi résolutions d'un SOPC

En analysant la structure d'une architecture embarquée, nous distinguons la présence de plusieurs composants de traitement qui constituent le design. Dans la figure IV.2, nous proposons une décomposition des modules d'une architecture embarquée en différents niveaux de granularité.

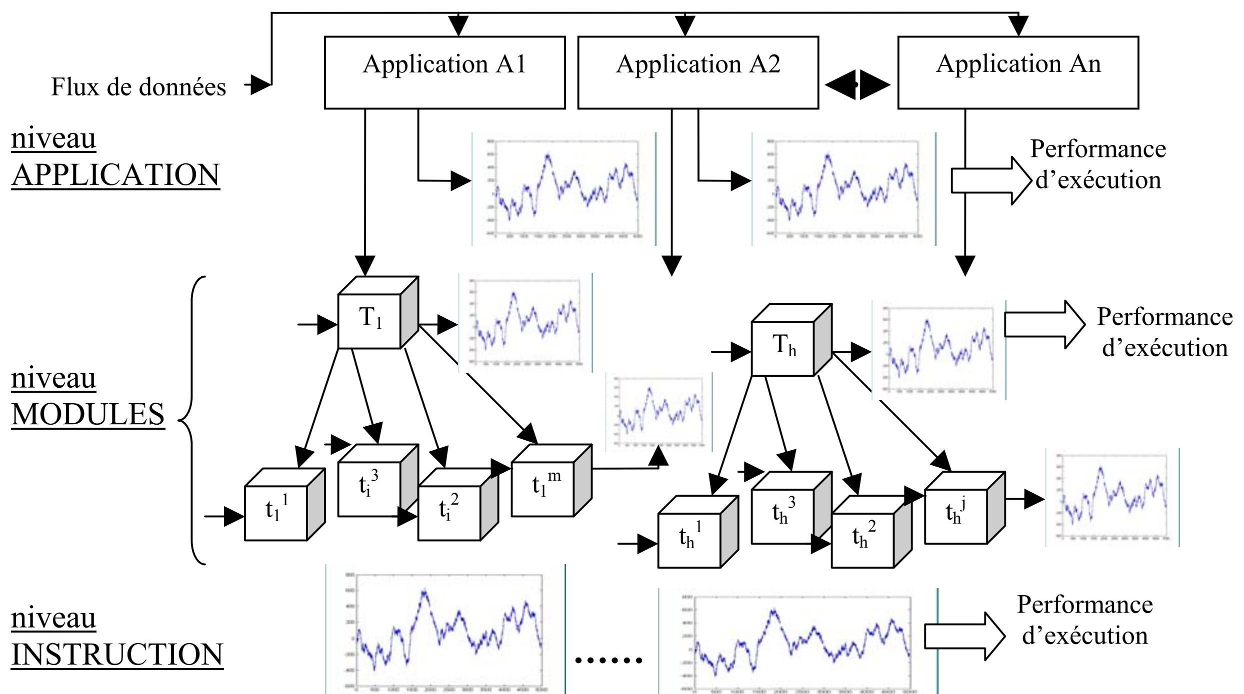


Fig. IV. 2 : Niveaux de granularité d'une architecture embarquée au niveau système

Cette architecture comporte plusieurs applications co-existant sur la même plateforme. La communication entre les différentes applications est assurée par des bus. Chaque application est composée d'un ou plusieurs modules communicants à travers des signaux de communication ou à travers des mémoires partagées. Le plus haut niveau de décomposition est le niveau application. Dans ce niveau, nous regroupons toutes les applications {A1, A2, …, An} logicielles, matérielles ou mixtes exécutées sur un SOPC. Chaque application peut

être décomposée en plusieurs modules {T1, T2, …, Th} dépendants ou indépendants. Finalement, chaque module logiciel est représenté par une description séquentielle d'instructions. Le niveau instruction représente le plus bas niveau de décomposition. La performance d'exécution dans ce niveau est contrôlée à travers l'analyse du comportement de plusieurs métriques d'exécution déjà vues dans la section III.4. L'information détectée à partir des instructions reflète notamment la variation de la performance d'un module suite à un changement de phase d'exécution.

Dans les travaux réalisés dans cette thèse, nous visons l'introduction d'une méthodologie de reconfiguration dynamique multi résolutions et à grains larges des ressources de traitement d'une plateforme SOPC. Chaque type d'entité de traitement (Ai, Tj), représenté dans la figure IV.2, admet un flux de données à traiter en entrée. Suite à l'excitation d'un module de traitement quelconque par une nouvelle entrée, chaque entité effectue le traitement désiré pendant un délai de temps variable. Le comportement de chaque composant de l'architecture est représenté par une série temporelle qui représente les performances de traitement d'une tâche au cours du temps. La représentation multi niveaux et multi composants (Fig. IV.2) de l'architecture caractérise la variation de la charge de travail de l'application ce qui aide à mieux contrôler le processus de reconfiguration.

IV.3. Méthodologie de reconfiguration multi résolutions

Pour implémenter un système reconfigurable, nous proposons une méthodologie qui se base sur l'introduction d'un ou plusieurs systèmes de prédiction de performance. Cette méthodologie agit sur différents composants appartenant à des niveaux distincts de granularité. En effet, les caractéristiques statistiques des métriques de performances changent selon les niveaux de granularité. A titre d'exemple, les taux de variance pour les métriques extraites au niveau des instructions sont beaucoup plus élevés que ceux extraites à partir du niveau des applications. Cette diversité au niveau des caractéristiques des métriques oblige le concepteur à adopter plusieurs modèles de prédiction associés à chaque niveau de granularité. Afin de mieux contrôler la reconfiguration de l'architecture, nous proposons d'associer un système de prédiction de performances à chaque composant qui possède un comportement variable. Le composant est appelé dans ce cas 'composant potentiellement reconfigurable'. Le schéma de la figure IV.3 expose le flot adopté pour implémenter un SOPC reconfigurable.

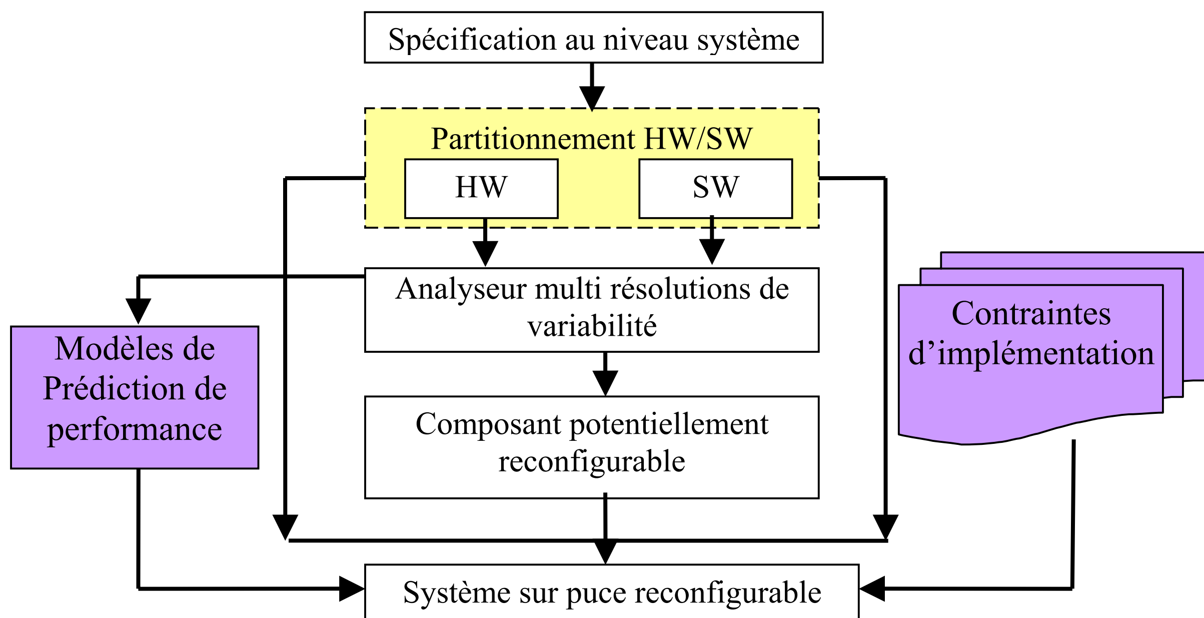


Fig. IV. 3 : Flot d'implémentation d'un système sur puce reconfigurable

Le flot d'implémentation d'un SOPC reconfigurable démarre à partir d'une description fonctionnelle du système sur puce à composition statique. Une fois le partitionnement effectué, nous sommes invités à analyser la variabilité de la performance de chaque composant logiciel ou matériel de l'architecture. Ensuite, nous déterminons les applications et les tâches à performance hautement variable. Une fois déterminées, nous associons à chaque composant potentiellement reconfigurable un prédicteur capable d'estimer la variation future de la performance d'exécution. Finalement, nous introduisons un caractère adaptatif dans le fonctionnement de l'architecture en implémentant un système de décision. Le système de décision fixe la composition de l'architecture en se basant sur les performances estimées. Les valeurs de performance prédites ainsi que les contraintes d'implémentation, telles que les contraintes de traitement temps réel, de consommation d'énergie et de surface, guident le processus de reconfiguration de l'architecture. Dans la suite, nous détaillons les étapes de ce flot dans lesquelles nous analysons la variabilité de chaque composant de l'architecture et nous implémentons l'aspect dynamique dans le fonctionnement d'un système sur puce.

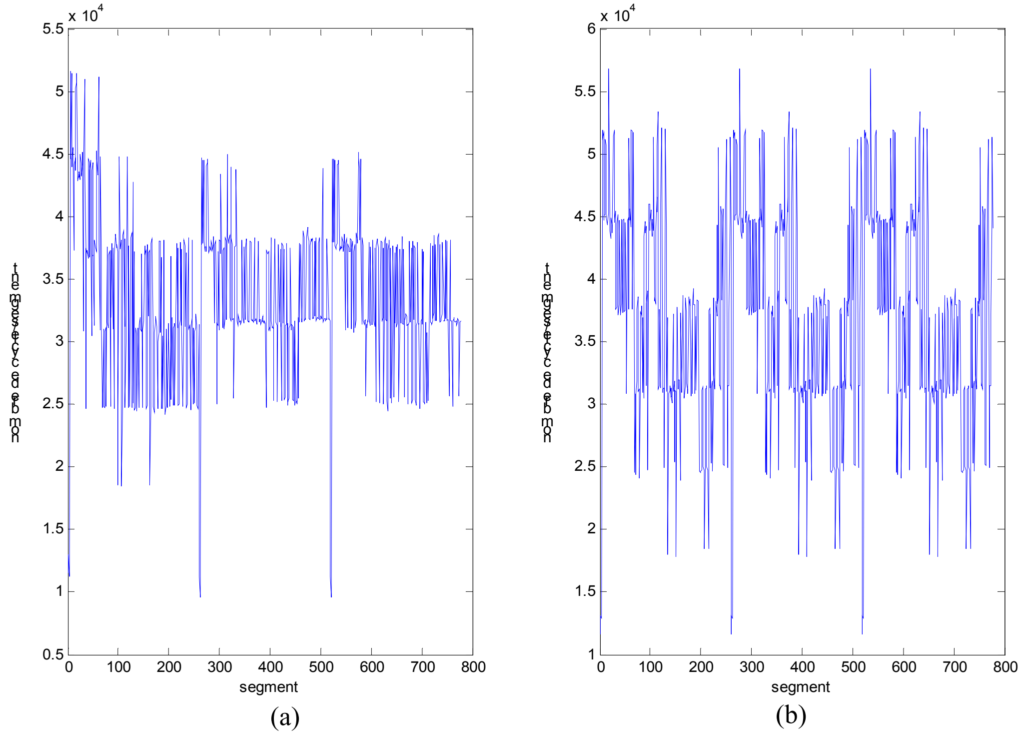
IV.3.1. Analyse multi niveaux de la variabilité

La variation de la performance d'exécution d'une tâche sur un composant matériel ou logiciel est guidée par plusieurs aspects vus dans la section III. 3. Dans cette section, nous allons présenter quelques exemples d'application qui génère un comportement variable. Nous allons également exposer les caractéristiques des traces d'exécution extraites durant l'exécution de l'application.

IV.3.1.1. Variabilité au niveau application

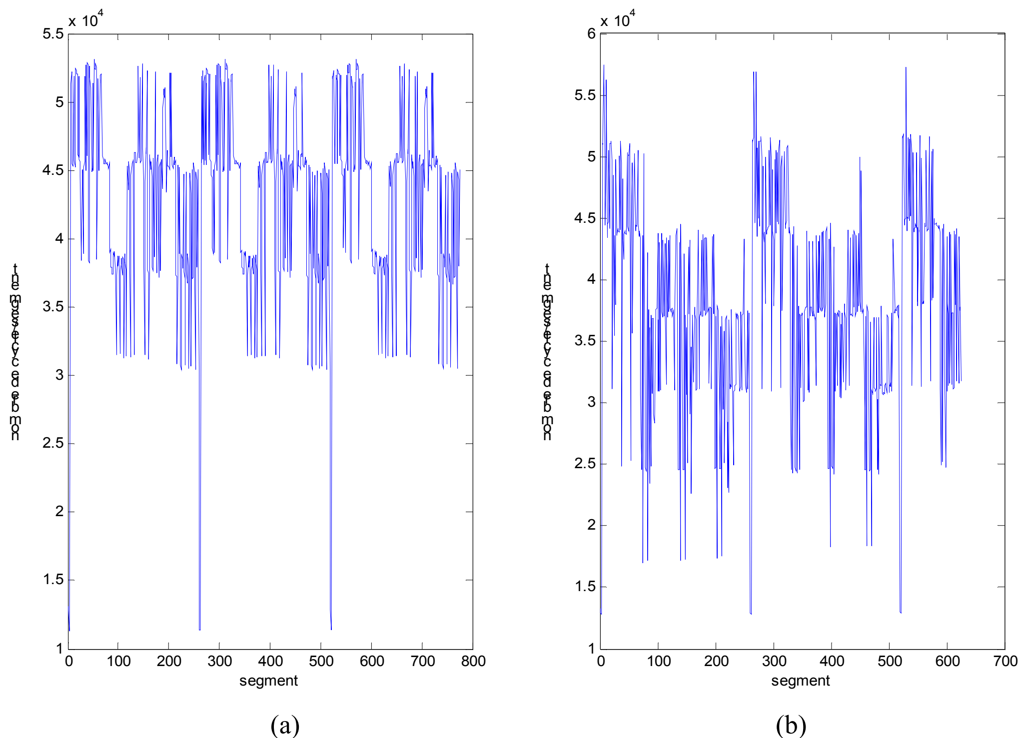
La variabilité au niveau des performances d'une application est testée en analysant le cas d'une implémentation matérielle d'une architecture de traitement multimédia. Le composant matériel sélectionné est un accélérateur qui représente le codeur entropique de la chaîne de compression d'images fixes JPEG2000 [2]. Nous détaillerons l'architecture du codeur dans la section VI.4. Nous présentons dans les figures IV.4 et IV.5, la variation de performance de codage exprimée en nombre de cycles nécessaires pour coder un segment d'image. La figure IV.4 expose les traces de performances des images 'Lena.ppm' et 'Barbara.ppm'. La figure IV.5 expose les traces de performances des images 'baboon.ppm' et 'fruit.ppm' (voir annexe).

Ces traces d'exécution disposent de 777 valeurs décrivant la performance d'exécution du codeur exprimée en nombre de cycles de codage par segment d'image. Dans les figures IV.4 et IV.5, nous exposons les traces de performance du codeur entropique appliqué sur quatre images couleur différentes qui sont 'Lena.ppm', 'Barbara.ppm', 'baboon.ppm' et 'fruit.ppm'. En analysant le comportement des traces de performance, nous remarquons la présence de trois phases similaires qui s'étendent sur 259 segments. Ces phases représentent le codage des trois composantes d'une image couleur qui sont le rouge, le vert et le bleu. Les traces exposées dans ces figures montrent également l'aspect variable qui dépend des valeurs des coefficients appartenant à des segments d'images différents. La variabilité de la performance de codage d'une seule image est une conséquence directe du chemin de traitement adopté par le codeur. En effet, la structure algorithmique du codeur entropique utilise fréquemment les structures conditionnelles qui déterminent le traitement selon les valeurs numérique des pixels traités. Nous remarquons aussi que les performances de codage varient aussi d'une image à une autre. En conclusion, nous remarquons que les performances de codage entropique sont très dépendantes des valeurs de pixels des images traités.



(a) (b)

Fig. IV. 4 : Variation de la performance de codage par segment d'image : (a) codage appliqué sur l'image Lena.ppm, (b) codage appliqué sur l'image Barbara.ppm



(a) (b)

Fig. IV. 5 : Variation de la performance de codage par segment d'image : (a) codage appliqué sur l'image baboon.ppm, (b) codage appliqué sur l'image fruit.ppm

Afin d'évaluer cette variabilité, nous présentons dans le tableau IV.1 des mesures statistiques décrivant l'ordre de variabilité de la trace. Nous déterminons la valeur maximale (Maximum_performance), la valeur minimale (Minimum_performance), la moyenne et la

variance des différentes traces. La variance dans ce cas est calculée à partir de l'équation IV.1 :

$$\text{var} = \frac{1}{l}\sum_{i=1}^{l}\left(y(i) - \bar{y}\right)^2 \qquad \text{(IV.1)}$$

Nous distinguons par l est la taille de la série de performances y et $\bar{y}$ est la valeur moyenne de la série.

Table. IV. 1 : Mesures statistiques de la trace de performance du codeur entropique

	MAXIMUM_PERFORMANCE (CYCLES)	MINIMUM_PERFORMANCE (CYCLES)	MOYENNE	VARIANCE
Trace du codage de Lena.ppm	51623	9559	33534	3.8090e+007
Trace du codage de Barbara.ppm	56856	11621	36865	7.3785e+007
Trace du codage de baboon.ppm	53166	11328	43065	4.7813e+007
Trace du codage de fruit.ppm	57480	12849	37922	7.3736e+007

La performance du codage entropique, exprimée en nombre de cycles par segment d'image, varie dans de larges intervalles de performance. A titre d'exemple, les performances de codage de l'image 'barbara.ppm' s'étendent sur un intervalle allant de 11621 jusqu'à 56856 cycles. Les valeurs de la variance des différentes traces de performances illustrent bien l'aspect variable de ces traces considérées comme séries temporelles.

Afin de décrire la localité des coefficients des séries temporelles obtenues, nous exposons dans la figure IV.6 les histogrammes des séries temporelles de performances d'exécution du codeur entropique appliqué sur les différentes images considérées.

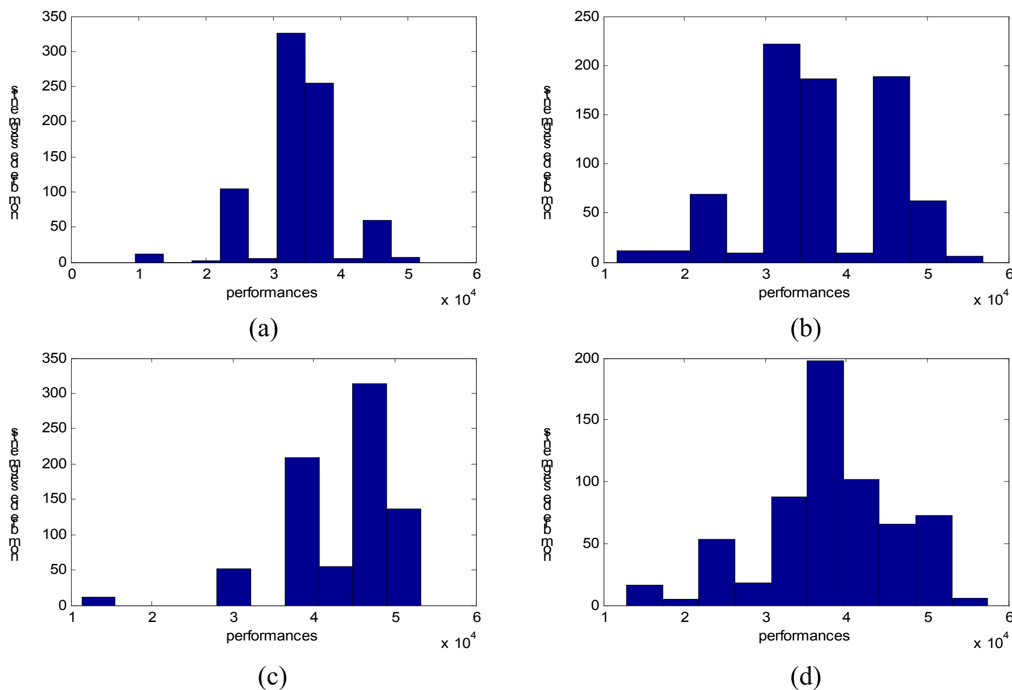


Fig. IV. 6 : Histogrammes des traces de performance du codeur entropique (a) application sur l'image 'Lena.ppm', (b) application sur l'image 'barbara.ppm', (c) application sur l'image 'baboon.ppm', (d) application sur l'image fruit.ppm

En analysant les histogrammes des différentes traces, nous remarquons que la majorité des coefficients se concentrent dans un niveau de performance précis. Nous notons que, respectivement pour les images 'Lena.ppm', 'Barbara.ppm' et 'fruit.ppm', 75%, 53% et 46% des performances sont comprises dans l'intervalle [30000,40000] cycles. Tandis que pour l'image baboon, nous remarquons que 48% des performances sont situées dans l'intervalle [40000, 50000] cycles. L'histogramme de la figure IV.6 montre que la majorité des performances de codage appartiennent à un intervalle de performance précisé. Le pourcentage des performances appartenant à cet intervalle de bornes variables dépend des valeurs des pixels traités de l'image. Les autres performances n'appartenant pas à cet intervalle sont représentées par des pics dans les graphes qui représentent le comportement de ces performances.

IV.3.1.2. Variabilité au niveau des modules

L'application sélectionnée pour illustrer la variabilité au niveau des modules est un réseau de neurones de type SOM (Self Organizing Map). Cette application est composée de plusieurs tâches élémentaires distribuées sur un ensemble de neurones interconnectés. Cette application est particulièrement intéressante dans ce contexte d'étude de variabilité puisque elle est constituée de plusieurs modules de traitement distribué. Dans cette section, nous décrivons l'environnement d'implémentation et de détection des performances des modules constituant l'application SOM. Nous présentons également les traces qui décrivent la variabilité de performance de quelques modules de traitement.

a) Architecture et algorithme d'apprentissage

Le réseau du SOM [27] représenté par un modèle de carte topologique auto adaptative qui permets de coder les vecteurs présentés en entrée, tout en conservant la topologie de l'espace d'entrée. Dans la plupart des applications, les neurones du SOM sont disposés sur une grille 2D (Fig.IV.7).

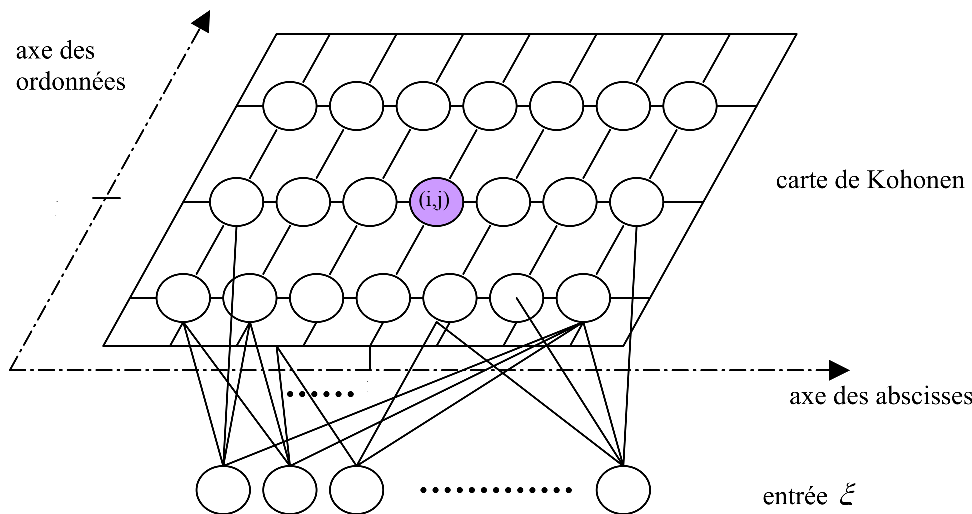


Fig. IV. 7 : Carte topologique auto adaptative de Kohonen

Chaque neurone (i,j) de la carte effectue un calcul de la distance euclidienne entre le vecteur d'entrée ξ et le vecteur poids w_j qui caractérise chaque neurone j de sortie. L'apprentissage du SOM est effectué à travers une mise à jour des poids. La modification des poids est accomplie sur tout un voisinage d'un neurone (i,j) appelé neurone gagnant. Un rayon de voisinage r représente la longueur de la zone du voisinage. Le neurone gagnant (i,j) est

déterminé à condition que la distance euclidienne entre le vecteur d'entrée et le vecteur poids est minimale. L'algorithme d'apprentissage est défini comme suit :

1) Initialiser aléatoirement les vecteurs poids w_{ij}. On affecte une valeur initiale au rayon r et au taux d'apprentissage η.
2) Calculer la distance euclidienne entre le vecteur d'entrée présenté ξ et le vecteur de poids de chaque neurone.
3) Sélection du neurone gagnant : le neurone (i,j) ayant la plus petite distance.
4) Les vecteurs poids de tous les neurones (i,j) de la carte de Kohonen sont alors mis à jour selon :

$$w_{kp}^{it+1} = w_{kp}^{it} + \eta \cdot h((k,p),(i,j)) \cdot (\xi - w_{kp}^{old}) \qquad \text{(IV.1)}$$

h((k,p),(i,j)) est une fonction qui retourne 1 si le neurone (k,p) est inclus dans le voisinage du neurone (i,j) et 0 si non.

5) Réduire la taille du voisinage et rependre l'apprentissage.

Après la phase d'apprentissage, le réseau converge vers la topologie souhaitée. La carte évolue de manière à représenter au mieux la topologie de la base d'entrée.

b) Implémentation et environnement de test

Dans cette section, nous présentons l'environnement d'implémentation et du test du SOM. L'implémentation et la mesure de la performance des différents modules implémentant les neurones du SOM est effectuée dans l'environnement PEK-1.0. Les traitements associés à un neurone sont effectués d'une manière indépendante sur un processeur de type PowerPC. Chaque neurone de la couche de sortie est représenté par un module indépendant qui admet ses propres entrées, une sortie et ses traitements spécifiques. Nous avons introduit, sur le bus de communication, un moniteur de performance qui détecte le nombre de cycles d'exécution associé à chaque neurone. Le schéma de la figure IV. 8 représente l'architecture développée.

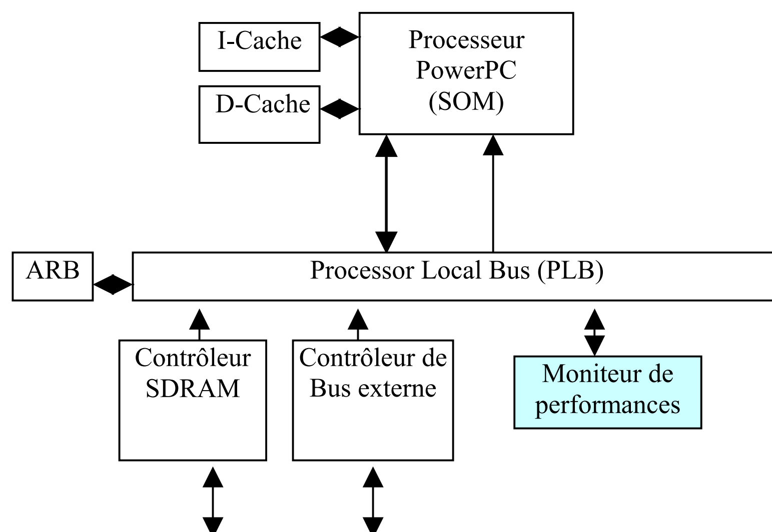


Fig. IV. 8 : Environnement d'exécution et de détection de performances des neurones d'un SOM

L'apprentissage du réseau SOM est effectué d'une manière séquentielle selon les neurones sur un processeur PowerPc. Nous supposons qu'on dispose d'une carte de Kohonen comportant neuf neurones de sortie. Une itération d'apprentissage relative à un vecteur d'entrée comporte dix étapes dont neuf consistent à calculer les distances qui séparent ce

vecteur aux vecteurs poids des neurones de sortie. La dixième étape consiste à déterminer la distance minimale et de corriger les poids du neurone gagnant et de son voisinage. Dans ce cas, chaque neurone représente une unité de traitement indépendante effectuant le calcul de la distance euclidienne entre le vecteur poids et le vecteur d'entrée. Elle adapte également les poids de connexion si l'algorithme d'apprentissage sélectionne ce neurone pour une éventuelle correction.

A la fin de chaque étape, le programme exécuté sur le processeur signale la fin de la tâche. Le moniteur de performance détecte le début et la fin de chaque étape. Il mesure également le nombre de cycles nécessaire pour finaliser chaque étape.

c) Analyse de variabilité

En analysant les traitements effectués pendant la phase d'apprentissage du SOM, nous remarquons que la variabilité au niveau des performances est provoquée par l'algorithme d'apprentissage. En effet, l'étape de correction des poids est effectuée uniquement pour les neurones situés dans le voisinage du neurone gagnant. Cette étape de correction représente, dans ce cas, une charge de travail supplémentaire pour le voisinage sélectionné.

L'exemple retenu dans cette section traite un problème de reconnaissance d'images en utilisant un SOM comportant 256 neurones (modules) de sorties. La base d'apprentissage est constituée de 4096 segments obtenus à partir de l'image 'lena.ppm' de taille 1024x1024 pixels. Dans la figure IV. 9, nous présentons un extrait des traces exprimant la variabilité de performance (nombre de cycles par étape) de quelques neurones. Dans ce cas, la taille de la couche d'entrée est égale à la taille du segment fixée à 256.

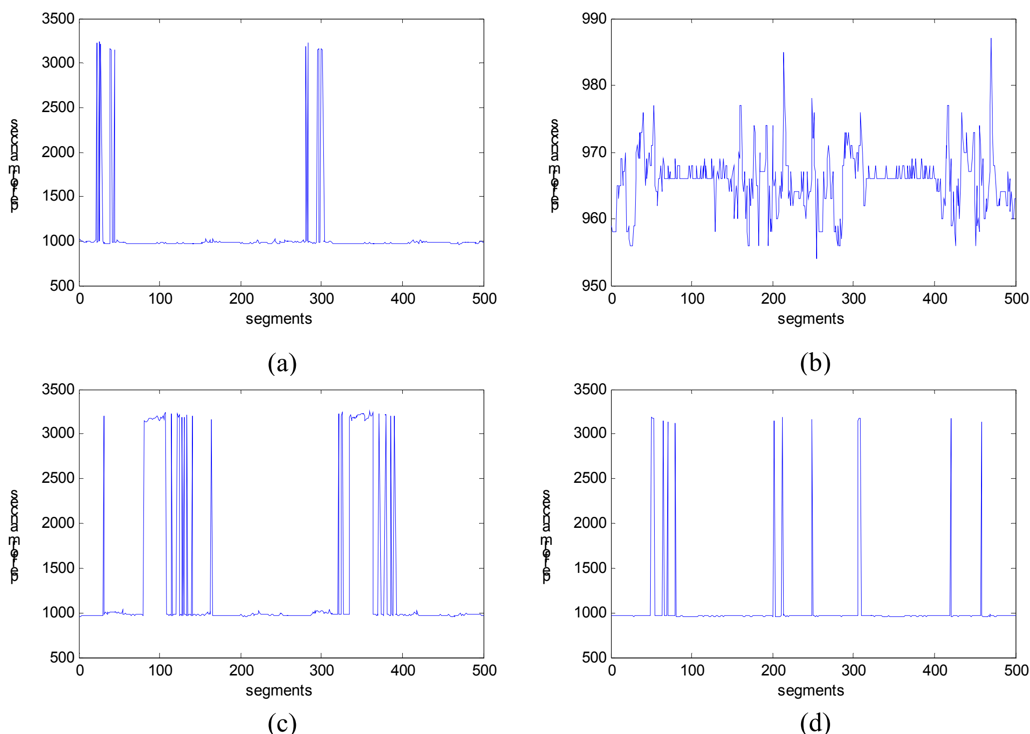


Fig. IV. 9 : Variation de la performance d'apprentissage par segment d'image et par neurone (a) performances du neurone (2, 14), (b) performances du neurone (3, 2), (c) performances du neurone (5, 16), (d) performances du neurone (12, 8)

Afin d'évaluer cette variabilité, nous présentons dans le tableau IV.2 les mesures statistiques décrivant l'ordre de variabilité de la trace. Nous déterminons la valeur maximale

(Maximum_performance), la valeur minimale (Minimum_performance) de performance, la moyenne et la variance de chaque trace.

Table. IV. 2 : Mesures statistiques de la trace de performance de quatre neurones

	TAILLE	MAXIMUM_PERFORMANCE (CYCLES)	MINIMUM_PERFORMANCE (CYCLES)	MOYENNE	VARIANCE
Neurone (2,14)	4096	3323	957	1.1218e+003	2.9529e+005
Neurone (3,2)	4096	3317	954	970.52	8.5307e+003
Neurone (5,16)	4096	3291	959	1.2215e+003	4.8135e+005
Neurone (12,8)	4096	3235	955	1.0313e+003	1.3472e+005

Nous détectons une grande variabilité au niveau des traces de performances des modules représentés par les neurones de la couche de sortie. En effet, les performances détectées varient brusquement entre deux zones distantes. Ces deux zones de valeurs sont situées au voisinage des valeurs maximales et minimales. Nous détectons respectivement pour les neurones (2,14), (3,2), (5,16), (12,8) que 93.25%, 99.83%, 89.09 et 97.14% des valeurs de performances résident dans le voisinage de la borne minimum de la trace.

IV.3.1.3. Variabilité au niveau instruction

La variabilité au niveau des instructions est dirigée par plusieurs métriques qui décrivent la performance d'exécution de l'application (voir section III.4). Parmi les métriques les plus déterminant dans le processus d'évaluation de la performance d'exécution, nous distinguons les valeurs des adresses d'instructions. En effet, les travaux présentés dans la section III.4 montrent e lien d'une telle métrique avec la performance de l'application.

Dans cette section, nous allons présenter en premier lieu l'outil d'extraction des traces d'exécution des programmes. Ensuite nous allons exposer les caractéristiques statistiques des traces des adresses mémoires.

a) Extraction des traces d'instructions

Dans ce paragraphe, nous nous intéressons à deux types de traces d'adresses d'instructions. Une première trace a été extraite suite à l'exécution de différentes applications sur un processeur généraliste de type Pentium mobile 4 de la famille Intel. La performance du processeur est estimée à 2.4GHZ, tandis que sa mémoire cache de second niveau est fixé à 512KO. L'extraction de la trace des adresses mémoires allouées lors de l'exécution de l'application est effectuée grâce à l'outil Pin-2.0 [3]. Pin-2.0 est un outil académique, il permet l'instrumentation de programmes s'exécutant dans un environnement Linux sur plusieurs types de processeurs tels que Intel, Xscale, IA-32 et Itanium. Le principe de PIN est d'ajouter dynamiquement des instructions d'instrumentation dans l'application que l'on désire tester. Cet outil est un compilateur qui traite les instructions de l'application en ligne 'JIT compiler' (Just In Time compiler). Dans ce cas, l'instrumentation des instructions est effectuée juste avant d'être exécutée. Pin-2.0 admet une application exécutable sous Linux en entrée. En compilant cette application, Pin-2.0 génère essentiellement la trace des adresses instructions et données de l'application. Les entrées/sorties de l'outil sont présentées par la figure IV.10. Pin-2.0 permet de générer les traces des adresses instructions et données d'un programme tout en spécifiant la nature de l'accès (lecture/écriture). Pin-2.0 génère aussi le compte des instructions déployés dans ce programme.

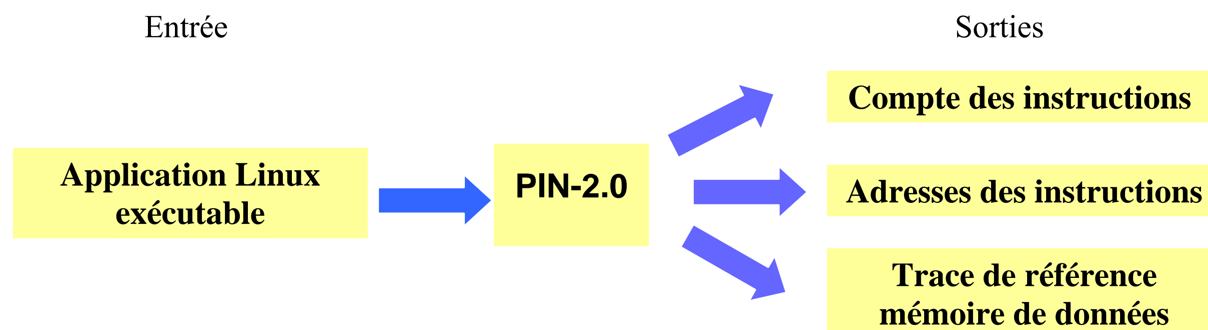


Fig. IV. 10 : Les entités d'entrées/sorties de l'outil PIN-2.0

Le deuxième type de trace est capté à la suite de l'exécution d'une application sur un simulateur de processeurs de type PowerPC d'IBM [4]. Le simulateur de processeurs PowerPC est proposé dans la bibliothèque PEK-1.0 [5] d'IBM. La bibliothèque PEK-1.0, qui sera détaillée dans le chapitre VI, introduit des interfaces SystemC regroupant des composants IBM et permettant la modélisation de systèmes sur puces au niveau transactionnel. Le modèle du processeur utilisé est le PowerPC 405 qui se base sur le simulateur d'instruction ISS[39] et sur un débogueur appelé RiscWatch (RW). Le RW permet le déboguage des instructions exécutées sur le ISS. Pour générer le code exécuté sur l'architecture PowerPC, nous utilisons le compilateur GNU PowerPC Cross-Compiler [6]. L'architecture du PowerPC proposée dans PEK-1.0 présente des aspects paramétrables tels que la taille du cache et la fréquence de l'horloge.

b) Caractéristiques des traces d'instructions

Les traces des adresses d'instructions générées suite à l'exécution d'une application sur un processeur généraliste ou un processeur embarqué de type PowerPC sont caractérisées par :

1. Des traces contenant des millions de données. Dans cette étude nous nous limitons à étudier une partie (100000 adresses d'instructions) des traces générées par l'outil.
2. Des adresses mémoires d'instructions qui appartiennent à plusieurs localités. Chaque localité contient un nombre variable de données.
3. Une grande variance au niveau des séries d'adresses. Ceci est expliqué par la dimension importante de l'espace d'adressage dans une mémoire.

Nous représentons dans la figure IV.11 un extrait (2000 valeurs entières des adresses) de la trace d'exécution de trois applications différentes exécutées sur le processeur généraliste. Les applications testées sont LS[40], CP[41] et GZIP[42] qui représentent des applications directement exécutables sur un terminal Linux.

[39] ISS : Instruction-Set Simulator
[40] LS : binaire linux
[41] CP : binaire linux
[42] GZIP : binaire linux

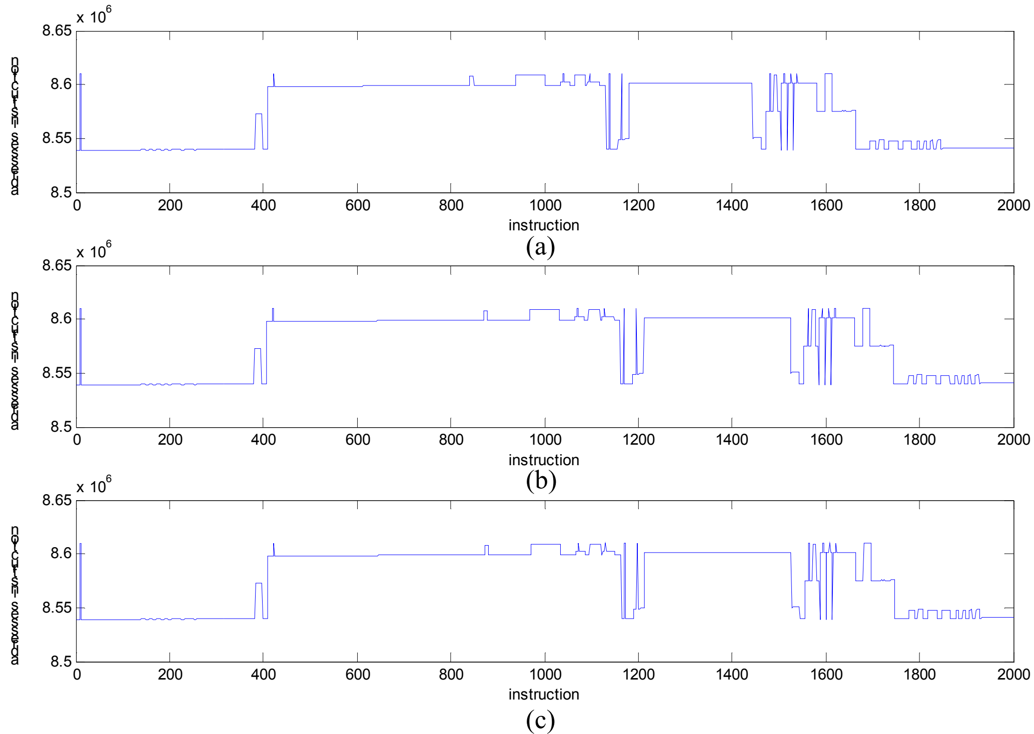


Fig. IV. 11 : Traces d'adresses des instructions (a) trace d'exécution de l'application LS, (b) trace d'exécution de l'application CP, (c) trace d'exécution de l'application GZIP

Les traces complètes d'adresses des instructions exposées dans la figure IV.11 se composent réellement de quelques millions d'éléments. Nous représentons dans cette figure une partie de cette série pour illustrer le comportement variable dans le processus d'allocation mémoire des instructions. Dans le tableau IV.3, nous exposons quelques mesures statistiques décrivant les différentes traces. Les caractéristiques exposées dans le tableau IV.3 sont l'adresse maximale, l'adresse minimale, la moyenne et la variance des différentes traces.

Table. IV. 3 : Caractéristiques des traces d'instructions d'exécution sur un processeur généraliste

	TAILLE	ADRESSE MAXIMALE	ADRESSE MINIMALE	MOYENNE	VARIANCE
LS	100K	134.565638 e+006	2.140644 e+006	8.8558 e+006	1.7011 e+013
CP	100K	134.547942 e+006	8.539104 e+006	1.2813 e+007	4.7488 e+014
GZIP	100k	134.555070 e+006	8.539104 e+006	1.0955 e+008	2.5202 e+015

La mesure de la variance présentée dans le tableau IV.3 illustre le comportement hautement variable des adresses mémoires allouées lors de l'exécution d'une application. A titre d'exemples, les valeurs entières des adresses allouées lors de l'exécution de l'application LS varient dans un intervalle allant de 2.140644 e+006 jusqu'à 134.565638 e+006.

Afin de décrire la localité des coefficients des séries temporelles obtenues, nous exposons dans la figure IV. 12 les histogrammes des séries temporelles d'adresses des instructions des différentes applications testées.

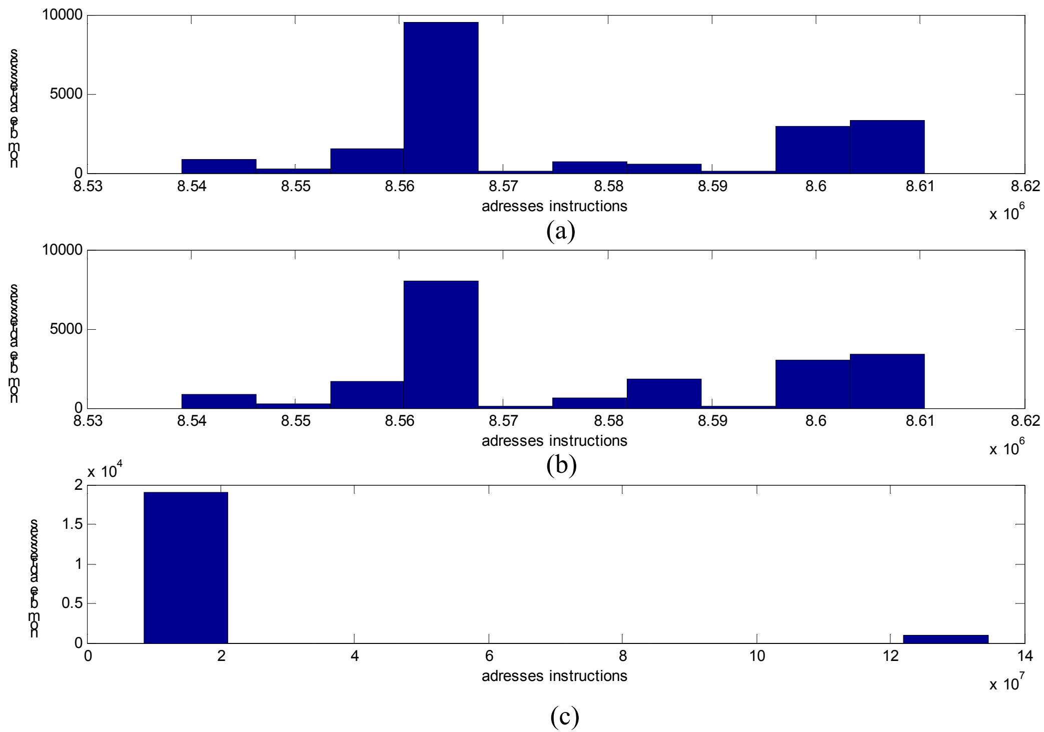


Fig. IV. 12 : Histogrammes des traces d'adresses d'instructions : (a) application sur le programme LS, (b) application sur le programme CP, (c) application sur le programme GZIP

Les histogrammes de la figure IV.12 démontrent la variabilité de la trace d'adresses des instructions entre plusieurs localités de l'espace d'adressage. Nous remarquons, pour la trace d'exécution de l'application LS et CP, que respectivement 56% et 49% des données appartiennent à l'intervalle [8.55 e+006, 8.57 e+006]. Pour le cas de l'application GZIP, nous remarquons la présence d'une variabilité plus importante. Cette variabilité est le résultat de l'apparition d'une localité assez distante de la zone d'adressage habituelle qui regroupe 95.43% des adresses. Les séries temporelles de coefficients exposées jusqu'à présent sont composées par les valeurs entières des adresses d'instructions allouées suite à l'exécution d'une application sur un processeur généraliste de type Pentium 4 d'Intel. Dans ce qui suit, nous allons présenter les caractéristiques de ces séries temporelles lors de l'exécution d'une application sur un processeur embarqué de type PowerPC.

Dans la figure IV.13, nous présentons un extrait (2000 adresses) de la trace d'exécution [34] de deux applications différentes exécutées sur un processeur embarqué de type PowerPC. Les applications testées sont le FFT[43] et le CRC[44] [7].

[43] FFT : Fast Fourier Transform
[44] CRC : Codage à Redondance Cyclique

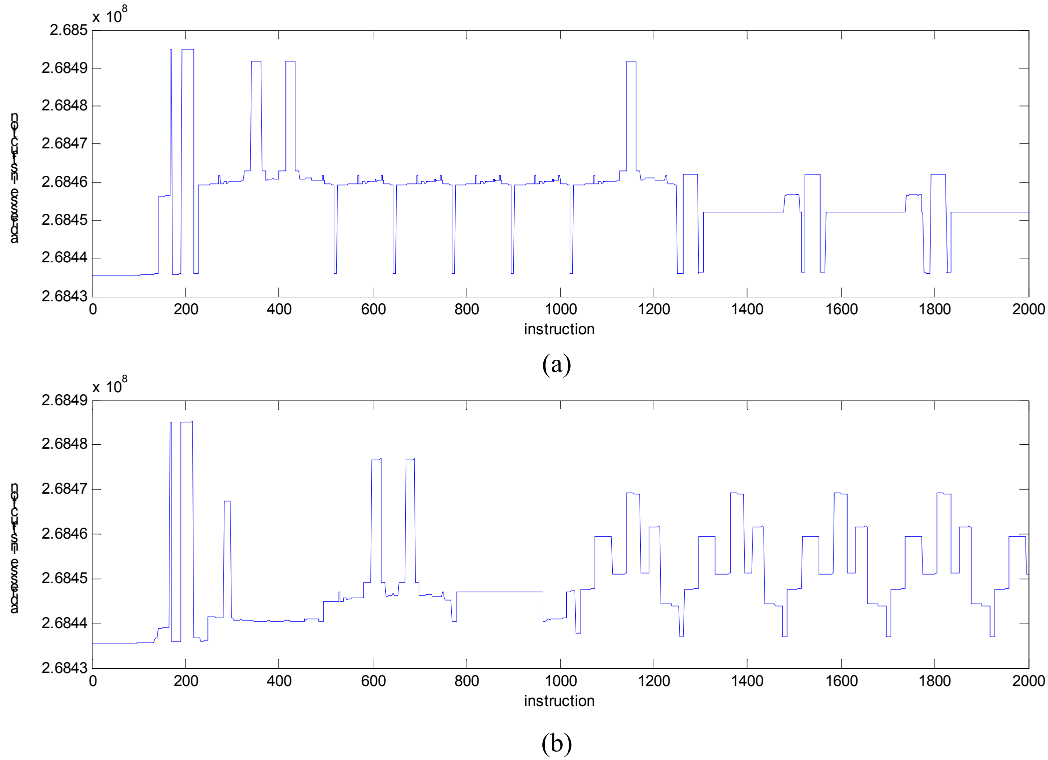


Fig. IV. 13 : Traces d’adresses des instructions (a) trace d’exécution de l’application FFT, (b) trace d’exécution de l’application CRC

Dans le tableau IV.4, nous exposons quelques mesures statistiques décrivant les différentes traces. Les caractéristiques exposées dans le tableau IV.4 sont l’adresse maximale, l’adresse minimale, la moyenne et la variance des différentes traces.

Table. IV. 4 : Caractéristiques des traces d’instructions d’exécution sur un processeur embarqué de type PowerPC

	TAILLE	ADRESSE MAXIMALE	ADRESSE MINIMALE	MOYENNE	VARIANCE
FFT	100k	268495020	268435460	2.6845e+008	1.8299e+007
CRC	100k	268485244	268435460	2.6845e+008	8.0372e+007

IV.3.2. Flot de conception dynamique de systèmes sur puce reconfigurables

Le schéma de la figure II.1 (voir chapitre II) représente le flot de conception d’un système sur puce à composition statique. Nous proposons dans cette thèse un nouveau flot de conception (Fig.IV.14) de systèmes sur puce effectuant l’exploration et l’adaptation en ligne de l’architecture du SOPC au niveau système. Cette méthodologie suppose que le concepteur a déjà analysé le comportement des applications à implémenter et il a ensuite détecté les modules qui génèrent un comportement hautement variable. Le flot de conception proposé dans la figure IV.14 permet d’explorer, à un niveau système, les différentes reconfigurations possibles de l’architecture. Cette étape d’exploration permet de déterminer, à un niveau système, les différentes phases d’exécution de l’application ce qui implique une maîtrise totale de l’évolution de l’architecture reconfigurable. L’application de ce flot de conception permet le test et la modélisation de systèmes sur puces réactifs et autonomes.

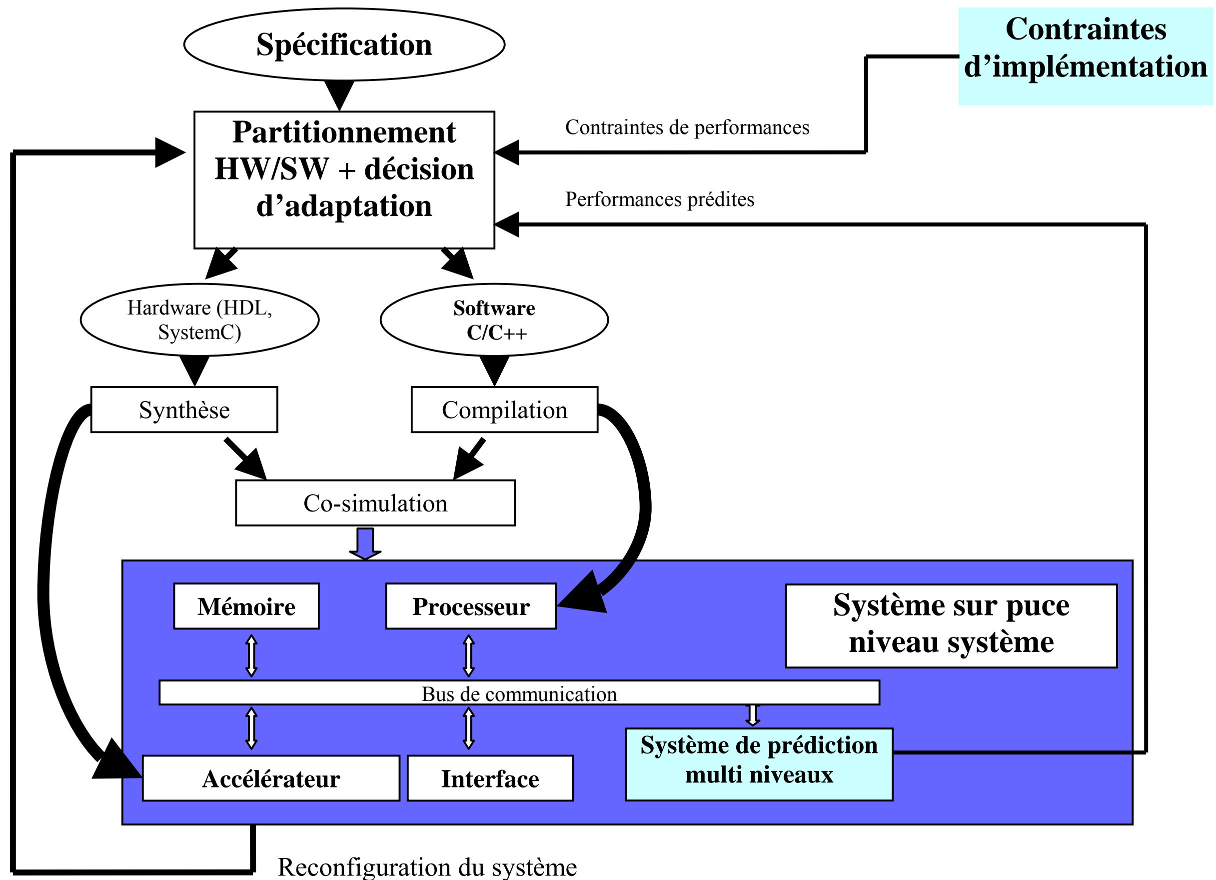


Fig. IV. 14 : Flot de conception d'un système sur puce reconfigurable [35]

Pour chaque application présentant une variation considérable de sa performance appelée composant potentiellement reconfigurable, nous associons un système de prédiction multi niveaux capable d'estimer les performances futures des différents composants logiciels ou matériels de l'application. Dans ce cas, le système de prédiction est inséré manuellement dans la plateforme. Les estimations générées par le système de prédiction multi niveaux associées aux contraintes temps réels imposées par l'application représentent les entrées d'un système de décision qui fixe l'architecture de la plateforme. Les entrées d'un système reconfigurable seront alors représentées par les contraintes d'implémentation et de performances en plus des données à traiter. Cette méthodologie détermine et implémente, en ligne et d'une manière autonome, les différentes reconfigurations possibles durant l'exécution de l'application.

La méthodologie de reconfiguration exposée dans cette section se base sur l'intégration d'estimateurs en ligne dans la plateforme. On distingue plusieurs techniques de prédiction qui ont été étudiées dans la littérature. Dans la section suivante, nous exposons les techniques de prédiction statistiques et à base des réseaux de neurones.

IV.4. Techniques de prédiction

Dès qu'un phénomène est géré par un système d'équations déterministes et connues, il est alors possible de résoudre le comportement du système en connaissant l'état initial. En revanche, lorsque ces équations sont inconnues, il est nécessaire d'approximer les lois qui dirigent l'évolution du comportement du système à partir d'un historique de ses observations antérieures. Les séries temporelles apparaissent alors naturellement dans les domaines

scientifiques où on détecte le besoin de caractériser un comportement spécifique d'un système. Le désir d'anticiper l'évolution de phénomènes particuliers a donc motivé l'emploi des méthodes de prédiction.

La prédiction des séries temporelles est une application d'estimation du comportement d'une suite d'observation $y(k), k \in IN$ d'une série temporelle y à différentes périodes t. En pratique, y est une métrique qui caractérise un système particulier [8], [9]. La prédiction du comportement de la série revient à prédire les valeurs futures de y telles que :

$$\hat{y}(k+n) = p(y(k+n-1), \quad y(k+n-2), ..., y(k+n-N)) \qquad \text{(IV.1)}$$

Nous distinguons ici deux paramètres : n qui désigne l'horizon de prédiction du prédicteur p, N désigne le nombre de valeurs antérieures de la série y prises en compte pour prédire la valeur $y(k+n)$. En effet, pour prédire $\hat{y}(k+n)$ qui représente la valeur prédite de $y(k+n)$, on se base sur la relation appelée corrélation qui existe entre les différents coefficients de la série. On peut définir le niveau de corrélation entre les composants de la série comme l'entité qui représente le degré de dépendance temporelle entre les différents composants de la série. De ce fait, connaissant le présent et le passé, on peut prédire la valeur d'une série à une date future.

Plusieurs techniques ont été proposées dans la littérature pour approcher une série temporelle. On peut distinguer deux ensembles de méthodes : l'ensemble des méthodes statistiques ARMA[45], ARIMA[46] et le prédicteur de Markov. Le deuxième ensemble regroupe les méthodes de prédiction à base de réseaux de neurones ou des systèmes flous.

IV.4.1. Techniques de prédiction statistiques

Les modèles linéaires ont la double particularité d'être solidement ancrés à la théorie des processus aléatoires, et de fournir un cadre d'étude simple. En contre partie, la linéarité est souvent inappropriée à la modélisation de systèmes gouvernés par des équations analytiques. Les modèles prédictifs d'un point de vue statistique se fondaient principalement sur une vision purement mathématique. De ce fait, un processus aléatoire était modélisé par :

$$y_t = p(t) + \varepsilon_t \qquad \text{(IV.2)}$$

Où ε_t est une variable aléatoire et p est la fonction qui estime les valeurs de la série temporelle y.

L'examen attentif du comportement irrégulier qu'exhibe un système aléatoire conduit à rompre avec l'approche mathématique traditionnelle. Clairement, le modèle AutoRegressive $AR(k)$ d'ordre k (IV.3) s'adapte plus au comportement aléatoire de la série temporelle.

$$y_t + a_1 y_{t-1} + ... + a_k y_{t-k} = \varepsilon_t \qquad \text{(IV.3)}$$

[45] ARMA : Auto-Regressive Moving Average
[46] ARIMA : Auto-Regressive Integrated Moving Average

ε_t désigne un bruit blanc, c'est à dire un processus aléatoire dont la densité spectrale de puissance est constante sur tout l'axe des fréquences. Une classe complémentaire de modèles linéaires $MA(l)$ est représentée à l'aide des moyennes mobiles d'ordre l définies par :

$$y_t = \varepsilon_t + b_1\varepsilon_{t-1} + ... + b_l\varepsilon_{t-l} \qquad \text{(IV.4)}$$

L'étape suivante consiste naturellement à mêler les deux modèles en un. On obtient ainsi un modèle $ARMA(k,l)$ du type :

$$y_t + a_1 y_{t-1} + ... + a_k y_{t-k} = \varepsilon_t + b_1\varepsilon_{t-1} + ... + b_l\varepsilon_{t-l} \qquad \text{(IV.5)}$$

Retenons que l'évaluation des paramètres a_i et b_i des modèles $AR(k)$, $MA(l)$ et $ARMA(k,l)$ reposent sur la fonction d'auto-corrélation de la série temporelle y_t. L'ordre du modèle est généralement déterminé d'une façon heuristique. Ainsi, la classe des modèles linéaires se prête à la modélisation des suites temporelles lorsque la fonction d'auto-corrélation est représentative des caractéristiques statistiques inhérentes au comportement de la série temporelle. Cette propriété d'auto-corrélation s'avère inexistante pour des non linéarités parfois excessivement simples. Pour résoudre ce problème de non linéarité, le modèle auto-régressif à seuil est proposé. L'idée consiste à compartimenter l'espace des phases et d'appliquer dans chaque région un modèle ARMA différent. Toutefois, cette approximation linéaire par morceaux nécessite un grand nombre de modèles pour modéliser une série temporelle complexe.

IV.4.2. Techniques de prédiction à base de réseaux de neurones

Les réseaux connexionnistes appelés aussi les réseaux de neurones artificiels occupent une place centrale parmi les outils de prédiction. Ils représentent également le thème d'un courant de recherche innovant qui s'intéresse à l'analyse et la prédiction des séries temporelles et des systèmes dynamiques [10-12]. Cette inspiration tient à la propriété des réseaux non bouclés à trois couches capables d'approximer une fonction non linéaire arbitraire [13] et à la convergence de l'algorithme d'apprentissage basé sur la descente du gradient [14]. De ce fait, un réseau de neurones est considéré comme un approximateur universel, il est alors capable d'émuler des caractéristiques non anticipées des séries temporelles décrivant un système dynamique sans rien connaître des équations qui le gouvernent.

IV.4.2.1. Réseaux de neurones et algorithme d'apprentissage

Les réseaux de neurones (RN) sont des systèmes de traitement distribué présentant de bonnes performances de calcul à travers une interconnexion dense d'éléments de calcul simple (neurone). L'activité parallèle et en temps réel de nombreux composants, la représentation distribuée des connaissances et l'apprentissage par modification des connexions sont les avantages marquant de l'approche connexionniste.

Les réseaux de neurones représentent un modèle de calcul actif pour une grande variété de problèmes tels que la classification, la reconnaissance et la synthèse de la parole, les

interfaces adaptatives homme/machine, l'approximation de fonction, la compression d'images, la prédiction des séries temporelles, la modélisation des systèmes non linéaires et le contrôle. Plusieurs paramètres déterminent le fonctionnement d'un réseau de neurones. Il s'agit essentiellement de l'architecture du réseau, les fonctions de transfert figées dans les neurones et l'algorithme d'apprentissage du réseau :

1. L'architecture d'un réseau de neurone est un facteur déterminant pour garantir la robustesse d'un réseau de neurones. Elle définit l'organisation des neurones (modèle en couches, modèle 2D), le modèle d'interconnexion entre les différents neurones (connexion directe ou récurrente, connexion totale ou partielle). L'architecture d'un réseau de neurones détermine également la cardinalité des neurones à l'intérieur de chaque couche.

2. Les fonctions de transfert représentent l'effet de chaque neurone sur un flux de données d'entrées. Par analogie avec la neurone biologique, le comportement du neurone artificiel se compose de deux phases (Fig. IV.15).

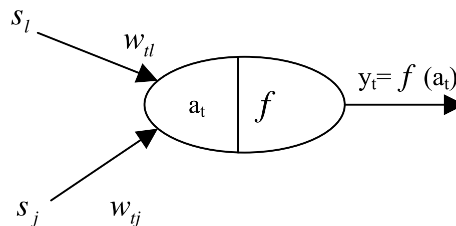


Fig. IV. 15 : Neurone formel

La première phase, appelée activation, représente le calcul de la somme pondérée des entrées du neurone selon l'expression suivante :

$$a_t = \sum_j w_{tj} s_j \qquad \text{(IV.7)}$$

Dans la deuxième phase, une fonction de transfert évalue la valeur de l'état du neurone. C'est cette valeur qui sera transmise aux neurones en aval. Il existe de nombreuses formes possibles pour la fonction de transfert dont les plus courantes sont présentées sur la figure IV.16. La plupart des fonctions de transfert sont continues, offrant une infinité de valeurs possibles dans l'intervalle [0,1] ou [-1,1].

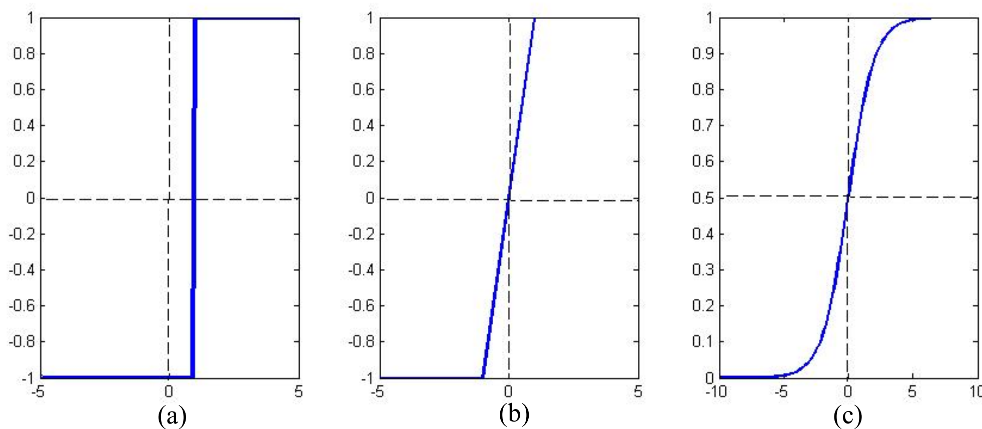


Fig. IV. 16 : Différents types de fonctions de transfert : a) fonction à seuil, b) fonction linéaire, c) fonction sigmoïde

3. L'une des caractéristiques les plus complexes du fonctionnement d'un réseau de neurones est la phase d'apprentissage. c'est une phase au bout de laquelle on agit sur les poids de connexion entre les neurones : quelques poids de connexion sont renforcés, les autres sont affaiblis. Le réseau converge alors vers le comportement souhaité. Ceci nous amène à la notion de mémoire figée dans les réseaux de neurones et à la notion de capacité de généralisation. Plusieurs algorithmes d'apprentissage ont été développés depuis la première règle de Hebb [20].

Dans la littérature, on trouve plusieurs architectures de réseaux de neurones statiques et dynamiques. Les réseaux de neurones statiques tels que les perceptrons multi couches et le SOM (Self Organizing Map) n'implémentent pas la notion temps. En revanche, on distingue deux grandes familles de réseaux de neurones avec la prise en compte d'une certaine représentation du temps : les réseaux à représentation externe du temps, et ceux à représentation interne. Les réseaux à représentation externe du temps admettent des informations sur les délais sur la couche d'entrée. En revanche, les réseaux de neurones à représentation implicite du temps utilisent des connexions bouclés afin de représenter cette notion de temps.

Nous nous sommes principalement intéressés aux réseaux de neurones récurrents (RNR) qui implémentent une représentation interne du temps. La figure IV.17 représente une classification de plusieurs réseaux de neurones statistiques et temporels.

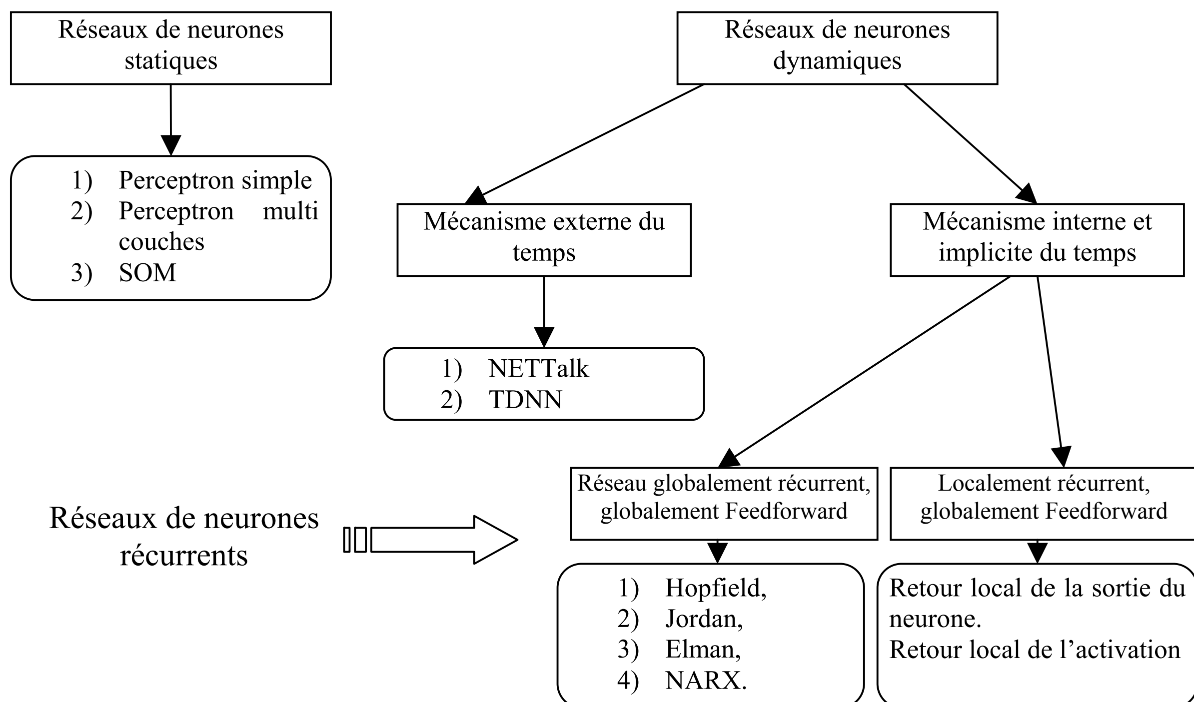


Fig. IV. 17 : Classification d'architectures de réseaux de neurones

Dans la partie des réseaux de neurones à mécanisme externes du temps, on dispose de deux exemples d'architectures qui sont NETTalk et le TDNN[47]. Le NETTalk est un réseau de neurones qui a été implémenté pour la synthèse de la parole. Le TDNN est un réseau de neurones appliqué à la reconnaissance de la parole. L'aspect temporel du TDNN est implémenté en utilisant des retards temporels pour chaque neurone du réseau.

[47] TDNN : Time Delay Neural Network

La deuxième partie comporte les réseaux à mécanisme interne et implicite du temps. Cette partie représente l'ensemble des réseaux de neurones récurrents. Dans le cas d'un modèle neuronal récurrent, la connectivité des unités dans les réseaux ne se limite pas à des connexions en avant (feedforward). Tout type de connexion est admis, c'est à dire d'un neurone à n'importe quel autre, y compris lui même. Cette architecture donne lieu à un comportement dynamique complexe. On distingue deux types de réseaux récurrents : les réseaux à récurrence globale et localement récurrent. La récurrence globale est représentée par la réinjection de la sortie du réseau ou de sa couche cachée dans la couche d'entrée. En revanche, la récurrence locale est représentée par un retour de l'activation d'un neurone à lui même. Dans les travaux de prédiction de séries temporelles effectués dans le cadre de cette thèse, on s'intéresse particulièrement à l'application des réseaux de neurones récurrents (RNR) pour résoudre le problème de prédiction des performances variables des composants qui constituent un système sur puce.

IV.4.2.2. Application des réseaux de neurones à la prédiction des séries temporelles

On peut trouver plusieurs architectures de réseau de neurones appliquées à la prédiction présentées dans la littérature notamment avec les réseaux de neurones dynamiques [11]. L'architecture récurrente d'un réseau de neurones dynamique utilise des connexions récurrentes locales au niveau de la couche cachée. Le réseau, et contrairement au réseau NARX, n'utilise pas des entrées $y(t)$ de la série. L'équation (IV.13) représente le modèle du réseau récurrent dynamique.

$$y(t) = f(u(t-d_u),...,u(t-1),u(t)) \qquad \text{(IV.13)}$$

$u(t)$ représente les entrées du RNR. On remarque que cette équation n'admet pas de sorties du réseau rediriger vers la couche d'entrée.

D'autres architectures de réseaux de neurones ont été appliquées sur des problèmes de prédiction comme les réseaux de Hopfield [21]. D'autres travaux ont été menés sur la flexibilité de l'architecture neuronale en termes de connexion et de structure durant la phase d'entraînement du réseau [22], [23] afin d'améliorer ses capacités d'apprentissage. Finalement, on distingue la présence de travaux qui portent sur la description d'architectures hybrides qui combinent dans un même système des techniques neuronales et des techniques statistiques [24] ou des techniques neuronales différents. Nous nous intéressons principalement à l'architecture récurrente appelée NARX[48] caractérisée par un modèle simple de récurrence associé à un algorithme d'apprentissage pratique à implémenter.

Dans cette section, nous allons présenter l'architecture du RNR et l'algorithme d'apprentissage appliqués à la prédiction des séries temporelles. Nous nous focalisons principalement sur le modèle NARX [15-17] représenté par la figure IV.18.

[48] NARX: Nonlinear AutoregRessive with eXogenous input

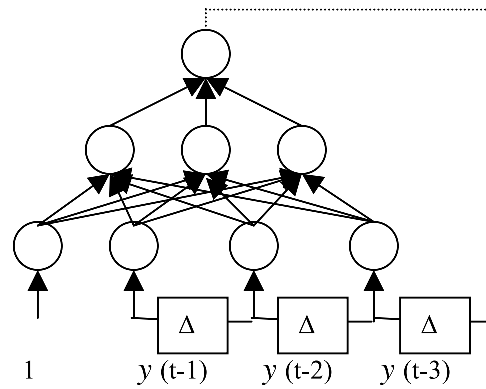


Fig. IV. 18 : Réseau de neurones récurrent de type NARX

En analysant la structure de l'architecture NARX, nous remarquons qu'il s'agit de connecter n (n désigne l'horizon de prédiction) réseaux de neurones multi couches. La figure IV.19 expose la décomposition du NARX en plusieurs réseaux multi couches dans le cas où n=3.

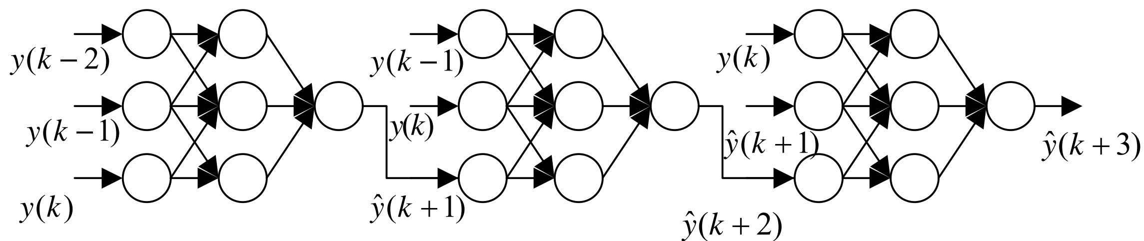


Fig. IV. 19 : Structure décomposée d'un réseau NARX (n=3)

Contrairement aux autres architectures récurrentes, l'architecture d'un modèle NARX implémente une récurrence limitée reliant la sortie du réseau à sa couche d'entrée. Malgré cette limitation, cette architecture représente un modèle de calcul puissant [18]. En pratique, et contrairement aux autres réseaux récurrents, l'apprentissage d'un réseau de type NARX est simple. En effet, l'algorithme de rétro propagation du gradient [14] est suffisant pour effectuer un apprentissage efficace du modèle NARX [19]. Le modèle NARX est présenté par l'équation (IV.8).

$$y(t) = \Psi(u(t-d_u),...,u(t-1),u(t),\hat{y}(t-d_y),...,\hat{y}(t-1)) \qquad \text{(IV.8)}$$

$u(t)$ et $y(t)$ représentent respectivement les entrées et les sorties du modèle à l'instant t. d_u, d_y représentent l'ordre d'entrée et de sortie. Ψ représente une fonction non linéaire à approcher par un modèle NARX.

La phase d'apprentissage du NARX est effectuée à l'aide de l'algorithme de rétro propagation du gradient de l'erreur à travers le temps BPT (Back Propagation through Time). Cet algorithme fournit une façon de modifier les poids de connexions de toutes les couches d'un réseau NARX. L'algorithme de rétro propagation standard est appliqué sur le réseau NARX assimilé à une connexion en cacade de n réseaux de neurones multi couches. Pour le BPT, l'erreur est calculée et incrémentée pour chaque couche et pour chaque itération. La correction des poids n'est effectuée qu'à la n$^{\text{iéme}}$ itération.

L'objectif de l'algorithme d'apprentissage est de minimiser la fonction coût J représentée par l'équation IV.9.

$$J = \frac{1}{2}\sum_{h=1}^{n}[\hat{y}(k+h) - y^{d}(k+h)]^2 = \frac{1}{2}\sum_{h=1}^{n} A_h \qquad \text{(IV.9)}$$

n étant l'horizon de prédiction. Le principe d'apprentissage est d'ajouter, dans le cas où la sortie obtenue $\hat{y}_h(k+h)$ du réseau est différente de celle désirée $y_h(k+h)$, une quantité Δw_{ij} aux poids de chaque connexion reliant le neurone j au neurone i.

$$w_{ij}^{it+1} = w_{ij}^{it} + \Delta w_{ij} \qquad \text{(IV.10)}$$

Δw_{ij} est la correction pour chaque exemple k+h de la base d'apprentissage, la valeur de la correction relative à l'exemple h de la base d'apprentissage est calculée en utilisant l'équation (IV. 11).

$$\Delta w_{ij} = -\eta \frac{\partial J}{\partial w_{ij}} = -\eta \sum_{h=1}^{n} \frac{\partial A_h}{\partial \hat{y}(k+h)} \cdot \frac{\partial \hat{y}(k+i)}{\partial w_{ij}} \qquad \text{(IV. 11)}$$

avec

$$\frac{\partial J}{\partial w_{ij}} = \sum_{h=1}^{n}[\hat{y}(k+h) - y(k+h)] \cdot \left[\sum_{j=1}^{N} \frac{\partial \hat{y}(k+h)}{\partial \hat{y}(k+h-j)} \cdot \frac{\partial \hat{y}(k+h-j)}{\partial w_{ij}} + \frac{\partial \hat{y}\ (k+h)}{\partial w_{ij}} \right] \qquad \text{(IV. 12)}$$

L'algorithme d'apprentissage du RNR a permis de dépasser les limites d'apprentissage des réseaux multi-couches. Il est capable de résoudre un grand nombre de problèmes. Mais il présente de nombreux défauts :

- Apprentissage très lent en termes de temps de calcul.
- Une grande sensibilité aux conditions initiales notamment à la manière dont l'algorithme initialise les poids de connexions.
- Des problèmes liés à la présence des minimums locaux dans la phase d'apprentissage.

IV.4.3. Prétraitement

La réduction de la base d'apprentissage d'un réseau de neurones en gardant la même fonction discriminante à apprendre par le réseau est réalisée à travers plusieurs méthodes. Ces méthodes permettent de détecter et d'éliminer la redondance, les vecteurs non influents dans la phase d'apprentissage ou de coder les valeurs d'apprentissage dans une nouvelle présentation de taille inférieure.

La réduction de données de la base d'apprentissage peut passer par une étape de classification en utilisant plusieurs classifieurs qui partagent la base initiale en plusieurs sous-ensembles séparés de taille modérée. Le partitionnement peut être fait d'une manière aléatoire (calcul simple, résultats satisfaisants), ou par groupement (clustering), qui est plus performant pour la phase d'apprentissage. Le partitionnement peut être le résultat d'application de plusieurs classifieurs comme le SOM ou le SVM[49] [25]. Dans [26] par exemple, les auteurs proposent des méthodes de réduction adaptées pour un problème de classification en utilisant un classifieur K-NN (K Nearest Neighbor). Une autre méthode de prétraitement comme l'analyse de sensibilité porte sur l'analyse de l'apport de chaque vecteur dans la phase d'apprentissage d'un réseau de neurones. Dans le cas d'analyse de sensibilité appliquée sur les réseaux de neurones, on distingue trois méthodes basées sur les poids des connexions du réseau.

1. Méthode d'équation (equation method) : l'influence de chaque entrée est calculée en utilisant cette équation :

$$I_i = \sum_k o(1-o) w_{k1}^2 v_k^2 (1-v_k^2) w_{ik}^1 \,\forall i \qquad \text{(IV.14)}$$

   o représente la sortie finale du réseau, w_{k1}^2 : représente les poids des neurones de la couche cachée, v_k^2 représente la sortie du k^ieme^ neurone caché et w_{ik}^1 représente les poids entre les nœuds de la couche d'entrée et le k^ieme^ nœud de la couche cachée. Cette méthode génère n coefficients dans le cas où on aurait n vecteurs dans la base d'apprentissage.

2. Méthode d'analyse d'amplitude des poids (weigth magnitude analysis method): cette méthode se base sur l'analyse des poids entre la couche d'entrée et de sortie. Si la connectivité d'un nœud d'entrée est élevée alors l'entrée correspondant à ce neurone est influente dans le processus d'apprentissage. Le degré d'influence de chaque entrée est calculé par cette équation :

$$I_i = \sum_k \frac{w_{ik}^1}{\max all_{i,k}(w_{ik}^1)} \qquad \text{(IV.15)}$$

3. Méthode de perturbation de variable (perturbation variable method) : cette méthode estime le taux de changement des sorties du réseau de neurones suite à la variation des entrées. Contrairement aux méthodes précédentes, l'estimation est calculée d'une façon directe (calcul à partir des valeurs entrées/sorties et non à partir des poids). Cette méthode introduit une perturbation au niveau d'une seule entrée (perturbation par addition ou multiplication par une constante). L'entrée la plus influente dans le processus d'apprentissage est celle qui génère la plus grande variation de la sortie une fois perturbée.

[49] SVM : Support Vector Machine

IV.5. Conclusion

Ce chapitre a porté sur la méthodologie générale adoptée dans cette thèse pour implémenter un système sur puce dynamique, réactif et autonome. Cette méthodologie basée sur l'analyse multi niveaux de la variabilité des composants logiciels et matériels du système. Cette variabilité est approchée par une série temporelle qui représente le comportement dynamique de l'exécution d'une tâche. Cette approche est basée sur l'insertion de prédicteurs neuronaux, dans un système reconfigurable, capables d'estimer le comportement d'une application jugée potentiellement reconfigurable.

Dans la première section de ce chapitre, nous avons proposé une décomposition en niveaux de granularité d'une description d'architectures au niveau système. Cette décomposition donne naissance à trois niveaux de granularité différents qui sont le niveau des applications, des modules et des instructions. Dans la même section, nous avons exposé la méthodologie adoptée pour explorer les différentes reconfigurations du système.

Dans la deuxième section, nous avons détaillé l'aspect variable qui apparaît dans les différents niveaux de granularité. Au niveau des applications, nous avons testé le codeur entropique de la chaîne de compression d'images fixes JPEG2000. Nous avons également analysé les caractéristiques de la trace d'exécution de cette application composée par des valeurs qui indiquent la performance de codage pour chaque segment de l'image initiale. Le deuxième test de variabilité est appliqué sur un réseau de neurones de type SOM (Self Organizing Map). Nous avons ainsi testé la variabilité au niveau des modules représenté par les neurones qui constituent le réseau SOM. Finalement, nous avons testé la variabilité au niveau le plus bas qui est le niveau des instructions. La série temporelle testée dans cette étude est la trace d'allocation d'adresses des instructions lors de l'exécution d'une application.

Dans la troisième section, nous avons exposé les techniques de prédiction développées dans la littérature. Deux types de techniques ont été analysés qui sont les techniques de prédiction statistique notamment les ARMA, les ARIMA et ceux basées sur les réseaux de neurones récurrents. Une partie de l'état de l'art des réseaux de neurones récurrents appliqués à la prédiction est également présenté dans cette section.

La robustesse des techniques de prédiction neuronales appliquées sur des séries temporelles complexes [28] associée à la flexibilité et la capacité de généralisation de ces réseaux nous a motivé à les utiliser pour prédire la performance des composants potentiellement reconfigurables implémentés dans un système réactif. Dans le cinquième chapitre nous allons détailler quelques techniques d'exploration et de synthèse automatique d'architectures embarquées de réseaux de neurones.

IV.6. Références

[1] CoreConnect : www.ibm.com/chips/products/coreconnect

[2] D.Taubman and M.W.Marcellin, 'JPEG2000 – Image Compression Fundamentals, Standards and Practice' Kluwer Academic Publishers, 2001.

[4] PIN: http://rogue.colorado.edu/Pin/

[5] PEK: http://www-128.ibm.com/developerworks/power/pek/

[6] Cross-compiler: http://www.kegel.com/crosstool/

[7] MIBENCH: http://www.eecs.umich.edu/mibench/

[8] Skabar. A, Cloete. I, 'Neural networks, financial trading and the efficient markets hypothesis', Conferences in research and practice in information technology series, Vol. 17. 2002, pp. 241-249.

[9] Ahn. C.W, Ramakrishana. R.S, 'QOS Provisioning Dynamic connection-Admission Control for Multimedia Wireless Networks Using a Hopfield Neural Network', IEEE Transactions on vehicular technology, Vol.53, No.1, 2004, pp. 106-117.

[10] Connor. J, Martin. R, Atlas. L, 'Recurrent neural network and robust time series prediction', IEEE Transactions on Neural Network, Vol. 5, 1994, pp. 240-254.

[11] Parlos. A, Rais. O, Atiya. A, 'Multi-step-ahead prediction using dynamic recurrent neural networks', Neural Networks 13, 2000, pp. 765-786.

[12] Li. X, Ho. J, Chow. T, 'Approximation of dynamical Time-variant systems by continuous-time recurrent neural networks', IEEE Transactions on Circuits and Systems II: Express Briefs, 2005, pp 656-660.

[13] Cybenko. G, 'Continuous value neural networks with tow hidden layers are sufficient', Math. Control Signals and Systems, Vol. 2, 1989, pp. 303-314.

[14] Rumelhart. D, hinton. G, Williams. R, 'Learning internal representations by error propagation', Parallel Distributed Processing Explorations in the Microstructure of Cognition, Vol. 1, 1986, pp. 318-362.

[15] Basso. M, Giarré. L, Groppi. S, Zappa. G, 'NARX Models of an Industrial Power Plant Gas Turbine', IEEE Transactions on control systems technology, Vol. 13, No.4, 2005, pp. 599-604.

[16] Hava. T, Horne. B.G, Ciles. C.L, 'Computational capabilities of recurrent NARX neural networks', IEEE Transactions on Systems, Man and Cybernetics, Vol. 27, No. 2 , 1997, pp. 208-215.

[17] Lin. T, Giles. L, Horne. B.G, Kung. S.Y, 'A Delay Damage Model Selection Algorithm for NARX Neural Networks', IEEE Transactions on Signal Processing, Vol. 45, No. 11, November 1997, pp. 2719-2730.

[18] Siegelman. H.T, Horne. B.G, and Giles. C.L, 'Computational capabilities of recurrent NARX neural networks', IEEE Transactions on Man an Cybernetics – Part B, Vol.27, No.2, 1997, pp. 208-223.

[19] Horne. B.G, Giles. C.L, 'An experimental comparison of recurrent neural network', Proceedings In Advances in Neural Information Systems Processings, 1995, pp. 697-704.

[20] Hebb. D.O, 'The organization of the behavior', New York: Wiley, 1949.

[21] Ahn. C.W, Ramakrishana. R.S, 'QOS Provisioning Dynamic connection-Admission Control for Multimedia Wireless Networks Using a Hopfield Neural Network', IEEE Transactions on Vehicular Technology, Vol. 53, No. 1, 2004, pp. 106-117.

[22] Y. Chen, B. Yang, J. Dong, A. Abraham, 'Time series forecasting using flexible neural tree model', Information sciences 174, 2005, pp. 219-235.

[23] Stephan K. Chalup, Alan D. Blair, 'Incremental training of first order recurrent neural networks to predict a context-sensitive language', Neural Networks 16 (2003), pp. 955–972.

[24] Wu. Shaun, Lu. R.P, 'Combining Artificial Neural Networks and Statistics for Stock-market Forecasting', ACM Annual Computer Science Conference, 1993, pp. 257-264.

[25] Vishwanathan, Murty, Narasimha, 'SSVM: A simple SVM Algorithm', Proceedings International Joint Conference on Neural Networks, IJCNN '02, 2002, Vol.3, pp. 2393-2398.

[26] J. C. Bezdek and L. I. Kuncheva, "Nearest prototype classifier designs: An experimental study," International Journal of Intelligent System, Vol. 16, No. 12, 2001, pp. 1445–1473.

[27] Kohonen. T: "Self-Organizing Maps and Learning Vector Quantization for Feature Sequences", Neural Processing Letters, 1999, pp. 151-159.

[28] Ho. S, Xie. M, Goh. T, "A comparative study of neural network and Box-Jenkins ARIMA modeling in time series prediction", Proceedings of the 26th International Conference on Computers and Industrial Engineering, Vol. 42, No. 2-4, 2002, pp. 371-375.

[29] QU. Y, Tiensyrja. K, Masselos. K, 'System-Level modeling of dynamically reconfigurable co-processors', Proceedings of the International Conference on FPL, 2004, pp. 881-885.

[30] Maestre. R, et al, 'A Framework for Reconfigurable Computing: Task Scheduling and Context Management', IEEE Transactions on VLSI Systems, Vol. 9, No. 6, 2001, pp. 858-873.

[31] Steiger. J, Walder. H, Platzner. M, 'Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks', IEEE Transactions on Computer, Vol. 53, No. 11, 2004, pp. 1393-1407.

[32] Leibson. S, Kim. J, 'Configurable Processors: A New Era in chip design', IEEE Transactions on Computer, Vol. 38, No. 7, 2005, pp. 51-59.

[33] Genz. C, Drechsler. R, 'System Exploration of SystemC Designs', Proceedings of the Emerging VLSI Technologies and Architectures, 2006, pp 335.

[34] S.Chtourou, M.Chtourou and O.Hammami, "Neural Network Based Memory Access Prediction Support for SoC Dynamic Reconfiguration", Proceedings of the WCCI 2006 IEEE International Joint Conference in Neural Network, July 16-21, Vancouver, Canada.

[35] S. Chtourou, O. Hammami, M. Chtourou, "Neural Network Based Variable Workload Prediction Driven System Level Design", Soumis au Journal of Systems and Software.

V. Synthèse haut niveau automatique et optimisée de prédicteurs neuronaux intégrés sur puce reconfigurable dynamiquement

V.1. Introduction

Dans le chapitre précédent, nous avons présenté les méthodologies d'implémentation de systèmes sur puce reconfigurables et réactifs au niveau système. Le flot de conception proposé est basé sur l'insertion de prédicteurs neuronaux dans la plateforme reconfigurable. Le système de prédiction estime les performances d'un composant jugé potentiellement reconfigurable. L'introduction d'un module de prédiction dans l'architecture engendre la surcharge de la plateforme par un nouveau composant qui ne représente pas réellement la fonctionnalité de l'application à implémenter. La surcharge de la plateforme implique sa pénalisation en termes de la performance d'exécution, de la surface d'implémentation et de la consommation d'énergie. De ce fait, nous proposons dans ce chapitre une méthodologie d'exploration capable de parcourir l'espace des architectures possibles de réseaux de neurones. Deux volets seront considérés dans ce contexte :

1. Dans le premier volet, nous nous intéressons à l'exploration des architectures synthétisées à partir d'une description comportementale d'un accélérateur matériel. Le flot d'exploration proposé dans ce contexte est applicable sur toutes les descriptions comportementales.
2. Dans le deuxième volet, nous considérons l'exploration des architectures des réseaux de neurones appliqués à la prédiction. Le flot d'exploration proposé est applicable uniquement sur des descriptions d'architectures neuronales.

Dans ce chapitre, nous allons présenter le flot du compilateur Cocentric SystemC Compiler de Synopsys [1] utilisé pour la synthèse d'une description SystemC. Nous allons également exposer un flot de synthèse basé sur plusieurs outils industriels afin d'explorer la surface, la performance et la consommation d'énergie des différentes implémentations à partir d'un niveau SystemC comportemental. Le deuxième flot implémente une exploration dirigée par un algorithme d'optimisation multi objectifs. Nous illustrons dans ce contexte l'effet de l'architecture sélectionnée d'un réseau de neurones sur sa performance de prédiction et sur le coût de son implémentation.

V.2. Exploration des architectures SOPC : motivations et objectives

L'étape d'exploration des architectures d'un système sur puces a été évoquée brièvement dans le premier chapitre. Ce problème est de plus en plus complexe surtout avec l'émergence de plusieurs méthodologies de conception, de synthèse et d'implémentation. Un mécanisme d'exploration peut être introduit à plusieurs niveaux :

- Niveau système : il s'agit d'explorer les différentes architectures possibles au niveau de partitionnement matériel/logiciel, d'ordonnancement des tâches ou au niveau du choix de la plateforme de communication (bus de communication, réseau sur puce ou mémoire partagée).
- Niveau synthèse : il s'agit d'explorer les contraintes qui gèrent l'opération de synthèse. Par définition, la synthèse d'une application matérielle est le passage d'un niveau de description simulable (comportemental ou RTL) à une description matérielle (netlist ou fichier de composants).
- Niveau implémentation : il s'agit d'explorer les technologies d'implémentation possible ainsi que les contraintes de placement et de routage du design.

Généralement, le but de l'exploration est de détecter les coûts d'implémentation et les performances de plusieurs implémentations possibles. Le coût est représenté par la surface d'implémentation de l'architecture ainsi que sa consommation d'énergie. La performance d'exécution du design est représentée par la fréquence du design et par le nombre de cycles nécessaire pour un traitement souhaité. Les objectifs à explorer sont dépendants, c'est à dire si on augmente, par exemple, la fréquence d'exécution alors on augmente la consommation d'énergie du système embarqué. Dans les travaux présentés dans la première partie de ce chapitre, nous nous intéressons à l'exploration au niveau synthèse effectuée à partir d'une description SystemC [1]. Dans la section suivante, nous détaillons le flot de conception et de synthèse d'une description SystemC.

V.3. Flot de synthèse et de conception comportementale SystemC

L'apparition de l'outil Cocentric SystemC Compiler (CSCC[50]) [2] effectuant la synthèse d'une description SystemC offre la possibilité d'implémenter un accélérateur matériel à partir d'un niveau SystemC comportemental. Le flot de synthèse admet une description SystemC comportementale en entrée. Le compilateur SystemC génère une architecture matérielle en se basant sur une méthodologie de synthèse architecturale à partir d'un niveau fonctionnel. Dans ce paragraphe, nous présentons les méthodologies de synthèse et de conception présentées par le compilateur SystemC.

V.3.1. Flot de synthèse comportemental SystemC

Le flot de synthèse (Fig. V.1) SystemC est composé de plusieurs étapes qui s'exécutent séquentiellement. L'entrée du compilateur est une description comportementale SystemC en plus de la spécification de deux bibliothèques : la première bibliothèque est appelée 'synthetic library' contenant des descriptions matérielles optimisées de composants logiques de base (additionneurs, multiplexeurs, comparateurs, …). Le choix entre les différents composants ne se fait pas uniquement par l'outil mais aussi par l'utilisateur. En effet, l'utilisateur peut spécifier des options de synthèse qui déterminent les composants élémentaires à utiliser dans l'architecture. La deuxième bibliothèque est appelée 'technology library' qui spécifie les contraintes de la technologie à utiliser dans la synthèse.

[50] CSCC : Cocentric SystemC Compiler

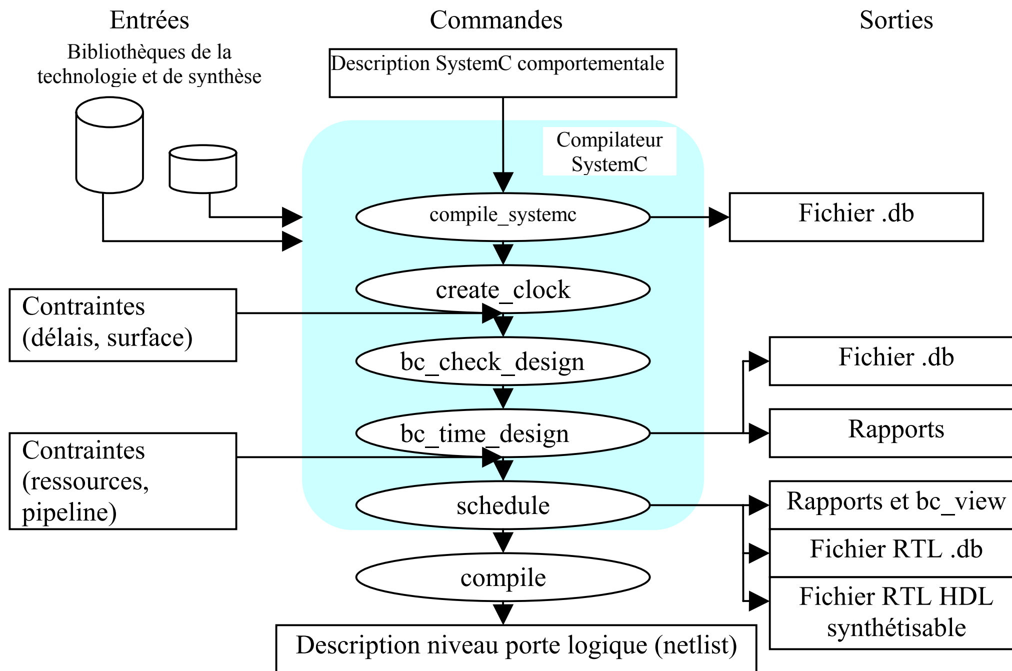


Fig. V. 1 : Flot de synthèse comportementale

L'opération de synthèse sélectionne les composants logiques qui traduisent le code SystemC en une description matérielle. Les résultats possibles du flot de synthèse SystemC sont énumérés par la liste suivante :

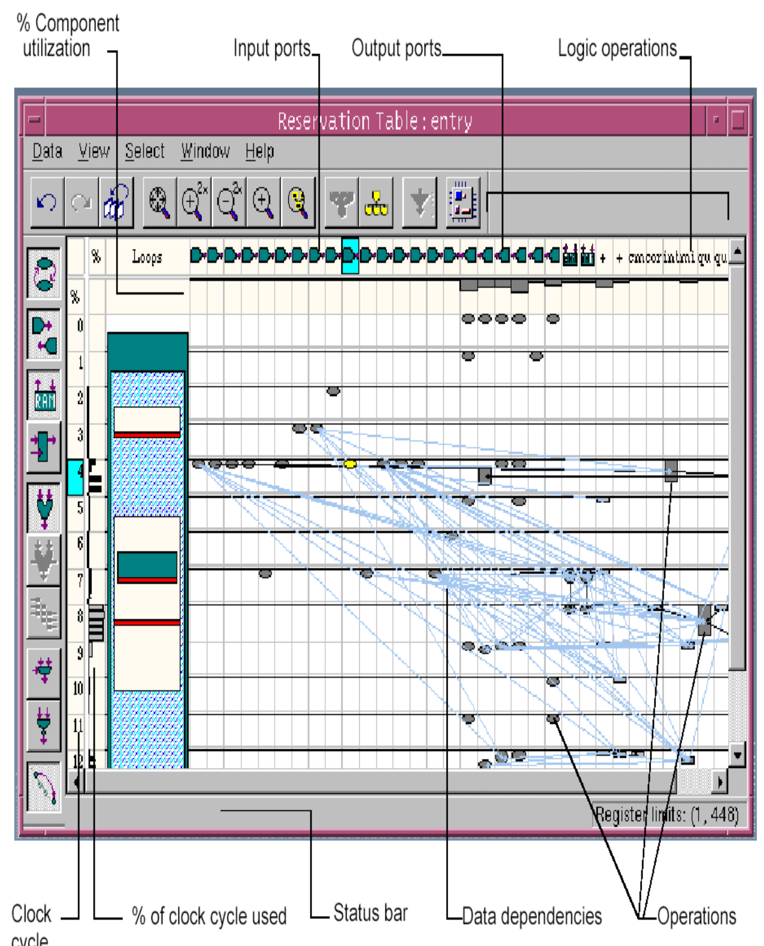
- Le premier résultat possible est une description HDL (VHDL ou Verilog) qui implémente les mêmes fonctionnalités qui existent dans la spécification SystemC. La description HDL est utilisée pour vérifier et implémenter le comportement de l'accélérateur matériel.
- Un fichier de base de données d'extension '.db' qui regroupe les composants sélectionnés par le compilateur pour construire un accélérateur matériel. Le fichier '.db' représentant l'architecture du composant associé à des contraintes de synthèse, telles que les contraintes de surface d'implémentation et de performance d'exécution, constitue les entrées de l'étape de la synthèse physique. Cette dernière est effectuée en utilisant les outils "FPGA Compiler II" [3] ou "physical Compiler" [4] de Synopsys.
- Des rapports de synthèse qui déterminent les composants utilisés dans l'architecture ainsi que leurs caractéristiques. Les rapports générés à la fin du processus de synthèse comportent des estimations de la surface d'implémentation et de la fréquence d'exécution de l'architecture synthétisée.
- Une interface graphique générée par l'outil bc_view qui décrit le design. La représentation graphique du bc_view regroupe les ressources (ports d'entrée/sortie, unité de calcul, registres, …) allouées par le compilateur pour effectuer le traitement désiré. Les liens entre les différentes ressources représentent les relations de dépendance entre elles. La notion du temps est illustrée en associant un nombre de cycles de traitement à chaque ressource déployée.

La synthèse comportementale se base sur une description SystemC. Cette opération de synthèse s'effectue suite à l'exécution séquentielle des tâches suivantes :

1. Etape de compilation : consiste à vérifier le syntaxe C et SystemC de la description. Le compilateur examine également la compatibilité entre la description à synthétiser et les contraintes d'implémentation au niveau matériel.
2. Etape d'association de délais d'exécution (timing): avant l'exécution de l'étape du 'timing', l'utilisateur sélectionne une période pour l'horloge du système. La primitive 'bc_time_design' décompose la spécification en composants élémentaires. Chaque composant est représenté par un élément de la bibliothèque 'synthetic library'. Le compilateur fixe ainsi un délai d'exécution pour chaque opération élémentaire.
3. Etape d'ordonnancement : consiste à organiser l'ordre d'exécution des différentes tâches élémentaires pour assurer un maximum de performance. L'étape d'ordonnancement vérifie également les contraintes de dépendance et d'occurrence des entrées et des sorties. Durant la phase d'ordonnancement, CSCC minimise le nombre de cycles horloge en favorisant le partage des ressources matérielles.
4. Etape d'allocation : allocation des registres mémoires pour stocker les variables, les signaux et les résultats intermédiaires.
5. Etape de synthèse : création d'un schéma complet pour la circulation de l'information à l'intérieur de la plateforme. Ceci est assuré par l'interconnexion des composants alloués du 'synthetic library' et des registres par des multiplexeurs ou par différents types de connexion.

V.3.2. Flot de conception SystemC

L'outil CSCC de Synopsys propose une méthodologie de conception basée sur une procédure itérative et manuelle de synthèse. La figure V.2 expose cet aspect itératif qui caractérise le processus de conception.



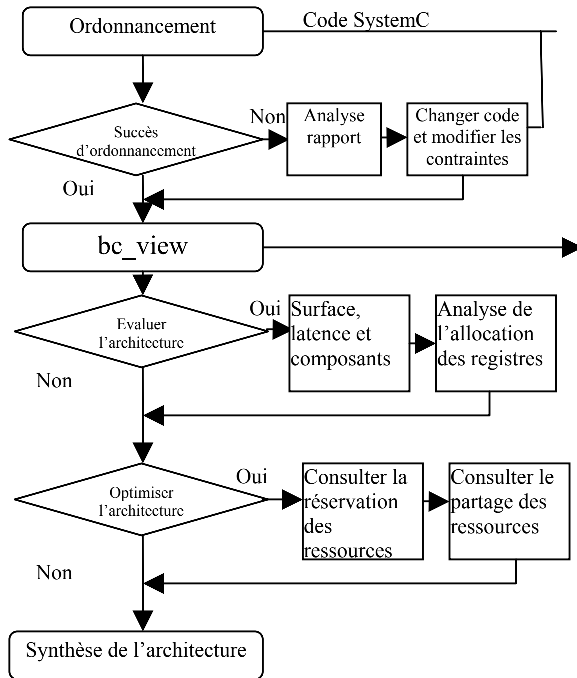


Fig. V. 2: Flot de conception à partir d'un niveau de description comportementale

L'outil CSCC effectue, en premier lieu, l'ordonnancement des tâches de l'application. Une fois effectué, le concepteur utilise l'utilitaire graphique bc_view pour évaluer les résultats de synthèse du système. Si le concepteur détecte une défaillance au niveau de la performance de l'architecture, alors il reprend le même cycle en changeant le code SystemC ou les options de synthèse. Dans le tableau V.1, nous regroupons une partie des options de compilation qui agissent sur les résultats de synthèse d'une description SystemC comportementale.

Table. V. 1 : Options de synthèse supportées par CSCC

Synopsys SystemC Compiler version U-2003.06		
Préférence	Etape de synthèse	Possibilités
exécution Spéculative	Etape de compilation	True/ false
Timing	Timing	Fastest/ force
Input delay	Timing	Préférence du concepteur
Io_mode	Timing	Super/cycle
Effort level	Ordonnancement	Low/medium/high
Extend_latency	Ordonnancement	True/false
Allocation_effort	Ordonnancement	Low/medium/high
Map_effort	Allocation	Low/medium/high
Période d'horloge	Timing	Préférence du concepteur

Les options présentées dans le tableau V.1 affectent les différentes étapes de compilation d'une description SystemC (voir Fig.V.1). Dans ce qui suit, nous détaillons l'effet de quelques options de synthèse.

- L'exécution spéculative est une technique de compilation. Elle permet l'accélération de l'exécution d'une structure conditionnelle. L'exécution spéculative permet d'exécuter les traitements d'une structure conditionnelle avant d'évaluer le résultat de la condition de branchement. La figure V.3 expose l'effet de l'exécution spéculative sur un code SystemC.

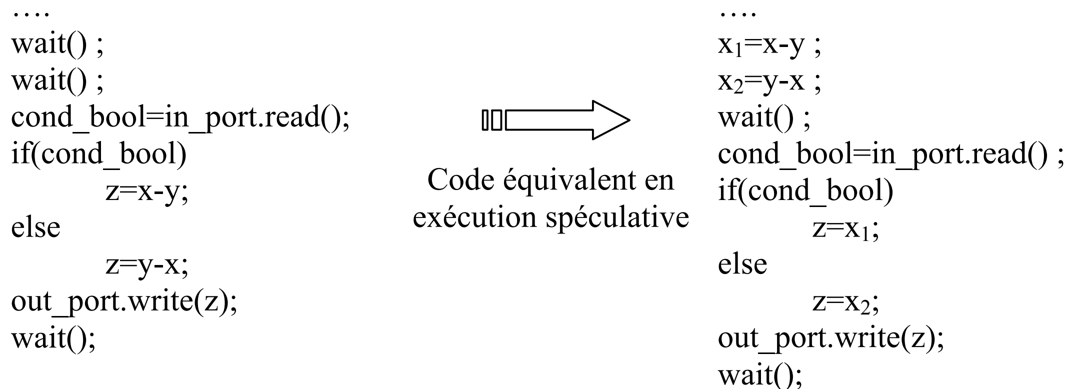
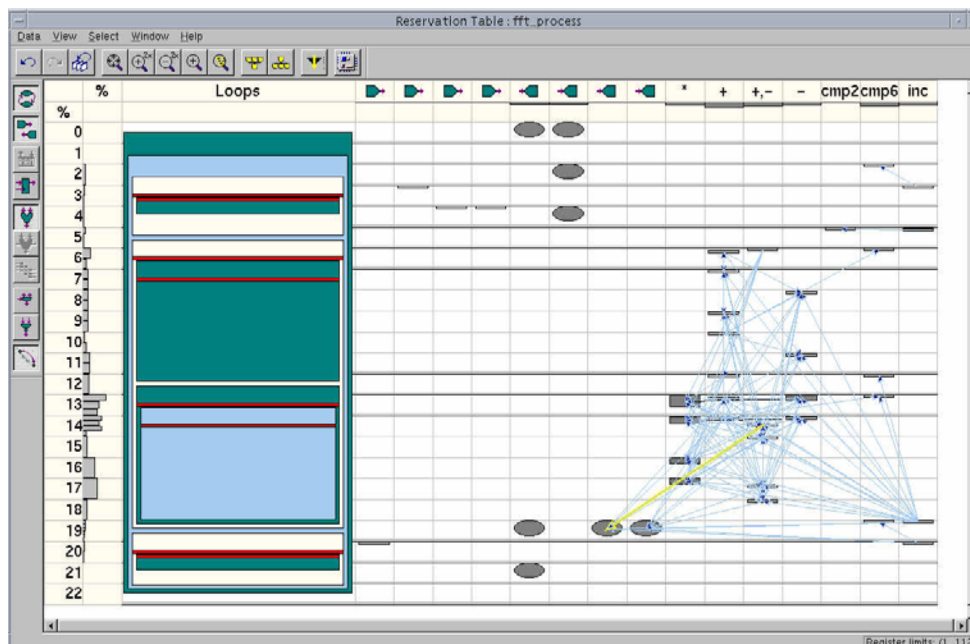


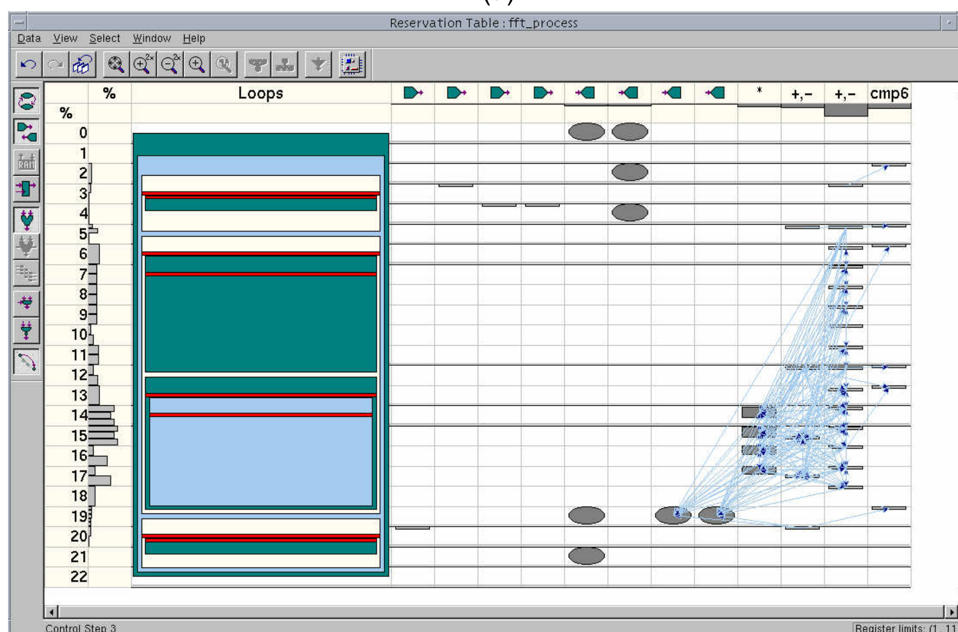
Fig. V. 3 : Effet de la compilation en exécution spéculative sur un code SystemC

- La commande Timing accepte deux valeurs d'option possibles qui sont 'fastest' ou 'force'. Si on sélectionne la valeur 'fastest', le compilateur favorise alors l'utilisation de composants élémentaires à exécution rapide. Dans ce cas, le compilateur favorise également la duplication des ressources afin d'accélérer l'exécution. De ce fait, la sélection de la valeur 'fastest' implémente des architectures performantes en termes d'exécution. En revanche, si on sélectionne l'option 'force', le compilateur optimise l'architecture afin de minimiser la surface d'implémentation.
- L'option 'effort_level' est initialisée au niveau de l'étape d'ordonnancement. Elle admet trois valeurs possibles (low/medium/high). Si on sélectionne la valeur 'high', le compilateur optimise au maximum le partage des ressources afin de réduire le nombre de composants alloués et par la suite la surface d'implémentation du design.

Nous remarquons que les options exposées dans le tableau V.1 affectent la composition de l'architecture synthétisée à partir d'une description comportementale SystemC. Dans la figure V.4, nous montrons l'effet de la période d'horloge sélectionnée manuellement par l'utilisateur sur l'architecture synthétisée. Ces effets sont détectés en utilisant l'utilitaire graphique de 'bc_view'. L'application synthétisée dans ce cas est une description SystemC effectuant la transformée de Fourrier rapide FFT appliquée sur 16 coefficients.



(a)



(b)

Fig. V. 4 : Effet de la période sur l'architecture synthétisée : (a) synthèse en fixant la période à 30 ns, (b) synthèse en fixant la période à 22ns.

Les points faibles de cette méthodologie de conception sont :

1. L'aspect manuel de l'exploration des architectures générées par la synthèse d'une description comportementale SystemC.
2. Cette exploration détermine la composition des différentes architectures mais elle ne fournit pas d'estimation sur la performance et le coût (surface, consommation d'énergie) de l'application implémentée.

Dans le paragraphe V.4, nous allons présenter un nouveau flot automatique qui génère des estimations de performance et de coût. Ce flot gère plusieurs outils professionnels de synthèse et de CAO afin d'explorer automatiquement l'espace des architectures synthétisées.

V.4. Flot d'exploration automatique surface/performance

Le travail effectué dans cette section s'intéresse à l'exploration automatique des différentes architectures générées par l'opération de synthèse. L'exploration est effectuée en changeant les valeurs des options de compilation. En effet, l'analyse des possibilités du compilateur SystemC CSCC a montré la flexibilité de cet outil. L'idée de ce travail est de concevoir et d'implémenter un environnement d'exploration capable de synthétiser plusieurs architectures différentes à partir d'une description SystemC comportementale. Le pilotage des architectures synthétisées est effectué d'une manière exhaustive à travers les options de synthèse supportées par le compilateur.

V.4.1. Flot d'exploration de descriptions SystemC

Le flot d'exploration automatique (Fig.V.5) développé dans le cadre de ce travail se base sur un module d'exploration exhaustive qui admet deux entrées :

1. La liste des options de synthèse explorées automatiquement à travers un module qui teste d'une manière exhaustive toutes les combinaisons des options possibles.
2. Une description SystemC au niveau comportemental synthétisable c'est à dire qui comporte des structures supportées [2] par CSCC.

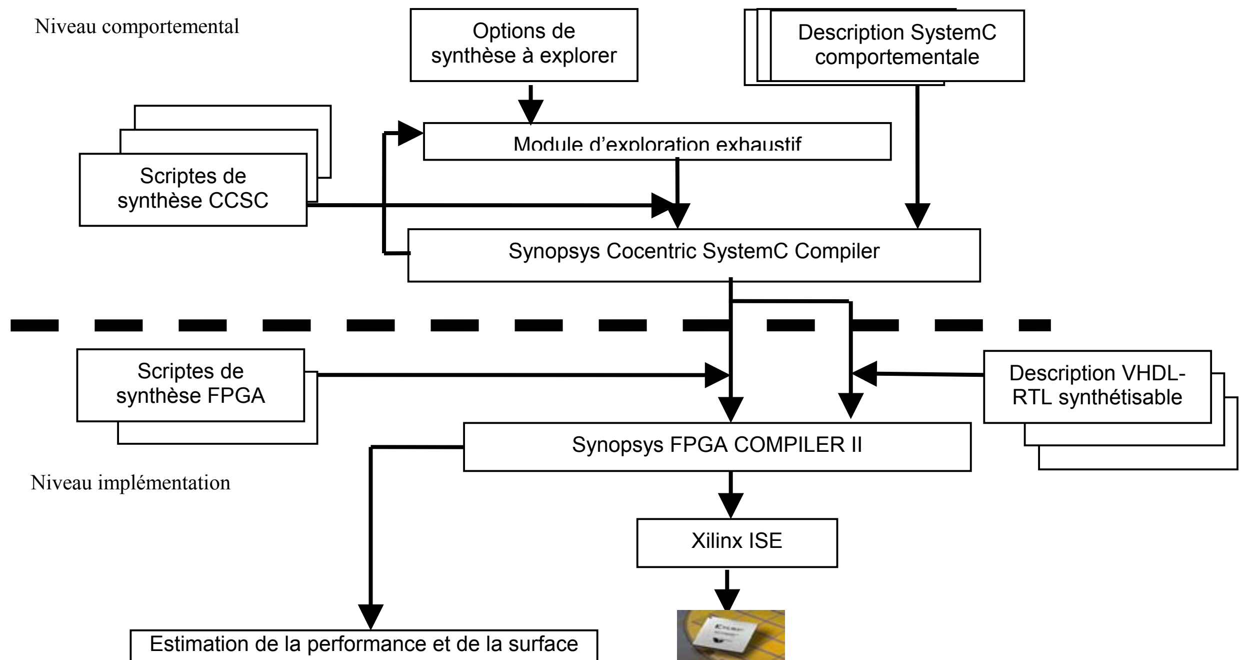


Fig. V. 5 : Flot d'exploration automatique à deux dimensions (surface, performance) des architectures synthétisées au niveau SystemC comportemental

Le flot d'exploration de la figure V.5 pilote les outils et fixe les options de synthèse à travers des scripts. Le module d'exploration exécute une opération de synthèse pour chaque scripte généré automatiquement. Un scripte de synthèse CSCC comporte des commandes supportées par le compilateur SystemC. Dans la figure V.6, nous exposons un exemple de scripte généré automatiquement. L'exemple synthétisé dans le scripte de la figure V.6 est le FFT. La cible de

la synthèse est une plateforme FPGA de type ‘VIRTEXE’. Ce scripte est exécuté automatiquement sur un terminal Linux.

```
search_path= search_path + "$SYNOPSYS/libraries/syn"
target_library    = {"tc6a_cbacore.db"};
synthetic_library = {"dw01.sldb","dw02.sldb"}
link_library      = {"*"} + target_library + synthetic_library
bc_enable_analysis_info    = "true"
bc_enable_speculative_execution    =false
effort_level=low
io_mode=super
top_unit="fft.cpp"
compile_systemc top_unit
set_fpga -target VIRTEXE -device V405EBG560 -speed "-6"
create_clock clk -p 30
set_operating_conditions typ_25_5.00
set_input_delay 0.1 -clock clk all_inputs() - clk
bc_time_design -fastest
schedule -io io_mode -effort effort_level -allocation_effort
low
report_schedule -sum > fir_beh_schedule.rpt
write_rtl -format vhdl -output design_vhdl_0.vhdl -rtl_script
script_fc2_0.scr
 exit
```

Fig. V. 6: Scripte de synthèse supporté par CSCC

A la suite de l’exécution du scripte supporté par CSCC, nous obtenons deux entités de sortie suite à l’exécution de la commande ‘`write_rtl -format vhdl -output design_vhdl_0.vhdl -rtl_script script_fc2_0.scr`’ :

1. Un ensemble de scriptes de commandes supportées par l’outil ‘FPGA compiler II’ [3] de Synopsys. Dans ce cas, le fichier scripte est appelé ‘script_fc2_0.scr’.
2. Un ensemble de codes VHDL générés automatiquement par l’outil CSCC. Les descriptions VHDL sont décrites au niveau RTL (Register Transfer Level) et sont directement synthétisables sur la plateforme cible. Dans ce cas, le fichier de description VHDL est appelé ‘design_vhdl_0.vhdl’.

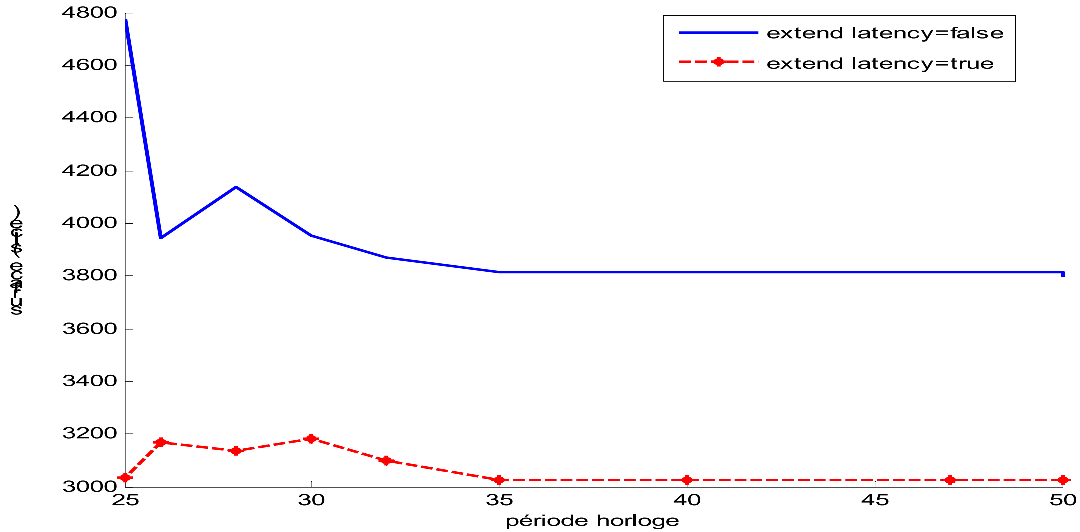
L’outil ‘FPGA compiler II’ compile la description VHDL en exécutant les commandes existant dans le scripte de synthèse FPGA. Le résultat final de ce flot d’exploration est un ensemble de rapports de synthèse. Ces rapports comportent les estimations de performance et de surface des différentes architectures synthétisées.

V.4.2. Résultats d’exploration en performance/surface

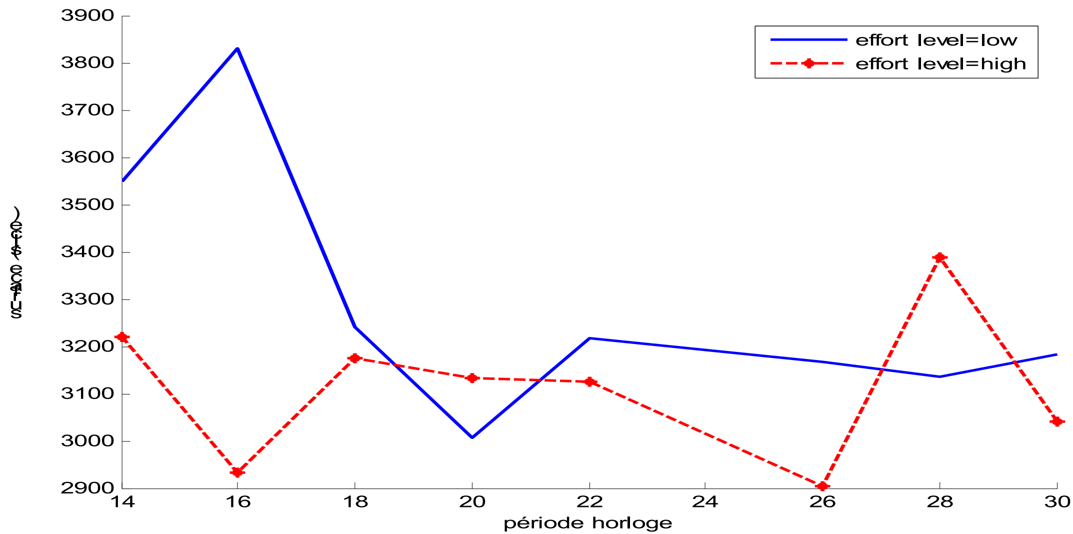
Les résultats de performance et de surface sont regroupés à la fin de l’exécution de la totalité des scriptes générés. La figure V.7 expose quelques résultats d’exploration 2D (surface, performance) de la synthèse de différentes applications. Les applications testées sont le FFT et le FIR qui exécutent respectivement la transformée de Fourrier rapide et une technique de filtrage récursive. Le tableau V.2 regroupe les options explorées dans les différents graphes.

Table. V. 2: Options explorées dans la figure V.7

GRAPHE	OPTIONS
Graphe 1 (G1)	Ordonnancement: Extend latency = true/ false
Graphe 2 (G2)	Ordonnancement: effort level= low/ high



G1 : Synthèse du FFT (FPGA) - surface en fonction de la période (extend_latency)



G2 : Synthèse du FFT (FPGA) - surface en fonction de la période (effort_level)

Fig. V. 7: Exploration des options d'ordonnancement sur la synthèse de la FFT: (G1) : effet de l'option extend_latency, (G2) : effet de l'option effort_level

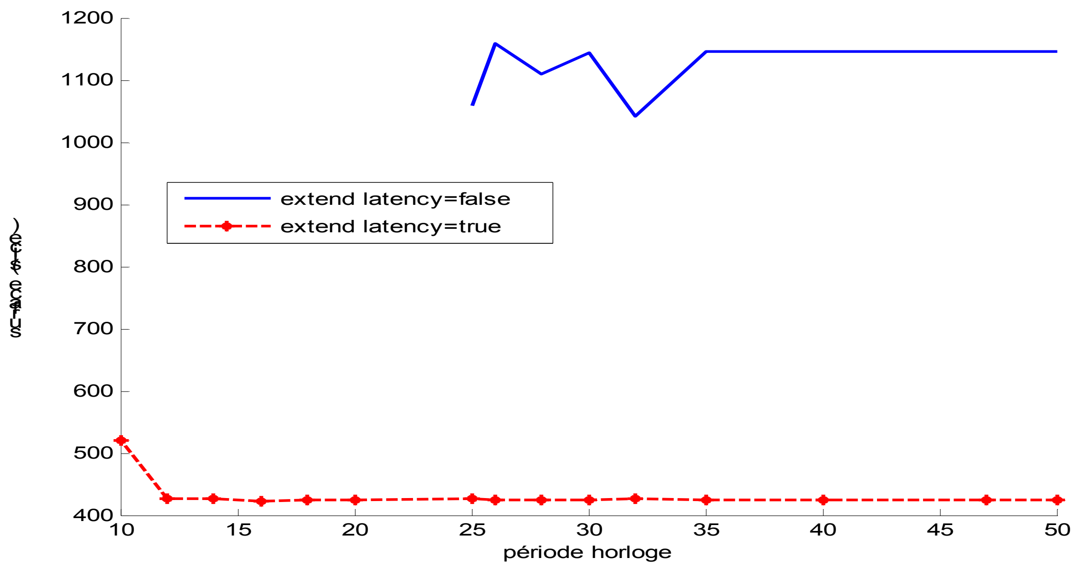
La figure V.7 (G1) illustre l'effet de l'option 'extend_latency' sur l'opération de synthèse. En effet, si on initialise 'extend_latency' à 'true', le compilateur favorise l'optimisation de la surface par rapport à la performance de l'architecture synthétisée. D'où la synthèse d'une architecture qui occupe moins de logique (Fig.V.7 (G1)).

En revanche, l'option 'effort_level' fixe l'effort fournit par le compilateur pour organiser les composants de l'architecture (allocation, réutilisation et partage des ressources). Nous remarquons qu'en fixant l'option 'effort_level' au niveau minimum (low), alors la surface de l'architecture synthétisée augmente (Fig.V.7 (G2)). Dans la figure V.8 et le tableau V.3, nous explorons l'effet des options 'extend_latency' et 'effort_level' sur la synthèse de la description SystemC comportementale du FIR.

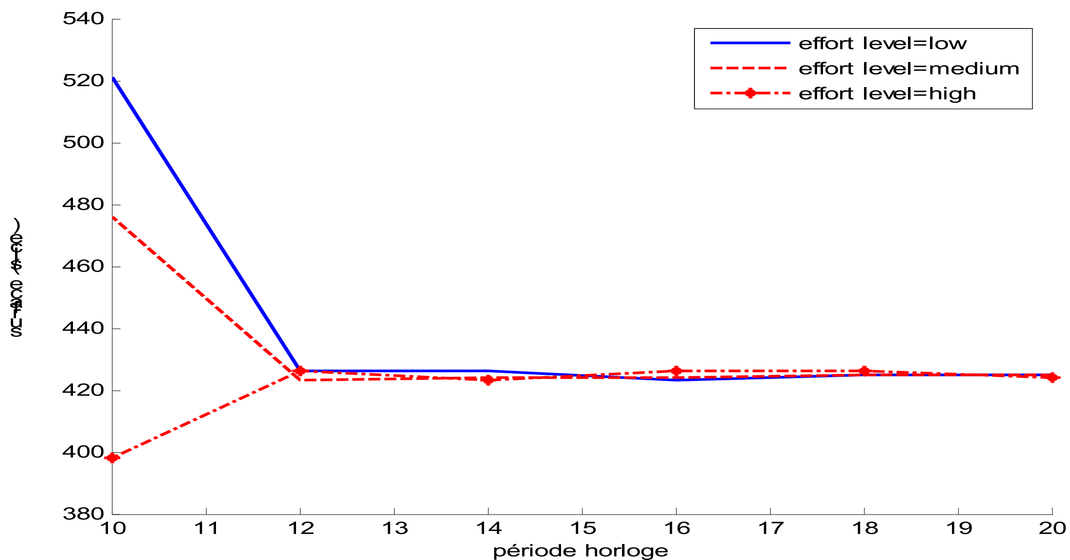
Table. V. 3: Options explorées dans la figure V.8

GRAPHE	OPTIONS
Graphe 1 (G1)	Ordonnancement: Extend latency = true/ false

Graphe 2 (G2)	Ordonnancement: effort level= low/ medium/ high



G1 : Synthèse du FIR (FPGA) – surface en fonction de la période (extend_latency)



G2 : Synthèse du FIR (FPGA) – surface en fonction de la période (effort_level)

Fig. V. 8 : Exploration des option d'ordonnancement sur la synthèse du FIR: (G1) : effet de l'option extend_latency, (G2) : effet de l'option effort_level

Nous remarquons que l'effet des options de compilation sur les résultats de synthèse dépend de l'application à synthétiser. En effet, il est clair que l'option 'effort_level' optimise d'avantage la surface d'implémentation du FFT. L'effet de cette option est négligeable sur les résultats de synthèse du FIR (Fig.V.8 (G2)). Dans la figure V.8 (G1), nous observons l'effet de l'option 'extend_latency' sur les résultats de synthèse.

Dans la figure IV.9, nous observons les effets de l'adoption de la technique d'exploration spéculative dans la synthèse du FFT. L'exploration spéculative pénalise la surface puisqu'elle duplique les traitements des structures conditionnelles.

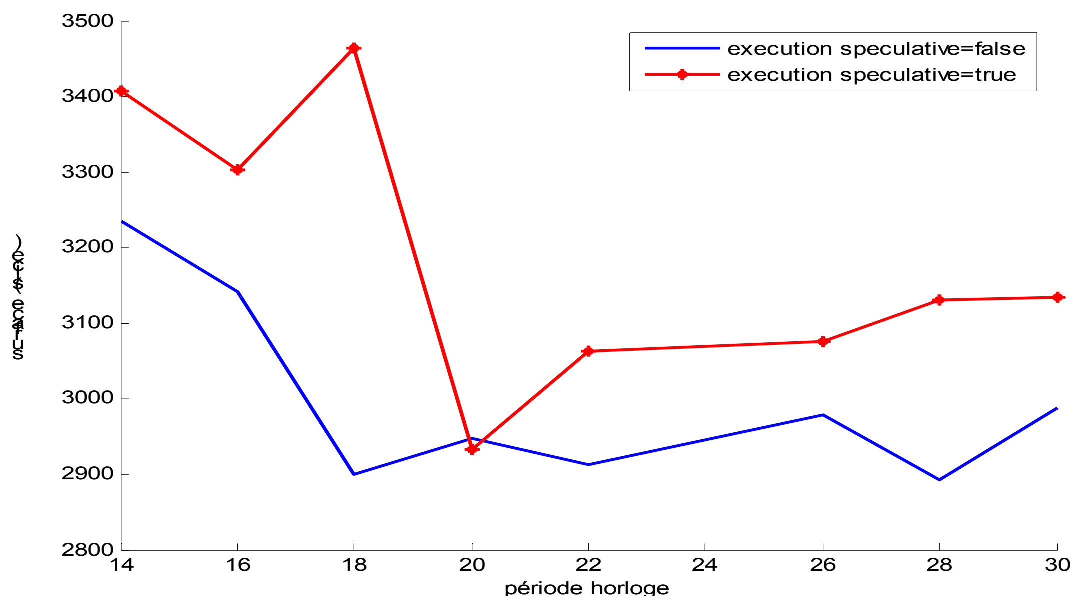


Fig. V. 9 : Effet de l'option permettant l'exécution spéculative sur les résultats de synthèse

Le flot exposé dans ce paragraphe explore les résultats de performance et de surface pour les différentes options de synthèse [9]. Les informations sur la consommation d'énergie restent inconnues à ce niveau. Dans la section suivante, nous nous intéressons à présenter un flot d'exploration à trois dimensions (surface, performance et consommation d'énergie).

V.5. Flot d'exploration automatique surface/performance/ consommation d'énergie

Les résultats présentés dans la section précédente montrent la relation qui existe entre les options de compilation et l'architecture synthétisée. Cette relation est illustrée par une variation da la surface de l'architecture synthétisée à la suite de l'adoption d'une nouvelle technique de compilation ou d'un changement de la période de l'horloge du système. Dans cette section, nous présentons un flot d'exploration qui parcourt les combinaisons possibles de plusieurs options sélectionnées. Ce flot détecte l'influence des options de synthèse sur la surface, la performance et la consommation d'énergie estimées des architectures synthétisées à partir d'un niveau SystemC comportemental [10].

V.5.1. Estimation de la consommation d'énergie

Les outils standards d'estimation de la consommation d'énergie comme 'Power Compiler' [5] de Synopsys effectuent l'analyse d'une architecture décrite au niveau RTL. L'outil 'power compiler' admet trois entrées :

1. Un fichier de description HDL simulable et synthétisable de l'application au niveau RTL.
2. Un fichier d'activité généré automatiquement par la simulation de la description ou manuellement par l'utilisateur. Le fichier appelé SAIF[51] généré automatiquement par le simulateur décrit l'activité des composants de l'architecture en cas de fonctionnement.
3. Un fichier de contraintes.

[51] SAIF : Switching Activity Interchange File

Le schéma de la figure V.10 représente l'environnement dc_schell[52] qui supporte le flot de compilation adopté par 'Power Compiler' pour analyser, optimiser et estimer la consommation d'énergie d'une description RTL.

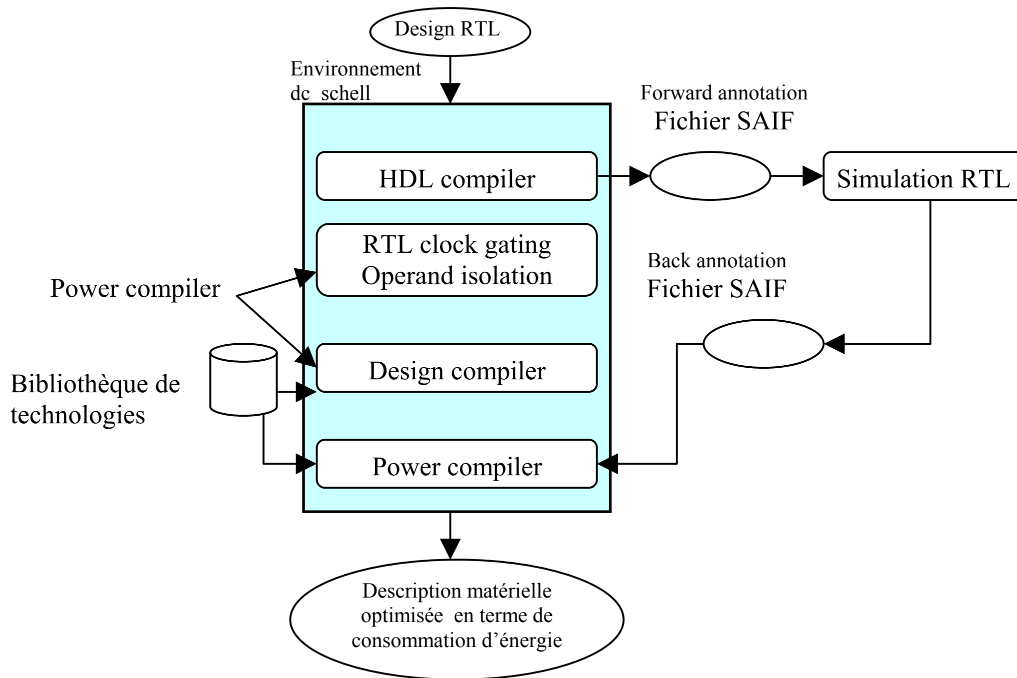


Fig. V. 10 : Flot d'estimation de la consommation d'énergie en utilisant 'power compiler'

Le compilateur HDL génère la description matérielle à implémenter (netlist). Ensuite, le flot exécute une phase appelée 'clock gaiting' qui représente une étape d'optimisation de l'architecture du design au niveau RTL. Le compilateur élimine dans ce cas les registres présents qui ne changent pas de valeur durant plusieurs cycles d'horloge. Finalement, le flot utilise l'outil 'power compiler' pour générer une architecture optimisée au niveau de la consommation d'énergie. Le flot d'estimation de la consommation d'énergie est intégré dans le flot d'exploration automatique développé dans la section V.4.

V.5.2. Flot d'exploration en performance/surface/consommation d'énergie

Dans la figure V.11, nous présentons le flot d'exploration à trois dimensions (surface, performance, consommation d'énergie) des architectures synthétisées à partir d'un niveau SystemC comportemental. Ce flot est une extension du flot présenté dans la figure V.5. L'extension simule la description VHDL-RTL synthétisée. Elle enregistre l'activité de l'architecture dans un fichier d'extension '.vcd'. Finalement, elle génère le fichier SAIF utilisé par le compilateur 'power compiler' pour estimer la consommation d'énergie de la description VHDL-RTL de l'architecture. L'exploration des différentes architectures est effectuée d'une manière automatique.

[52] dc_schell : Design Compiler_schell

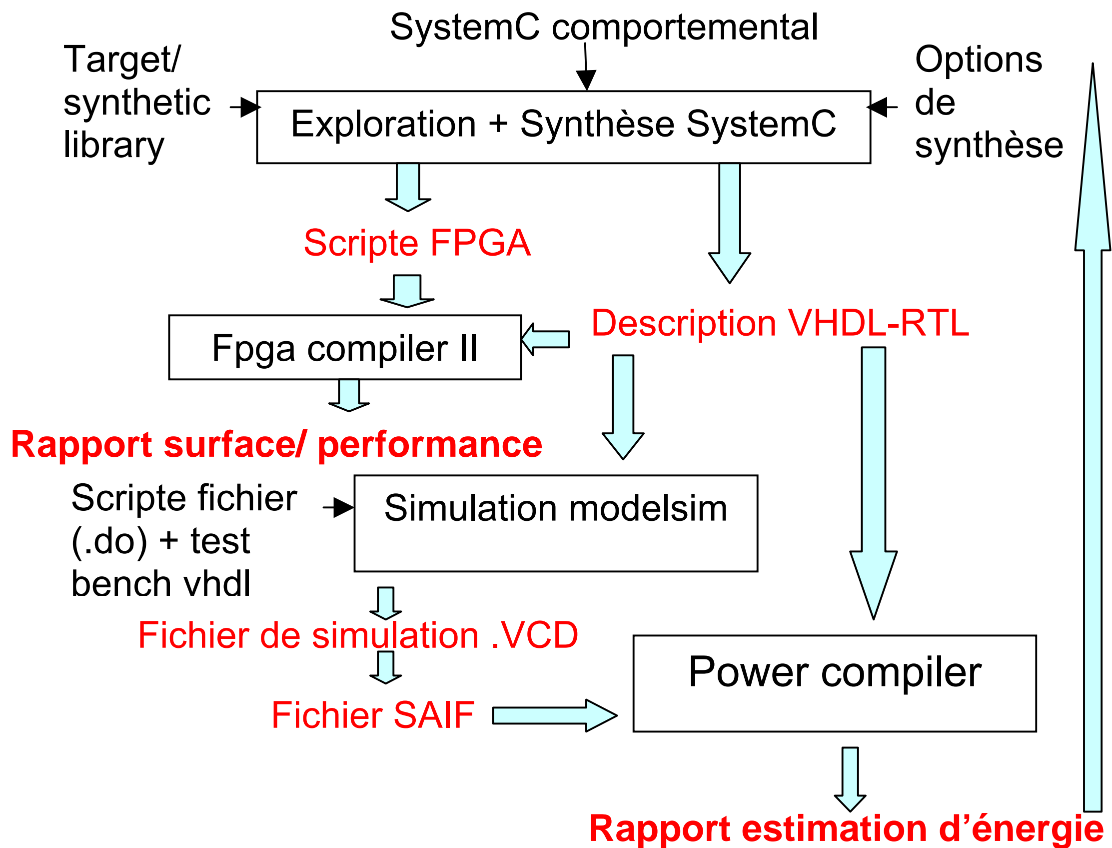


Fig. V. 11 : Flot d'exploration automatique à trois dimensions (surface, performance, consommation d'énergie) des architectures synthétisées au niveau SystemC comportemental

Pendant la phase de simulation, le flot utilise l'outil Modelsim [6] de 'Mentor Graphics'. Cet outil admet deux entrées qui sont un scripte de simulation contenant les commandes exécutées par Modelsim et un code VHDL qui génère des vecteurs de test (test bench). La simulation Modelsim génère un fichier '.vcd' qui peut être recompilé en un fichier '.SAIF'. Ce dernier représente l'activité de l'application en simulation. Le modèle RTL et le fichier '.SAIF' représentent les deux entrées de l'outil 'Power Compiler' de Synopsys [7]. L'outil estime la consommation d'énergie de l'application pour les différentes conditions fixées par le scripte de synthèse SystemC.

Un exemple de scripte est présenté par la figure V.12. Le module simulé est l'application FFT synthétisée par CSCC. La description VHDL-RTL est générée dans le fichier 'design_vhdl_1.vhdl'. Le module 'testfft2.vhd' représente le banc de test de l'application.

```
vlib work
vcom -93 -explicit design_vhdl_1.vhdl
vcom -93 -explicit testfft2.vhd
vsim -t 1ps -lib work fft_module_testfft2_vhd_tb
vcd file sim_script_1.vcd
vcd add *
run 1000000 ns
quit -f
```

Fig. V. 12 : Scripte de simulation par Modelsim

La simulation de la netlist de l'application représentée par le module 'fft_module_testfft2_vhd_tb' est effectuée en exécutant la commande 'vsim'. Le résultat de l'exécution de ce scripte est un fichier '.vcd'. La commande 'run' simule la description et détermine le temps de simulation de l'architecture en nano seconde.

V.5.3. Résultats d'exploration en performance/surface/ consommation d'énergie

La figure IV. 10 présente les résultats d'exploration de synthèse pour les différentes valeurs des options 'extend_latency' et 'speculative_execution'. Les applications testées dans cette section sont l'application FFT (Fast Fourrier Transform) et un réseau de neurones de types 'Self Organizing Map' (SOM). Dans ces explorations, l'énergie est exprimée en milli Watt (mw). La période et la surface sont exprimées respectivement en nano seconde (ns) et en micro mètre carré (μm^2)

V.5.3.1. Exploration de la synthèse des architectures FFT

La FFT effectue la transformée de Fourrier sur 16 composants. Le composant implémentant la FFT admet quatre entrées. Ces entrées sont représentées par deux entiers codés sur 16 bits correspondant aux parties réelle et imaginaire d'un coefficient complexe, un port pour transmettre le signal de l'horloge et un port de remise à zéro (reset). La sortie est représentée par deux ports qui génèrent les parties réelle et imaginaire du coefficient résultat de la transformée de Fourrier. La figure V.13 représente les résultats d'exploration des options effectuant la compilation en exécution spéculative et en 'extend_latency'.

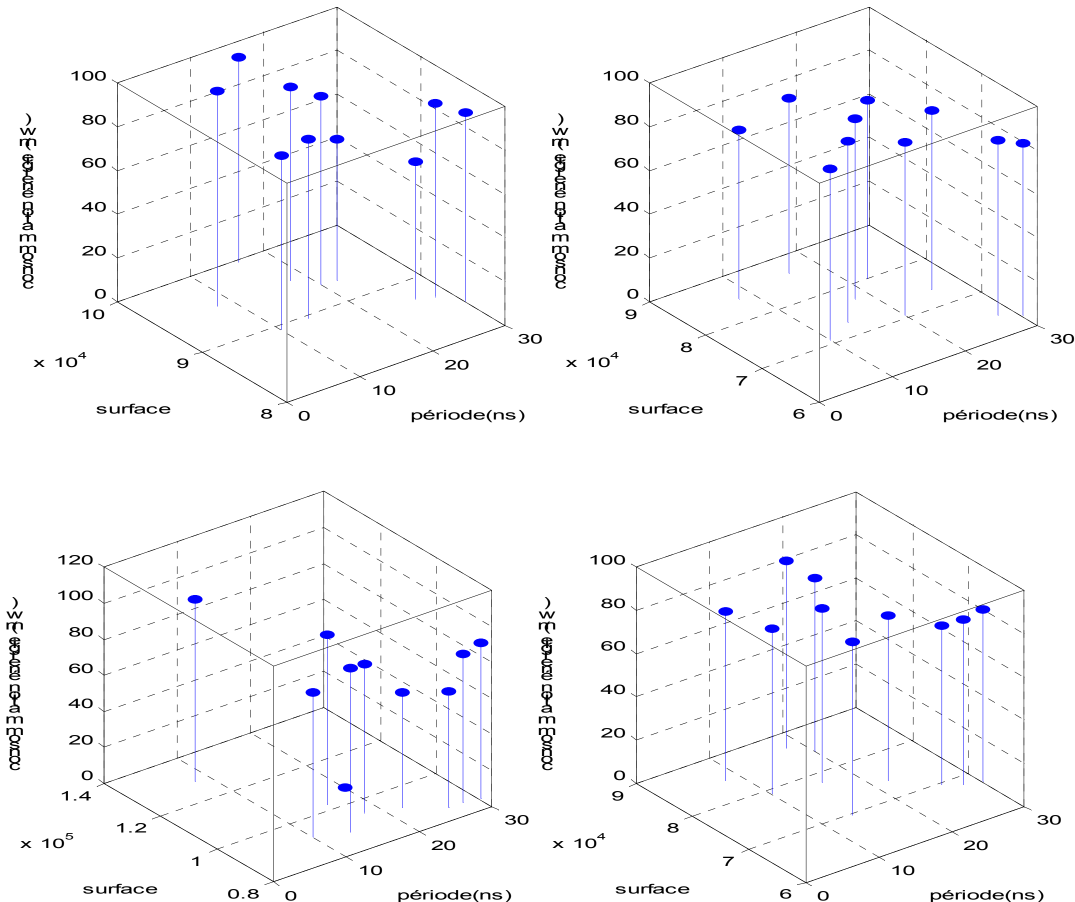


Fig. V. 13 : Exploration à trois dimensions de la synthèse de la FFT : (a) cas speculative_execution=false et extend_latency=false, (b) cas speculative_execution=false et extend_latency=true, (c) cas speculative_execution=true et extend_latency=false, (d) cas speculative_execution=true et extend_latency=true

Nous remarquons qu'en élevant la fréquence d'exécution de la plateforme (ce qui signifie une réduction de la période de l'horloge), le niveau de la consommation d'énergie garde sa

stabilité. En revanche, la surface d'implémentation est très influencée par le changement de la période. Cette variabilité au niveau des valeurs estimées de la surface est guidée non seulement par l'architecture synthétisée de l'application mais aussi par le type des ressources élémentaires utilisées à chaque opération de synthèse.

Nous remarquons qu'en fixant la condition 'extend_latency=true', nous réduisons la valeur estimée de la consommation d'énergie. Ceci est expliqué par le fait qu'en adoptant cette option, les contraintes de performance sont devenues de moins en moins strictes ce qui permet de réduire la performance du design et par la suite nous obtenons une architecture qui consomme moins. Dans le même contexte, c'est à dire 'extend_latency=true' nous remarquons la réduction de la surface d'implémentation. En effet, les surfaces d'implémentation sont comprises entre $8x10^4$ et $10x10^4$ μm^2 pour 'extend_latency=false'. Par contre, nous obtenons des surfaces d'implémentation estimées comprises entre $6x10^4$ et $8x10^4$ μm^2 pour 'extend_latency=true'.

L'apport de l'option 'extend_latency' est très remarquable dans le processus de synthèse d'architectures matérielles à partir d'une description SystemC comportementale. Cette option permet de réduire simultanément la surface d'implémentation et la consommation d'énergie de l'architecture synthétisée. Par contre, l'effet de l'adoption de la technique d'exécution spéculative n'influence pas réellement les résultats de la synthèse. Ceci est expliqué par l'absence des structures conditionnelles dans la description SystemC du FFT. Dans le paragraphe suivant, nous appliquons ce même flot d'exploration tridimensionnel sur une description SystemC comportementale d'un réseau de neurones de type SOM.

V.5.3.2. Exploration de la synthèse des architectures SOM

Comme il a été mentionné dans la section IV.3.2, nous nous servons des réseaux de neurones afin d'estimer les performances futures et hautement variable d'un composant matériel ou logiciel. L'objectif final de cette recherche est d'explorer les différentes architectures d'une description d'un réseau de neurones avant de l'introduire dans une plateforme reconfigurable [11]. Dans ce contexte, nous explorons l'effet de l'architecture synthétisée à partir d'un niveau haut de description sur les estimations de surface, de performances et de consommation d'énergie.

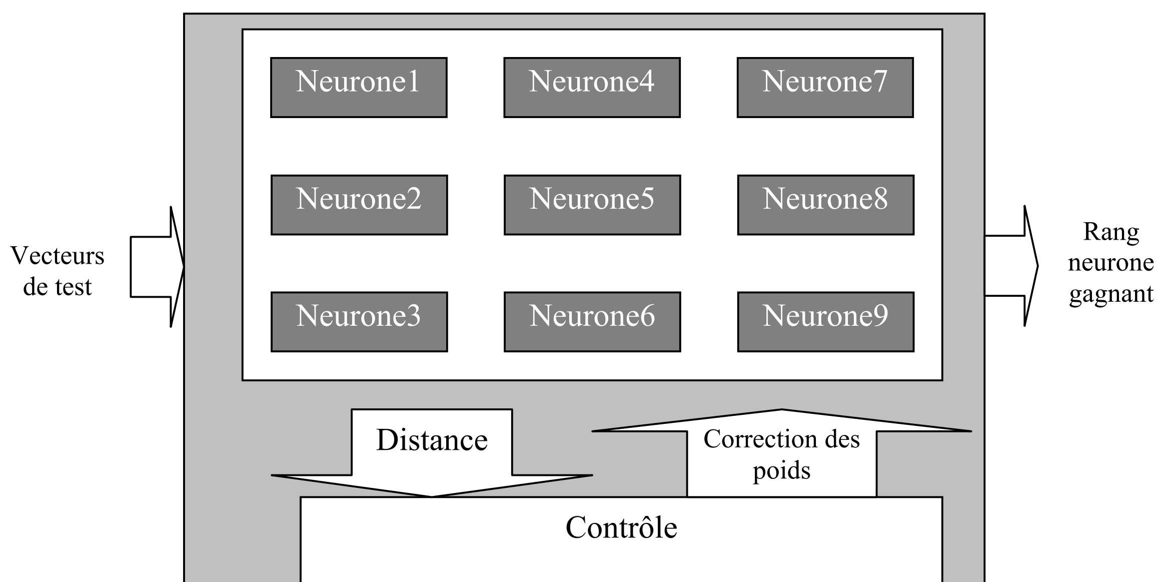


Fig. V. 14: Architecture implémentée effectuant l'apprentissage et le test d'un réseau SOM

Le réseau SOM, exploré par le flot de la figure V.11, est composé de neuf neurones. Dans la figure V.14, nous présentons l'architecture du SOM développée au niveau SystemC comportemental. L'algorithme d'apprentissage du SOM est détaillé dans le paragraphe IV.3.1.2. Cette architecture est composée de neuf unités de traitement élémentaire. Chaque module détermine la distance qui sépare le vecteur poids à celui d'entrée. Ensuite, chaque neurone propage cette distance vers le module contrôle qui détermine le neurone gagnant et démarre le processus de mise à jour des poids.

L'exploration tridimensionnelle de l'effet de l'architecture synthétisée sur la surface, la performance et la consommation d'énergie est illustrée dans la figure V.15.

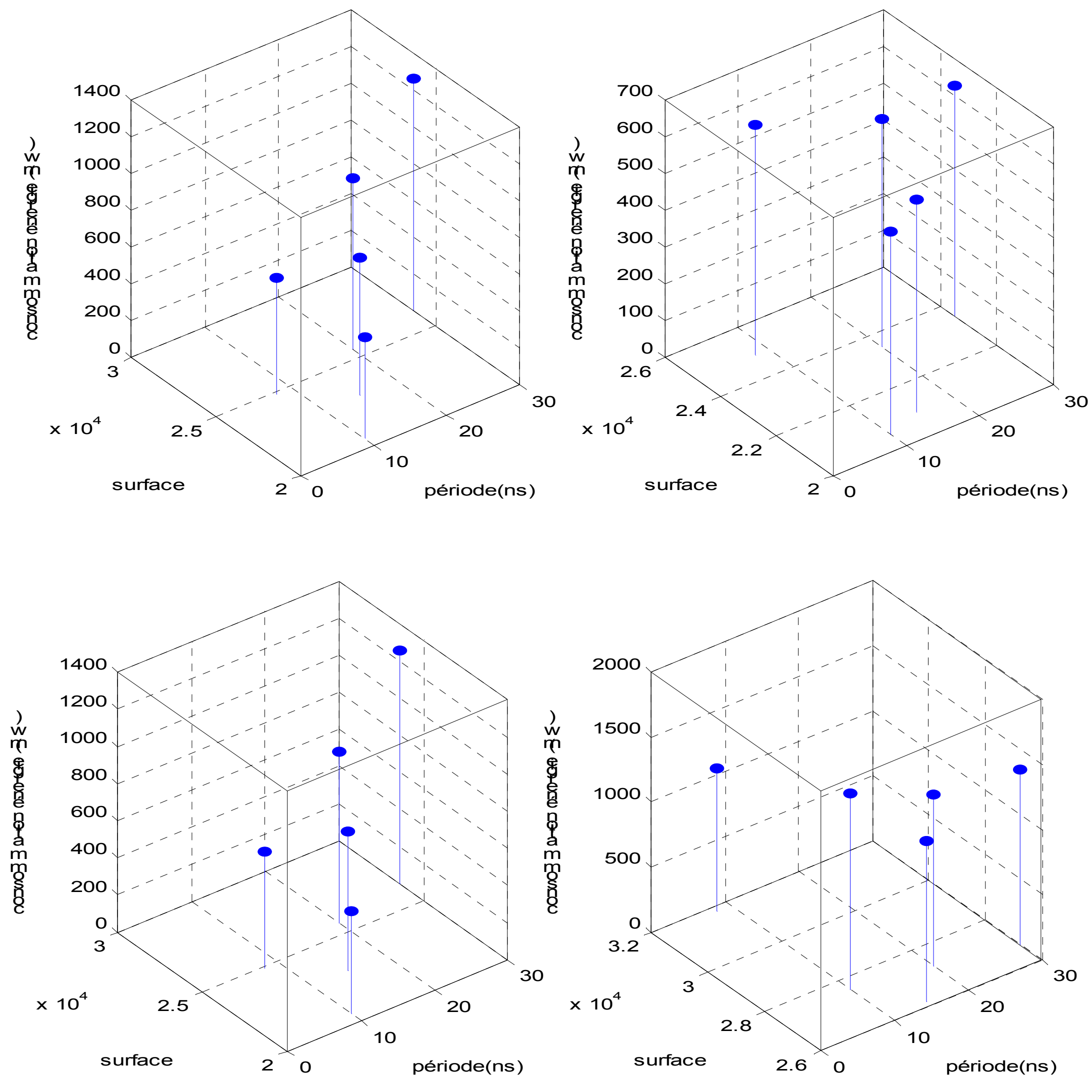


Fig. V. 15 : Exploration à trois dimensions des architectures synthétisées du SOM : (a) cas speculative_execution=false et extend_latency=false, (b) cas speculative_execution=true et extend_latency=false, (c) cas speculative_execution=false et extend_latency=false, (d) cas speculative_execution=false et extend_latency=true

Nous remarquons d'abord qu'un accélérateur matériel implémentant un réseau de neurones de type SOM consomme moins de surface que l'application FFT. Quel que soit les options de synthèse adoptées, l'exploration montre qu'un réseau SOM consomme entre 2x10^4 et 4 x10^4 μm^2. En revanche, les surfaces estimées des architectures synthétisées du FFT sont

comprises entre $6x10^4$et $10x10^4$ μm^2 . Cette différence au niveau de la surface d'implémentation est expliquée par la simplicité des calculs et des traitements d'un neurone par rapport au ceux d'un FFT. Nous remarquons de même que l'effet des options 'extend_latency' et 'speculative_execution' sur la consommation en logique et en consommation d'énergie est plus distingué. L'intervention de l'option 'speculative_execution' dans la synthèse du SOM est plus distinguée par rapport à celle du FFT. En effet, pour une période de synthèse fixée à 7 ns et en fixant 'speculative_execution=false', le flot estime la surface d'implémentation à 2.5 $x10^4$ μm^2 et la consommation d'énergie à $600\,mw$. Par contre, en adoptant les mêmes conditions et en permettant la compilation en exécution spéculative ('speculative_execution =true'), le flot estime la surface à 3.2 $x10^4$ μm^2 et la consommation d'énergie à $1100\,mw$.

Dans cette section nous avons proposé un flot d'exploration capable de parcourir les différentes architectures synthétisées à partir d'un niveau SystemC comportemental. Ce flot est applicable sur toute description SystemC et en particulier sur une description d'un réseau de neurones. Dans le prochain paragraphe, nous proposons un deuxième flot capable d'explorer des architectures d'un réseau de neurones récurrent (nombre de neurones cachés, l'horizon de prédiction, nombre d'entrées) appliqué à la prédiction. Ce flot sert à optimiser les caractéristiques du prédicteur matériel synthétisé.

V.6. Flot d'exploration des paramètres d'un réseau de neurones récurrent

Le choix de l'architecture d'un réseau de neurones récurrent (Fig.IV.18), déployé pour estimer les performances des composants embarqués sur un système sur puce, est un critère déterminant pour sa performance de prédiction et pour son coût d'implémentation. L'architecture d'un système de prédiction neuronal dépend de trois facteurs qui sont :

1. le nombre de neurones de la couche d'entrée qui représente en fait le nombre N (voir IV.1) qui désigne le nombre de coefficients antérieurs pris en compte pour estimer les valeurs postérieures de la série temporelle.
2. le nombre de neurones de la couche cachée qui représente un paramètre déterminant dans le processus d'apprentissage.
3. l'horizon de prédiction (notée n dans l'équation IV.1) détermine la complexité du processus d'apprentissage d'un réseau de neurones récurrent.

Les trois paramètres énumérés ci-dessus n'influencent pas uniquement la performance d'apprentissage et de test des prédicteurs neuronaux mais aussi le coût d'implémentation matérielle des prédicteurs (surface et consommation d'énergie). L'implémentation d'une architecture optimisée d'un prédicteur neuronal nous ramène à résoudre un problème d'optimisation multi objectifs. Dans cette section, nous présentons en premier lieu l'algorithme d'optimisation multi objectifs adopté pour explorer les différentes architectures d'un réseau de neurones récurrent appliqué à la prédiction. En deuxième lieu, nous exposons l'influence des paramètres explorés sur la performance et la consommation d'énergie du prédicteur matériel développé.

V.6.1. Algorithme d'optimisation multi objectifs

Un problème d'optimisation se définit comme la recherche du minimum ou du maximum (de l'optimum donc) d'une fonction donnée [7]. On peut aussi trouver des problèmes d'optimisation pour lesquels les variables de la fonction à optimiser sont contraintes d'évoluer dans une certaine partie de l'espace de recherche. Dans ce cas, on a une forme particulière de

ce qu'on appelle un problème d'optimisation sous contraintes. Ce besoin d'optimisation vient de la nécessité de fournir à l'utilisateur un système qui répond au mieux au cahier des charges. Ce système devra être calibré de manière à :

- Occuper le volume minimum nécessaire à son bon fonctionnement.
- Consommer le minimum d'énergie (coût de fonctionnement).
- Répondre à la demande de l'utilisateur.

V.6.1.1. Notions et définitions

Mathématiquement parlant, un problème d'optimisation est formulé sous la forme suivante :

$$\begin{cases} \min f(\vec{x}) & (f \text{ est la fonction à optimiser}) \\ \text{avec } \vec{g}(\vec{x}) \leq 0 & (\text{m contraintes d'inégalité}) \\ \text{et } \quad \vec{h}(\vec{x}) = 0 & (\text{p contraintes d'égalité}) \end{cases} \qquad (V.1)$$

avec $\vec{x} \in \Re^n$, $\vec{g}(\vec{x}) \in \Re^m$ et $\vec{h}(\vec{x}) \in \Re^p$. Les vecteurs $\vec{g}(\vec{x})$ et $\vec{h}(\vec{x})$ représentent respectivement m contraintes d'inégalité et p contraintes d'égalité. Cet ensemble de contraintes délimite un espace restreint de recherche de la solution optimale.

Dans la littérature, On distingue quelques notions et définitions relatives à l'optimisation multi objectifs.

On définit en premier lieu la fonction objectif (appelée encore fonction de coût ou critère d'optimisation) représentée par f dans l'équation V.1. C'est cette fonction que l'algorithme d'optimisation va devoir optimiser. Dans le cas d'une optimisation multi objectifs, la fonction f sera composée alors de plusieurs composants représentant des objectifs différents dépendants ou non à optimiser. Dans le cas d'une implémentation matérielle d'un réseau de neurones appliqué à la prédiction, la fonction objectif englobe le triplet : l'erreur de prédiction, la surface d'implémentation et la consommation d'énergie).

En deuxième lieu, on distingue les variables de décision regroupées dans le vecteur $\vec{x}$. C'est en faisant varier ce vecteur que l'on recherche un optimum de la fonction f . La variation des variables de décision est souvent soumise aux contraintes d'inégalité et d'égalité. Dans le cas de la recherche d'une implémentation optimisée d'un prédicteur neuronal, nous varions les facteurs énumérés ci-dessus (nombre de neurones d'entrées, nombre de neurones cachées, fenêtre de prédiction).

La troisième notion définie dans le contexte de l'optimisation multi objectif est la notion de dominance. En effet, lorsque nous achevons la résolution du problème d'optimisation multi objectif, nous obtenons une multitude de solutions. Seul un nombre restreint de ces solutions va nous intéresser. Pour qu'une solution soit intéressante, il faut qu'il existe une relation de dominance entre la solution considérée et les autres solutions. On dit alors que $\vec{x}_1$ domine le vecteur $\vec{x}_2$ si :

- $\vec{x}_1$ est au moins aussi bon que $\vec{x}_2$ dans tous les objectifs.
- $\vec{x}_1$ est strictement meilleur que $\vec{x}_2$ dans au moins un objectif.

Les solutions qui dominent les autres mais ne se dominent pas entre elles sont appelées solutions optimales au sens de pareto.

Finalement, on définit des notions reliées à l'organisation des données dans un algorithme d'optimisation multi objectifs. Les notions présentées ici sont essentiellement inspirées des algorithmes génétiques. On démarre dans l'hiérarchie des données par la définition d'un individu qui représente une seule solution explorée. Un individu correspond au codage sous forme de gènes d'une solution potentielle à un problème d'optimisation. Une population regroupe plusieurs individus qui représentent les solutions à explorer. A la fin de l'exploration de la population, les algorithmes évolutionnaires et en particulier les algorithmes multi objectifs génèrent une nouvelle génération d'individus à partir de la population actuelle. Dans la littérature, on distingue plusieurs propositions d'algorithmes d'optimisation multi objectifs. Nous détaillons dans la suite un algorithme particulier qui est le NSGA II (Non dominating Sorting Genetic Algorithm).

V.6.1.2. Algorithme génétique multi objectifs

Les algorithmes génétiques sont très bien adaptés au traitement d'un problème d'optimisation multi objectifs. L'algorithme génétique, représenté par le pseudo code de la figure V.16, implémente plusieurs tâches qui s 'exécutent d'une manière séquentielle et itérative.

```
Initialisation de la population
Evaluation des fonctions objectifs
Calcul de l'efficacité
For i=1 to MaxIter
      Sélection aléatoire proportionnelle à
      l'efficacité
      Croisement
      Mutation
      Evaluation de la fonction objectif
      Calcul de l'efficacité
End For
```

Fig. V. 16 : Algorithme génétique

Chaque individu va être codé à la manière d'un gène. Par exemple, le plus souvent, on fait correspondre une chaîne binaire à l'individu. A chaque individu, on associe une efficacité qui va correspondre à la performance d'un individu dans la résolution d'un problème donné. Après avoir déterminé l'efficacité des individus, on opère une sélection. On copie les individus proportionnellement à leur efficacité. Ensuite, on effectue les croisements entre individus. Plus un individu sera efficace dans la résolution du problème, plus celui-ci se croisera avec un grand nombre d'autres individus. Le croisement est effectué en sélectionnant aléatoirement une position dans le codage de l'individu. A cette position, on sépare le codage puis on échange les morceaux de droite entre les deux individus. Ici, on constate que l'on a obtenu deux nouveaux éléments dans la colonne résultat. Notre population s'est diversifiée. A ce moment, soit on conserve toute la population en supprimant les doublons, soit on supprime les éléments les moins performants, de manière à conserver le même nombre d'individus dans la population. Pour finir avec la génération de la nouvelle population, on peut accéder à une mutation au niveau des gènes d'un individu. Pour cela, on commence par choisir au hasard un certain nombre d'individus. Ensuite, pour chaque individu sélectionné, on choisit au hasard le gène où aura lieu la mutation. La mutation consiste à changer le gène 0 en 1 et réciproquement. On recommence ce procédé jusqu'à ce que l'on atteigne un critère d'arrêt représenté par exemple par un nombre maximal d'itérations.

Les algorithmes génétiques sont très bien adaptés au traitement d'un problème d'optimisation multi objectifs. La seule différence que l'on note entre un algorithme

génétique mono objectif et multi objectifs est au niveau de l'étape d'évaluation. Dans le cas multi objectifs, cette étape se divise en deux : une première étape d'évaluation des différents objectifs et une deuxième étape qui consiste à transformer les vecteurs résultats de l'évaluation en une unique valeur indiquant l'efficacité de la solution. La plupart des algorithmes utilisent une distance qui caractérise la différence entre deux individus. Dans la littérature, nous distinguons plusieurs algorithmes implémentant le fonctionnement d'optimisation multi objectifs. Dans le paragraphe suivant nous détaillons l'algorithme NSGA II.

V.6.1.3. Non dominating Sorting Genetic Algorithm (NSGA II)

Cette méthode est basée sur une classification en plusieurs niveaux des individus. Dans un premier temps, avant de procéder à la sélection, on affecte à chaque individu de la population un rang. Tous les individus non dominés de même rang sont classés dans une catégorie. A cette catégorie, on affecte une efficacité qui est inversement proportionnelle au rang de la catégorie considérée. Pour maintenir la diversité de la population, ces individus classés se voient affectés d'une nouvelle valeur d'efficacité. Pour cela, on utilise la formule suivante :

$$m_i = \sum_{j=1}^{K} Sh(d(i,j)) \qquad \text{(V.2)}$$

Avec

$$Sh(d(i,j)) = \begin{cases} 1 - \left(\dfrac{d(i,j)}{\sigma_{share}} \right)^2 & si\, d(i,j) \prec \sigma_{share} \\ 0 & si\ non \end{cases} \qquad \text{(V.3)}$$

Ici, K désigne le nombre d'individus dans la catégorie considérée et $d(i,j)$ représente la distance entre l'individu i et l'individu j. On calcule cette distance de la manière suivante : chaque différence entre les gènes des individus compte pour +1. σ_{share} est la distance d'influence. Comme on peut le voir dans l'expression ci-dessus, tous les individus qui sont suffisamment proches dont la distance $d(i,j)$ est inférieure à σ_{share} seront pris en compte dans le calcul de m_i. Les autres seront ignorés. La valeur d'efficacité de l'individu i au sein de la catégorie considérée sera alors :

$$f_i = \frac{F}{m_i} \qquad \text{(V.4)}$$

Où F est la valeur d'efficacité affectée à la catégorie à laquelle appartient l'individu. Le pseudo-code de cette méthode est donné dans l'algorithme de la figure V.17. G représente le nombre de classes d'individus.

```
Initialisation de la population
Evaluation des fonctions objectives
Assignation d'un rang basée sur le rang de dominance
Calcul des comptes des voisins
Assignation d'une efficacité partagée
For i=1 to G
      Sélection aléatoire proportionnelle à l'efficacité
      Croisement
      Mutation
      Evaluation de la fonction objectif
      Assignation d'un rang basée sur le rang de
      dominance
      Calcul des comptes des voisins
      Assignation d'une efficacité partagée
End For
```

Fig. V. 17 : Algorithme NSGAII

V.6.2. Flot d'exploration des architectures de réseaux de neurones récurrents

Le flot d'exploration adopté est représenté par le schéma de la figure V.18. Ce flot est basé sur un noyau d'exploration qui implémente l'algorithme NSGAII. Le réseau de neurones récurrent implémenté en C présente une architecture paramétrable au niveau du nombre de neurones d'entrée, de sortie et cachés. Le flot utilise l'outil DALE [8] d'ORINOCO[53] pour estimer la surface et la consommation d'énergie de l'architecture neuronale développée en C.

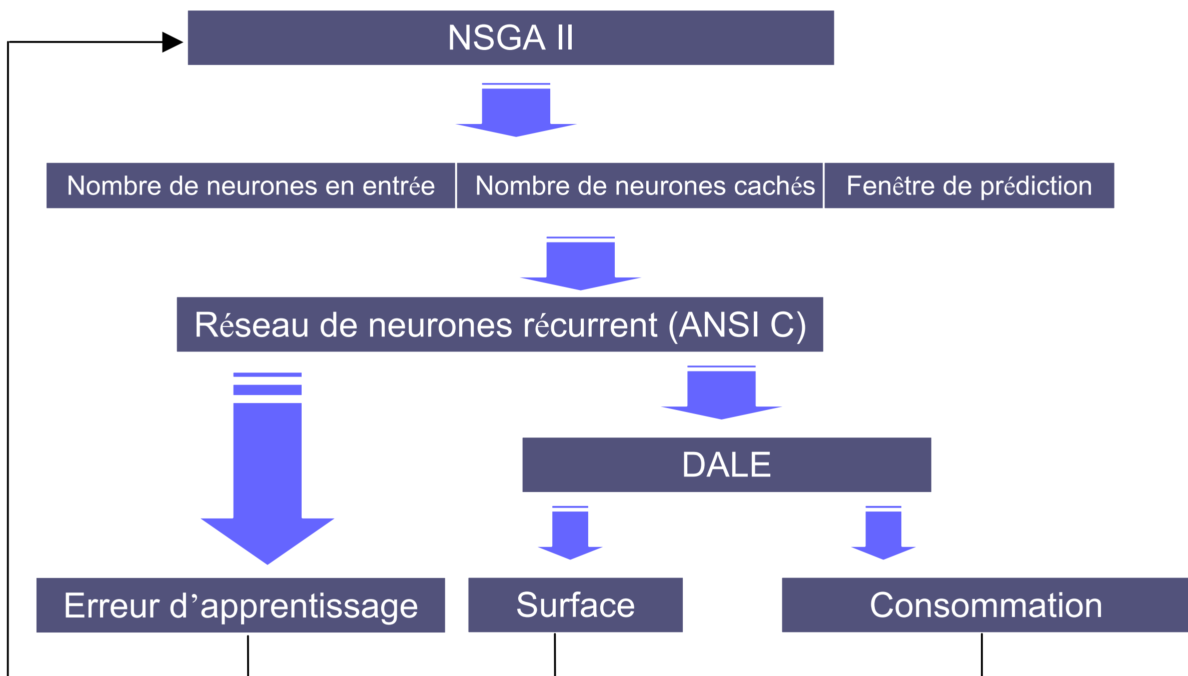


Fig. V. 18 : Flot d'exploration d'architectures de réseaux de neurones de prédiction

Le flot est géré par l'algorithme d'optimisation multi objectifs NSGAII. En premier lieu, le NSGA II fixe la population de départ. Cette population est composée de plusieurs individus qui regroupent les valeurs du nombre de neurones d'entrée, du nombre de neurones cachés et de la fenêtre de prédiction. Nous disposons d'une description en C de l'architecture neuronale récurrente appliquée à la prédiction des séries temporelles. Chaque architecture correspond à un individu unique de la population de départ.

[53] ORINOCO : www.chipvision.com

Pour produire la nouvelle génération d'individus, nous évaluons les fonctions objectives des différentes solutions. Ces objectifs sont représentés par l'erreur d'apprentissage, la surface et la consommation d'énergie de la solution sélectionnée. Pour minimiser le temps d'évaluation des solutions, nous implémentons une version distribuée qui s'exécute en parallèle sur plusieurs calculateurs. A la fin de l'étape d'évaluation, le module du NSGA II reçoit les valeurs des objectifs afin de générer la nouvelle population. Dans le prochain paragraphe, nous présentons les résultats des explorations effectuées.

V.6.3. Résultats d'exploration

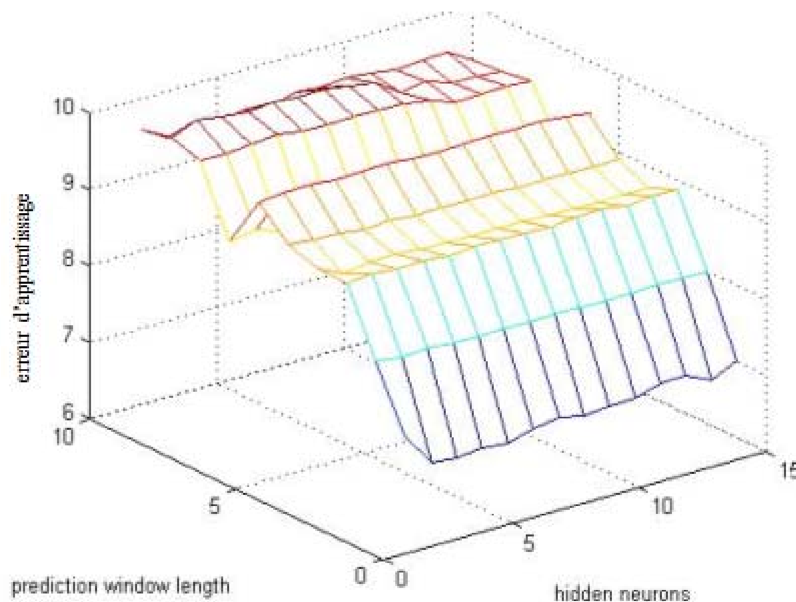
Nous commençons l'évaluation par la présentation de l'effet du choix de l'architecture d'un réseau de neurones sur la performance de prédiction. La performance est représentée par les valeurs de l'erreur de l'apprentissage E. Dans ce cas E est la somme des erreurs de prédiction effectuées sur toute la base d'apprentissage.

$$E = \sum_{i=1}^{l} (y(i) - y^d(i))^2 \qquad \text{(V.5)}$$

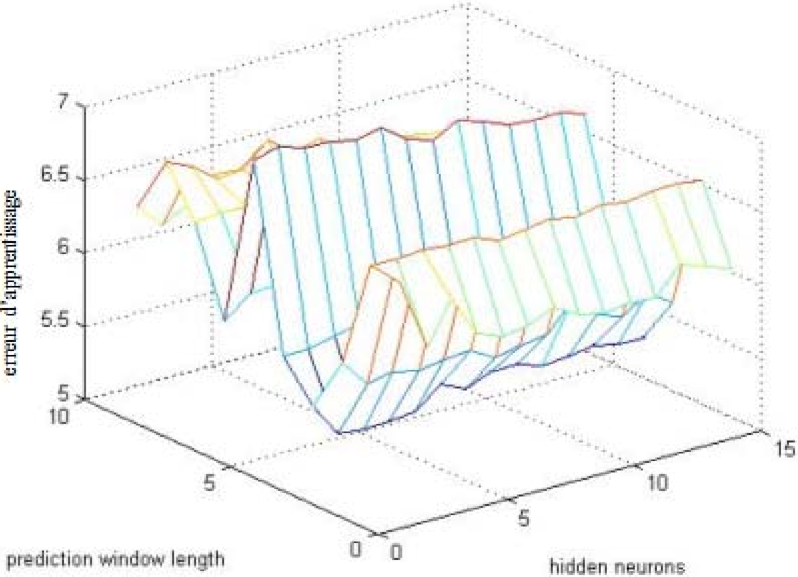
Ici, l est la taille de la base d'apprentissage représentée par les valeurs de la série temporelle y. Dans la deuxième section, nous appliquons un algorithme d'optimisation mono objectif pour optimiser l'apprentissage de l'architecture du réseau de neurones de prédiction. Finalement, dans la troisième section, nous appliquons le flot de la figure V.18 pour explorer et optimiser l'apprentissage et le coût d'implémentation matérielle d'une architecture de réseau de neurones de prédiction.

V.6.3.1. Exploration exhaustive mono objectif

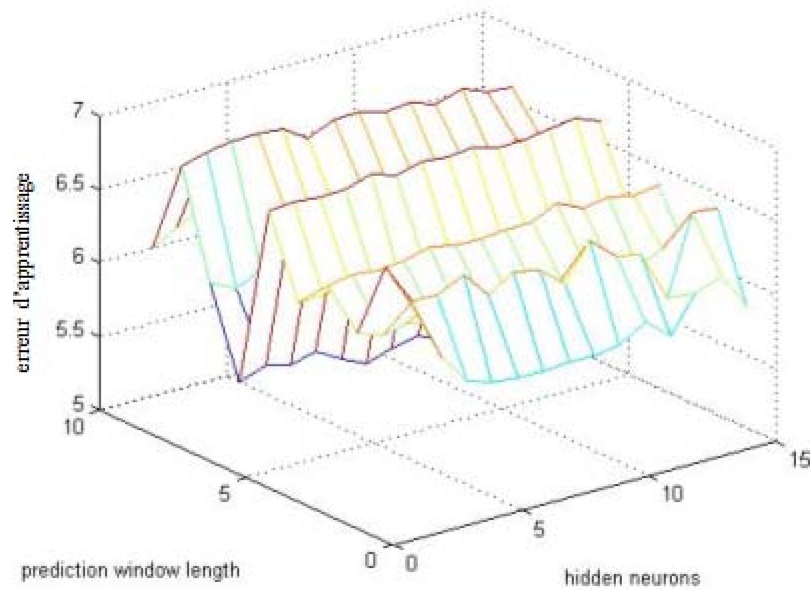
Nous présentons dans cette section les résultats d'exploration exhaustive des architectures neuronaux. L'objectif testé dans cette section sera uniquement l'erreur globale d'apprentissage. Les courbes de la figure V.19 illustre le comportement de l'erreur de l'apprentissage suite à une variation de l'architecture du réseau de neurones récurrent.



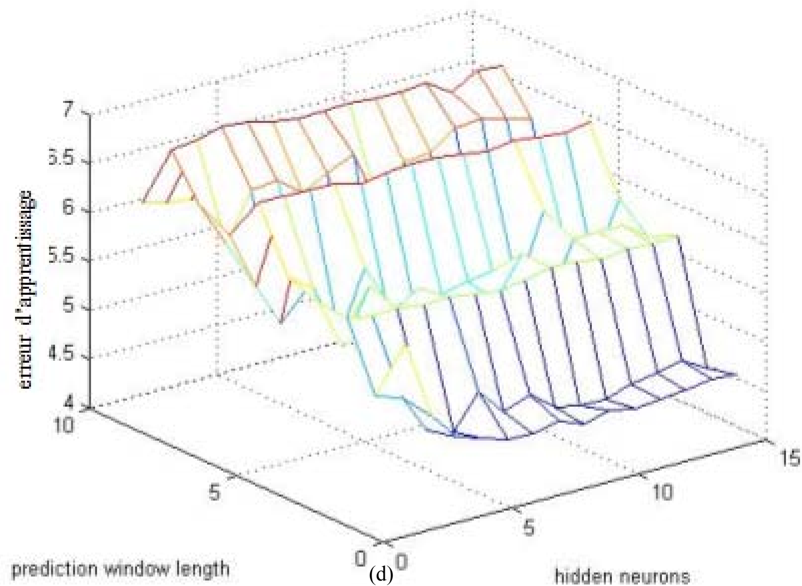
(a)



(b)



(c)



(d)

Fig. V. 19 : Variation de l'erreur d'apprentissage en fonction des paramètres du réseau de neurones de prédiction : (a) prédiction en utilisant un réseau de neurones à deux neurones d'entrée, (b) prédiction en utilisant un réseau de neurones à quatre neurones d'entrée, (c) prédiction en utilisant un réseau de neurones à six neurones d'entrée, (d) prédiction en utilisant un réseau de neurones à onze neurones d'entrée

Cette exploration exhaustive a été effectuée sur une série contenant 777 valeurs. Elle montre l'effet des différents paramètres sur l'apprentissage d'un réseau de neurones récurrent appliqué à la prédiction d'une série temporelle hautement variable. La série de test sélectionnée est celle représentée dans la figure IV.4 (a). Nous remarquons qu'en augmentant le nombre de neurones d'entrée, nous arrivons alors à minimiser l'erreur globale d'apprentissage E. Pour un réseau à deux neurones d'entrée, l'erreur d'apprentissage varie entre 6 et 10. Par contre, en entraînant un réseau de neurones à 11 neurones d'entrée, nous obtenons une erreur d'apprentissage qui varie entre 4 et 6,5. D'une autre part, nous notons que l'effet de l'ajout ou de la suppression de neurones cachés est négligeable sur la phase d'apprentissage.

V.6.3.2. Optimisation mono objectif

Le temps consacré pour le parcours de l'espace des architectures neuronales peut être minimisé en appliquant une technique d'exploration génétique. Le tableau V.4 regroupe un extrait des 20 générations d'architectures neuronales crées en utilisant un algorithme génétique. Chaque individu est représenté dans l'ordre par les trois entiers : nombre de neurones d'entrée, nombre de neurones de sortie et la taille de l'horizon de prédiction.

Table. V. 4 : Architectures et caractéristiques des individus crées dans le cas d'exploration mon objectif

Population	Génération 1	Erreur globale	Génération 3	Erreur globale	………	……….	Génération 20	Erreur globale
1	14, 25, 5	18.024788	17 2 1	8.38343	………	………	15 2 1	3.327906
2	6, 42, 1	5.299604	17 65 1	12.4088	………	……….	15 2 1	3.327906
3	10, 78, 2	7.776421	17 65 1	4.64419	………	………..	15 2 1	3.327906
4	11, 80, 10	16.123573	10 11 1	18.0477	………	……….	15 2 1	3.327906
5	25, 6, 10	21.877352	6 42 1	8.67933	………	……….	15 2 1	3.327906
6	7, 92, 10	14.531813	6 42 1	8.77619	………	……….	15 2 1	3.327906
7	27, 70, 4	14.977207	6 42 1	7.86461	………	……….	15 2 1	3.327906
8	6, 76, 2	8.583758	11 6 3	5.29960	………..	……….	15 2 1	3.327906
9	30, 17, 9	10.168088	11 6 3	7.77642	……….	……….	15 2 1	3.327906
10	17, 33, 5	13.637641	10 5 2	6.62325	……….	……….	15 2 1	3.327906
11	15, 10, 5	9.035599	6 73 1	4.78962	……….	……….	15 2 1	3.327906
12	17, 65, 1	4.632979	10 78 2	4.63297	……….	……….	15 2 1	3.327906
13	6, 93, 3	9.470036	14 83 2	6.44817	……….	……….	15 2 1	3.327906
14	9, 74, 5	15.829240	14 83 2	5.94340	……….	……….	15 2 1	3.327906
15	23, 84, 5	14.575846	10 70 4	7.07669	……….	……….	15 2 1	3.327906
16	11, 6, 3	6.323550	14 82 2	8.49388	……….	……….	15 2 1	3.327906
17	2, 83, 10	22.086594	6 76 2	5.29960	……….	……….	15 2 1	3.327906
18	22, 43, 4	11.123331	6 76 2	8.99238	……….	……….	15 2 1	3.327906
19	22, 86, 10	23.474697	10 67 4	8.67933	……….	……….	15 2 1	3.327906
20	14, 83, 2	7.864616	10 2 4	8.30973	……….	……….	15 2 1	3.327906
21	22, 74, 8	19.215597	15 10 5	4.73543	……….	……….	15 2 1	3.327906
22	23, 55, 3	9.105778	23 55 3	6.07114	……….	……….	15 2 1	3.327906
23	18, 89, 6	20.717373	23 55 3	4.63297	……….	……….	15 2 1	3.327906
24	10, 2, 4	8.992386	23 55 3	5.29960	……..	……..	15 2 1	3.327906
25	19, 33, 5	18.789377	11 79 2	4.99337	……….	……….	15 2 1	3.327906
26	9, 30, 7	15.445537	6 93 3	5.79432	……….	……….	15 2 1	3.327906
27	5, 55, 9	20.133587	6 93 3	6.62325	……….	……….	15 2 1	3.327906
28	21, 55, 8	22.313644	30 17 9	6.32355	……….	……….	15 2 1	3.327906

Nous remarquons que l'algorithme d'optimisation converge vers une architecture optimale dans le sens qu'elle génère une erreur minimale d'apprentissage. Cette architecture correspond à un réseau non récurrent. Les contraintes d'inégalités de cette exploration sont fixées de sorte que le nombre de neurones des couches d'entrée et cachées varie entre 1 et 100 neurones. La taille de la fenêtre de prédiction varie également entre 1 et 30. Nous remarquons aussi que l'algorithme d'optimisation converge rapidement (au bout de 6 générations) vers la solution optimale représentée par le triplet (15, 2, 1). La figure V.20 illustre cette convergence rapide. Cette figure représente l'erreur d'apprentissage en fonction du nombre d'individus dans la population et du numéro de la génération. Nous remarquons l'instabilité de l'erreur d'apprentissage des architectures qui représentent les individus des premières générations. Après la sixième génération, l'erreur globale d'apprentissage se stabilise.

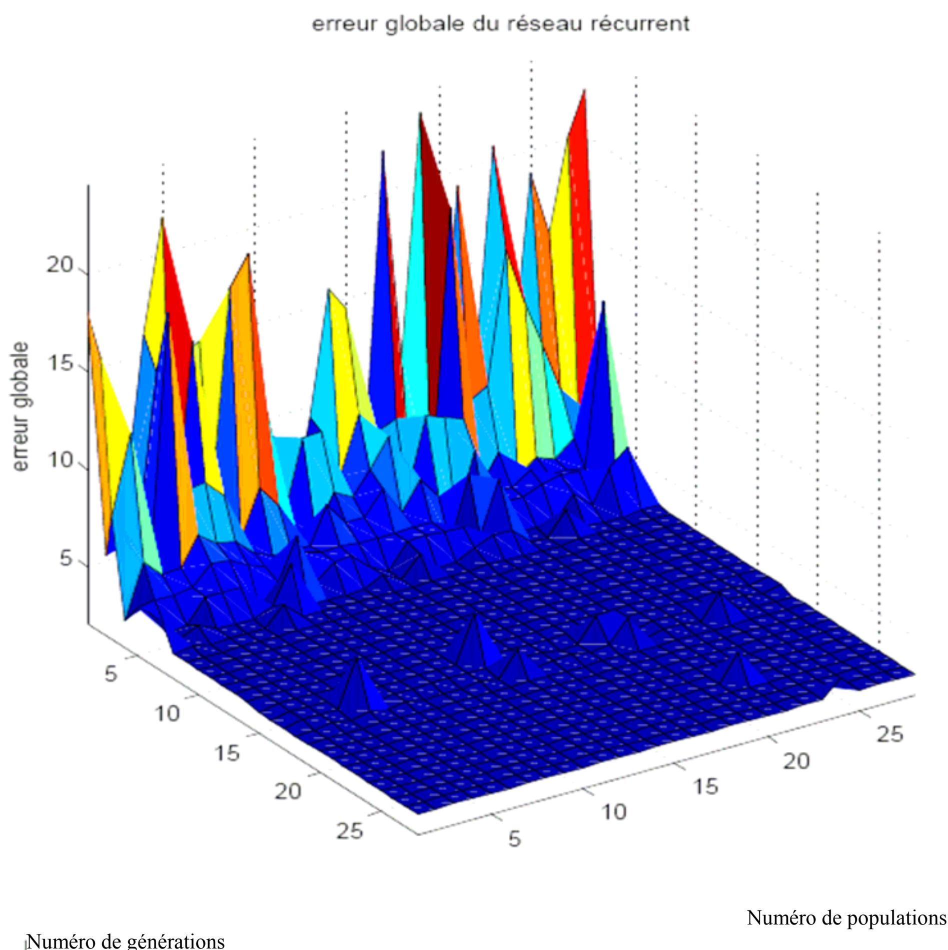


Fig. V. 20 : Exploration des architectures neuronales en optimisant l'erreur d'apprentissage

Cette exploration optimise l'objectif relié à la performance de l'apprentissage d'un réseau de neurones appliqué à la prédiction des séries temporelles. D'autres objectifs interviennent aussi pour caractériser l'efficacité, la performance et l'implémentabilité de l'architecture sélectionnée. Ces objectifs sont essentiellement la surface d'implémentation et la consommation d'énergie de l'architecture neuronale.

V.6.3.3. Optimisation multi objectifs

Dans cette exploration, nous nous intéressons essentiellement à deux objectifs qui sont la performance de prédiction et la consommation d'énergie. Pour estimer l'énergie consommée

par une architecture d'un réseau de neurones, nous utilisons un outil industriel appelé ORINOCO-DALE [8].

a) Présentation d'ORINCO-DALE

ORINOCO-DALE est un outil d'application pour l'estimation et l'optimisation de la consommation d'énergie d'une description système d'une application embarquée. Cette description peut être définie en ANSI C, C++ ou également en SystemC. Cette description est synthétisée sur un ASIC tout en choisissant la technologie d'implémentation (180 nm, 90nm, ..).
ORINOCO-DALE est le premier outil de conception qui fait la synthèse au niveau ESL (Electronic System Level). Les utilisateurs ciblés par cet outil sont les concepteurs des SOCs. Le schéma de la figure V.21 suivant montre les différentes étapes de conception adoptées par ORINOCO-DALE.

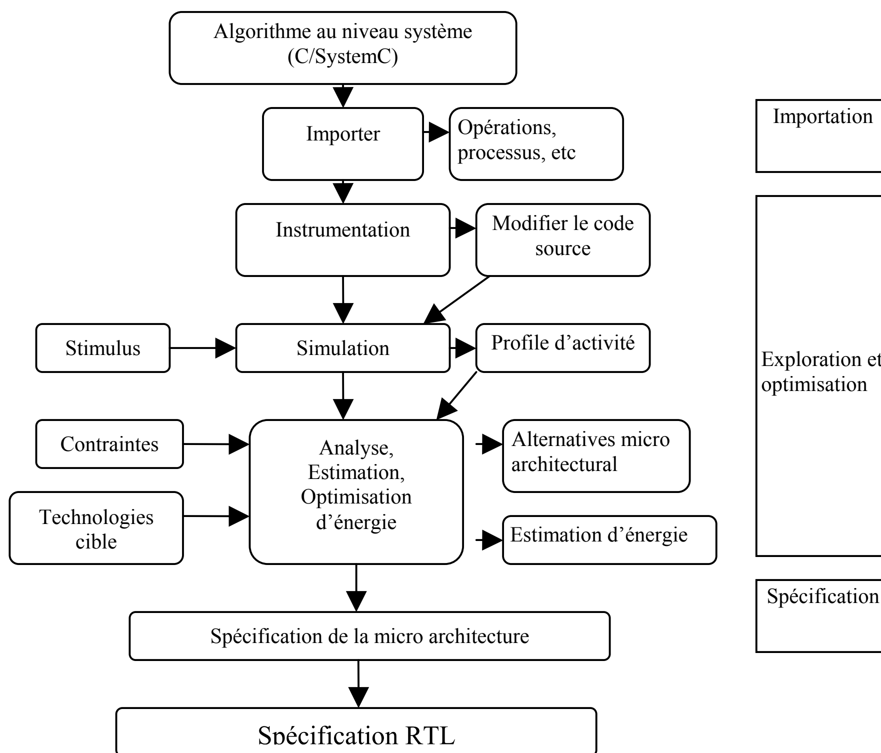
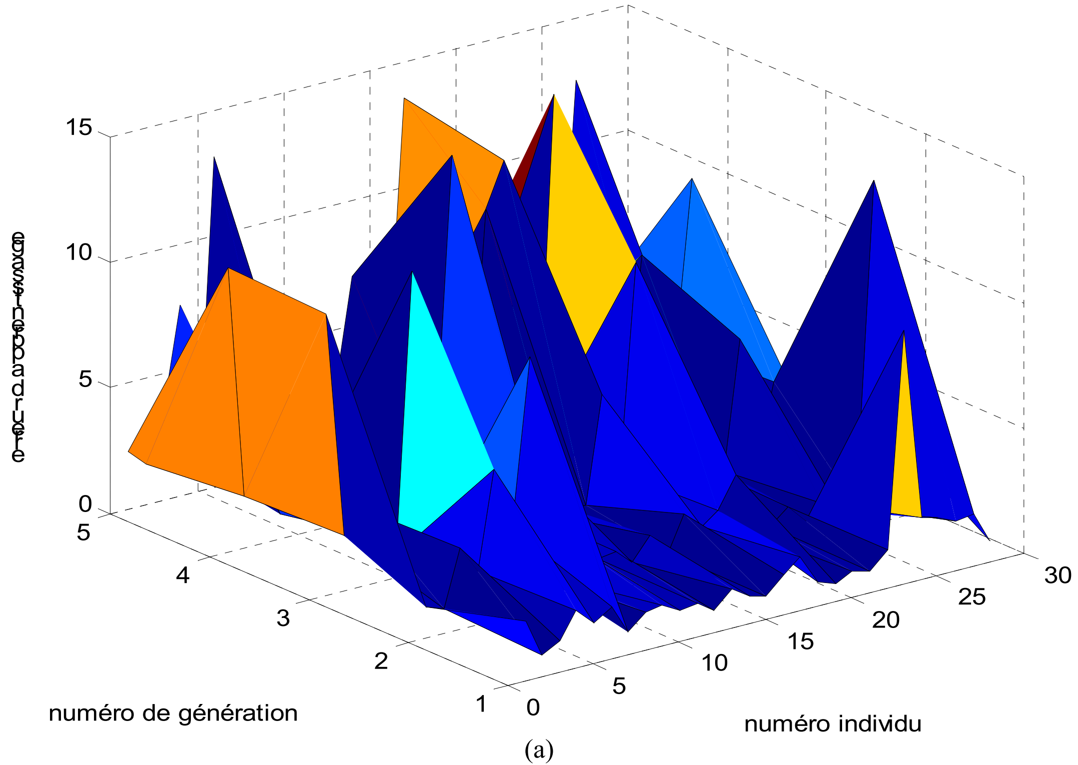


Fig. V. 21 : Flot de conception d'ORINOCO-DALE [8]

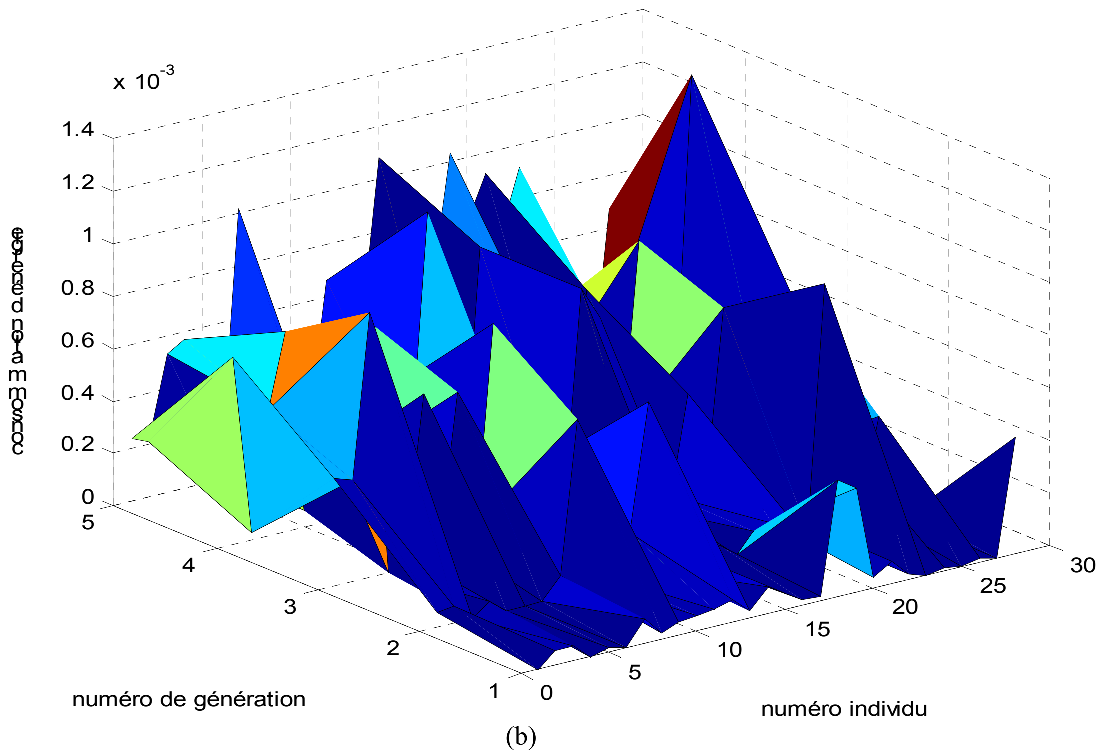
ORINOCO-DALE importe en premier lieu l'application décrite en ANSI C, ou en SystemC. A partir de cette description algorithmique, ORINOCO-DALE génère un flot de contrôle de données. Pour déterminer l'activité du design, ORINOCO-DALE simule l'application en utilisant un banc de test qui stimule l'application par des vecteurs de test représentatifs. L'estimation de la consommation d'énergie à partir d'une spécification système est effectuée suite à une étape de prédiction des composants micro architecturaux alloués pour implémenter l'application sur le matériel. ORINOCO-DALE optimise l'allocation, l'ordonnancement, le placement et le routage des différentes ressources matérielles. Le résultat de cette étape d'optimisation est représenté par un code en ANSI C qui reflète les modifications approuvées durant la phase d'optimisation. ORINOCO-DALE génère également une spécification architecturale définissant la composition de l'architecture synthétisée. Cette architecture est considérée comme une description au niveau RTL de l'application.

b) Résultats d'exploration multi objectifs

Les résultats d'exploration multi objectifs considérés dans cette section illustrent la variation de la performance de prédiction et de la consommation d'énergie de plusieurs réseaux de neurones appliqués à la prédiction.



(a)



(b)

Fig. V. 22 : Evaluation des fonctions objectives des architectures neuronales explorées: (a) évaluation de l'erreur d'apprentissage, (b) évaluation de la consommation d'énergie

La figure V.22 illustre l'évolution des populations gérées par l'algorithme d'optimisation multi objectifs NSGA II. Dans le graphe V.22 (a), nous montrons l'évolution de l'erreur d'apprentissage des différents réseaux de neurones en fonction de la population. Nous remarquons que l'algorithme converge à partir de la cinquième génération vers un minimum global estimé à 2. Cette convergence au niveau de la performance de prédiction coïncide avec une consommation minimale d'énergie illustrée par la figure V.22 (b). Le minimum de consommation est estimé à une valeur équivalente à 13 μw . Le tableau V.5 regroupe les architectures ainsi que les valeurs des fonctions objectives de la première et la dernière population.

Table. V. 5 : Architectures et caractéristiques des individus crées dans le cas d'une exploration multi objectifs

INDIVIDU	GEN1	ERR APPRENTISSAGE	CONSO μw	GEN5	ERR APPRENTISSAGE	CONSO μw
2	6, 42, 1	1.641	214	27, 65 1	0.88	54
3	10, 78, 2	3.237	533	24, 6, 1	1.27	56
4	11, 80, 10	7.623	569	6, 16, 9	2.29	0
5	25, 6, 10	1.114	82	4, 4, 1	1.63	20
7	27, 70, 4	1.633	1019	10, 2, 4	0.96	95
8	6, 76, 2	2.380	389	6, 16, 9	1.57	26
9	30, 17, 9	1.342	277	23, 4, 1	1.669	51
10	17, 33, 5	1.780	334	27, 28, 3	1.277	56
11	15, 10, 5	1.207	97	17, 2, 4	1.362	64
12	17, 65, 1	0.962	664	5, 2, 9	0.903	86
13	6, 93, 3	2.874	476	24, 4, 3	1.575	26
14	9, 74, 5	6.975	477	9, 2, 2	1.143	105
15	23, 84, 5	3.385	1083	25, 7, 10	0.962	62
16	11, 6, 3	1.669	51	17, 10, 1	1.632	20

Nous remarquons que l'algorithme ne converge pas forcément vers la même architecture. En effet, contrairement à l'optimisation mono objective, les solutions de la cinquième génération présentent des architectures différentes. En revanche, nous obtenons des prédicteurs performants qui consomment de moins en moins d'énergie.

V.7. Conclusion

Dans ce chapitre, nous avons focalisé sur les méthodologies d'exploration de l'espace d'architectures synthétisées à partir d'un niveau haut de description des architectures matérielles. Cette exploration est une étape particulièrement importante dans le processus d'optimisation de l'architecture à implémenter. Les applications visées dans ce contexte sont les réseaux de neurones appliqués à la prédiction. Nous proposons alors deux flots d'exploration des architectures neuronales. Le premier flot explore l'influence des options supportées par le compilateur haut niveau sur l'opération de synthèse architecturale. Le deuxième flot explore l'influence de l'architecture d'un réseau de neurones (composition en neurones) sur sa performance de prédiction et sur son coût d'implémentation.

Dans une première partie, nous avons proposé un premier flot afin d'explorer l'effet de l'architecture sur le coût d'implémentation du système. Le choix de l'architecture est fixé à l'aide des options de synthèse proposées par le compilateur haut niveau. Ces options, qui dirigent l'opération de synthèse architecturale, forment une espace d'exploration limitée. Nous avons opté alors pour l'implémentation d'une technique exhaustive d'exploration. Ce flot automatique regroupe plusieurs outils de CAO qui coopèrent afin d'estimer la surface, la performance et la consommation d'énergie des différentes architectures. L'exploration exhaustive des différentes architectures a montré l'effet du choix des options de synthèse sur les coûts d'implémentation. Les applications synthétisées et explorées par ce flot sont le FFT et un réseau de neurones de type SOM.

La deuxième partie a porté sur les méthodologies d'optimisation de l'architecture d'un réseau de neurones récurrent afin d'implémenter, à un coût minimum, un système de prédiction performant. Le choix de l'architecture d'un réseau de neurones récurrent est effectué en fixant le nombre de neurones d'entrée, le nombre de neurones cachés et la taille de la fenêtre de prédiction. Ces paramètres forment un espace vaste de prédiction. Nous avons opté alors pour l'implémentation d'une technique d'exploration dirigée par un algorithme d'optimisation multi objectifs (NSGA II). L'évaluation des différentes architectures est effectuée d'une manière distribuée. Cette étape d'évaluation consiste à estimer deux aspects différents qui sont la performance de prédiction et la consommation d'énergie de l'architecture sélectionnée. La performance de prédiction est estimée à la suite d'une étape d'apprentissage du réseau. La consommation d'énergie est estimée à partir d'une description ANSI C de l'architecture neuronale en utilisant un outil de CAO appelé ORINOCO-DALE.

V.8. Références

[1] SystemC : www.systemc.org

[2] CoCentric SystemC Compiler Behavioral User and Modeling Guide, version U-2003.06, June 2003.

[3] FPGA Compiler User Guide Version 2001.08-FC3.6.1, Synopsys, August 2001.

[4] Physical Compiler User Guide, version 2001.08, Synopsys, August2001.

[5] Power Compiler User Guide, version U-2003.06, June 2003.

[6] Modelsim : http://www.model.com/

[7] Collette. Y, Siarry. P, "Optimisation multiobjectif", Edition Eyrolles, ISBN : 2212111681.

[8] ORINOCO-DALE: Chip vision design systems, "ORINOCO-DALE 2006.1.1 User Guide", 2006.

[9] S.Chtourou and O.Hammami, 'Space Exploration of Behavioral Synthesis Options on Area and Performance', Third IEEE International Conference on Systems, Signals & Devices SSD'05, Tunisia.

[10] S.Chtourou and O.Hammami, 'Space Exploration of Behavioral Synthesis Options on Area, Performance and Power consumption', IEEE International Conference on Microelectronics, ICM'05, Décembre 13-15, 2005, Pakistan

[11]S.Chtourou and O.Hammami, 'Design Space Exploration of SystemC SOM Implementation', 3rd IEEE International Conference on Information & communications technology ICICT'05, Décembre 05-06, 2005, Egypt.

VI. Prédiction neuronale pour la reconfiguration dynamique de systèmes sur puce

VI.1. Introduction

Dans le chapitre précédent, nous avons proposé deux méthodologies d'exploration et d'optimisation des architectures de réseaux de neurones appliqués à la prédiction de séries temporelles. La première méthodologie explore l'effet des options fixées par l'utilisateur pendant la phase de synthèse de l'architecture à implémenter à partir d'une description SystemC. La deuxième méthodologie explore l'effet de l'architecture d'un réseau de neurones (nombre de neurones cachés, neurones d'entrées et taille de la fenêtre de prédiction) sur les coûts d'implémentation (consommation d'énergie, surface) du prédicteur matériel. L'optimisation de l'architecture du réseau de neurones récurrent aide à générer un prédicteur matériel performant et qui consomme un minimum de portes logiques et d'énergie. Cette phase d'optimisation sert à alléger les coûts d'implémentation du prédicteur ce qui implique la décharge partielle de la plateforme pour implémenter l'architecture reconfigurable souhaitée. Dans ce chapitre, nous allons présenter deux cas d'implémentation d'un système sur puce reconfigurable au niveau transactionnel.

D'après le schéma de la figure IV.2, nous représentons la décomposition d'un système sur puces en trois niveaux de granularité qui sont : le niveau d'applications, niveau des modules et le niveau d'instructions. Dans la première partie de ce chapitre, nous allons explorer les différentes possibilités de prédiction au niveau des instructions. La caractéristique dominante d'une base de métriques extraite à ce niveau est la grande quantité de coefficients qui représentent l'évolution de l'application. Le but de cette première partie est d'étudier la robustesse de la technique de prédiction neuronale appliquée sur des séries temporelles représentant la trace d'allocation des instructions d'une application logicielle. Le comportement testé dans cette partie est celui des adresses instructions d'un code ANSI C.

Dans la deuxième partie de ce chapitre, nous allons présenter un deuxième cas d'implémentation de systèmes sur puce reconfigurables au niveau des applications. Le cas d'étude pris en compte est le codeur entropique de la chaîne de compression d'images fixes JPEG2000 [1]. En premier lieu, nous allons présenter brièvement le standard JPEG2000. Nous allons focaliser particulièrement sur la phase de codage entropique du standard. Ensuite, nous allons présenter la description au niveau SystemC comportemental du codeur implémenté. Finalement, nous allons détailler l'architecture de la plateforme multi-codeurs, parallèle et reconfigurable implémentée à base d'un prédicteur neuronal associé à plusieurs codeurs entropiques. Les résultats de reconfiguration sont également exposés et commentés dans cette partie.

VI.2. Environnement d'implémentation : PEK-1.0

L'outil PEK-1.0 d'IBM a été introduit dans le paragraphe IV.3.1.3 de cette thèse. Cet outil représente l'environnement de test haut niveau adopté pour l'implémentation de systèmes sur puce réactifs au niveau transactionnel. L'environnement principal proposé par PEK-1.0 est le CbSLD[54]. Il sert à la construction, simulation et la vérification haut niveau de l'architecture d'un système embarqué. La description de ce système est effectuée en utilisant une bibliothèque d'interfaces développées en SystemC au niveau TLM. Ces interfaces représentent les composants de base développés pour connecter des cœurs de processeur et des périphériques IBM. Ces architectures n'englobent pas seulement les composants existant d'IBM mais aussi des accélérateurs matériels développés par le concepteur. La bibliothèque d'interfaces d'IBM comporte donc des composants capables de simplifier la création d'architectures au niveau système. En utilisant cette bibliothèque, les concepteurs de systèmes sur puce seront capables de regrouper des processeurs PowerPC, des bus de communication comme le PLB et des périphériques. Le modèle est simulé en utilisant la bibliothèque standard OSCI SystemC. La conception au niveau système permet de :

1. Simuler des applications logicielles communiquant avec des accélérateurs matérielles et des périphériques. Le concepteur peut vérifier aussi le fonctionnement des applications sous des contraintes de traitement temps réel.
2. Vérifier l'interconnexion et la communication à travers des bus. La modélisation à ce niveau permet d'approcher la communication entre les composants au niveau cycles.
3. Evaluer les transactions à travers les bus de communication.
4. Evaluer le degré de concurrence entre les différents modules ce qui permet d'implémenter une architecture potentiellement parallèle.

Les principaux composants de cette bibliothèque sont :

1. Une description PowerPC 405 ou 440 modélisée en utilisant un simulateur d'instructions (ISS Instruction-Set Simulator). Le ISS est implémenté à l'intérieur d'un bloc (wrapper) qui assure le lien entre le processeur et le bus de communication PLB. Le ISS est synchronisé avec le PLB même s'ils sont connectés à deux horloges différentes. Le ISS est exécuté en parallèle avec un débogueur appelé 'RiscWatch'. Le 'RiscWatch' permet de déboguer un code en ANSI C. Il permet également de visualiser le contenu des registres et de la mémoire cache. Pour générer le code exécuté avec ISS, l'utilisateur doit compiler un code ANSI C en utilisant le compilateur GNU Cross-Compiler. Les zones d'adressage de la mémoire locale et de la mémoire cache sont configurées à travers le ISS.
2. Le PLB est un bus de communication qui sert à interconnecter des composants à fréquence d'exécution rapide comme les cœurs de processeurs. L'accès au PLB est géré par un mécanisme d'arbitrage. Le PLB supporte des opérations de lecture et d'écriture entre un composant maître et un autre esclave. La communication dans ce cas est composée de trois étapes. Le composant maître commence par émettre une requête au PLB. Cette requête est gardée par l'arbitre du bus. Finalement, le PLB envoie la requête vers le composant esclave.

[54]CBSLD : ChipBench System Level Design

3. Un contrôleur de mémoire composé d'un ensemble de registres DCR[55]. Comme dans le cas d'une implémentation réelle, le délai d'extraction d'une donnée à partir de la mémoire est fixé dans un registre spécifique du contrôleur.

La reconfiguration dynamique, comme vue dans le chapitre trois, touche à plusieurs niveaux de granularité de l'architecture. Dans ce chapitre, nous analysons les possibilités de reconfiguration à deux niveaux. Nous étudions la possibilité de reconfiguration au niveau des instructions ainsi qu'au niveau des applications.

VI.3. Architecture neuronale appliquée à la prédiction de traces d'instructions

Nous focalisons dans cette section sur la prédiction des adresses d'instruction en utilisant un réseau de neurones récurrent. Nous rappelons que l'architecture du prédicteur neuronal a été présentée dans la figure IV.18. Les traces des adresses d'instructions été représentées dans les figures IV.11 et IV.13. Les caractéristiques statistiques de ces traces sont regroupées dans les tableaux IV.3 et IV.4. Ces traces sont composées par plusieurs coefficients (quelques millions d'adresses) appartenant à plusieurs localités. En effet, l'exécution d'un programme comportant quelques dizaines d'instructions génère une trace contenant des millions d'adresses d'instructions élémentaires. En outre, l'allocation des adresses mémoires des instructions est effectuée d'une manière répétitive dans des zones bien précises de la mémoire. Ce phénomène engendre un effet de localité qui caractérise ces traces. Finalement, nous notons la grande variance des différentes traces testées. Dans cette section, nous présentons les résultats de prédiction d'une série d'adresses d'instructions en utilisant le modèle de prédiction NARX déjà présenté dans le paragraphe IV.4.2.2.

VI.3.1. Prédiction neuronale des adresses d'instructions

La prédiction neuronale est effectuée en exécutant deux phases qui sont l'apprentissage et le test. Pendant la phase d'apprentissage, le réseau propage les vecteurs de la base et met à jour ses poids de connexion. En revanche, la phase de test consiste à utiliser le réseau de neurones déjà conditionné pour générer les prochains vecteurs de la base de test. La génération des vecteurs prédits est effectuée suite à l'excitation de l'entrée du réseau par un vecteur de la base de test. L'étape de test détermine la capacité de généralisation du réseau de neurones. Cette propriété permet au réseau de neurones de prédire des vecteurs même s'ils n'appartiennent pas à la base d'apprentissage. Le diagramme d'état de la figure VI.1 représente l'algorithme d'apprentissage et de test du modèle neuronal. Dans ce diagramme nous utilisons trois variables :

1. L'indice k indique l'ordre du composant dans la base d'apprentissage représentée par la série temporelle y . k est initialisé à 0. $k \in [0, l]$ durant la phase d'apprentissage ou de test avec l est la taille de la série.
2. L'indice n représente l'horizon de prédiction. Dans la phase de test nous fournissons le vecteur $(y(k-N+1),..., y(k))$. Le réseau prédit alors le vecteur $(\hat{y}(k+1), \hat{y}(k+2),..., \hat{y}(k+n))$.
3. L'indice 'itération' indique l'ordre de prédiction intermédiaire. Initialement, 'itération' est égale à 0.

Ce diagramme expose les principaux états d'exécution du programme d'apprentissage et de test du réseau de neurones récurrents appliqué à la prédiction. Pour prédire un coefficient

[55] DCR : Device Control Register

d'ordre $(k+n)$, l'utilisateur commence par exciter la couche d'entrée du réseau par le vecteur d'ordre k composé par les coefficients $(y(k-N+1),..., y(k))$. Pour tester le réseau, il faut charger les poids du réseau à tester puis propager le vecteur d'entrée vers l'avant. Le réseau estime, en premier lieu, la sortie $\hat{y}(k+1)$ qui correspond à la valeur prédite du coefficient $y(k+1)$. On utilise cette valeur pour construire le vecteur d'ordre k+1 composé par les coefficients $(y(k-N+2),..., \hat{y}(k+1))$. Ce vecteur est ensuite propagé dans le réseau de neurones. L'algorithme d'apprentissage répète cette opération n fois (n représente la largeur de la fenêtre de prédiction). On prédit à la fin de cette phase la valeur du coefficient $\hat{y}(k+n)$. La phase d'apprentissage du réseau est effectuée selon l'algorithme de rétro propagation de l'erreur à travers le temps exposé dans la section IV.4.2.2.

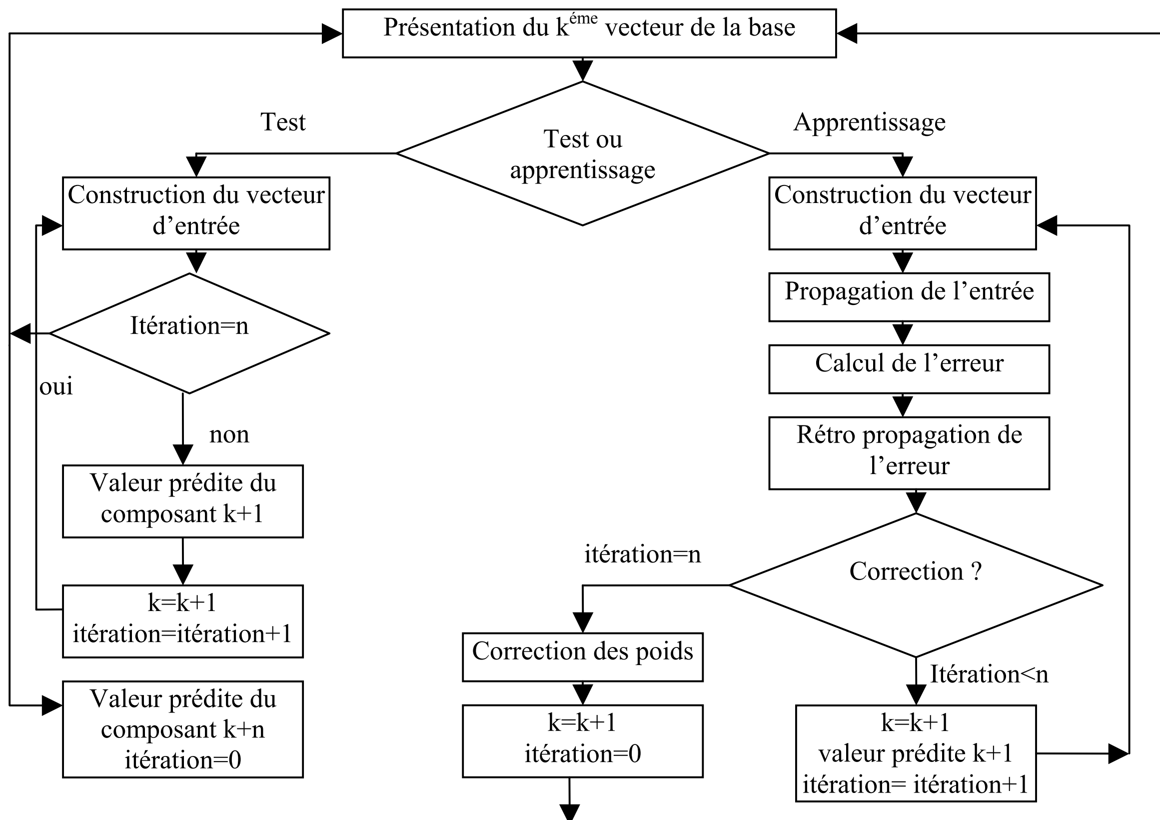


Fig. VI. 1 : Diagramme d'état de l'algorithme d'apprentissage et de test du système de prédiction neuronale implémenté

Les résultats du tableau VI.1 exposent les erreurs quadratiques moyennes (MSE) d'apprentissage en adaptant des bases d'apprentissage normalisées et de différentes tailles. Le MSE est calculé à partir des sorties normalisées selon l'équation (VI.1) :

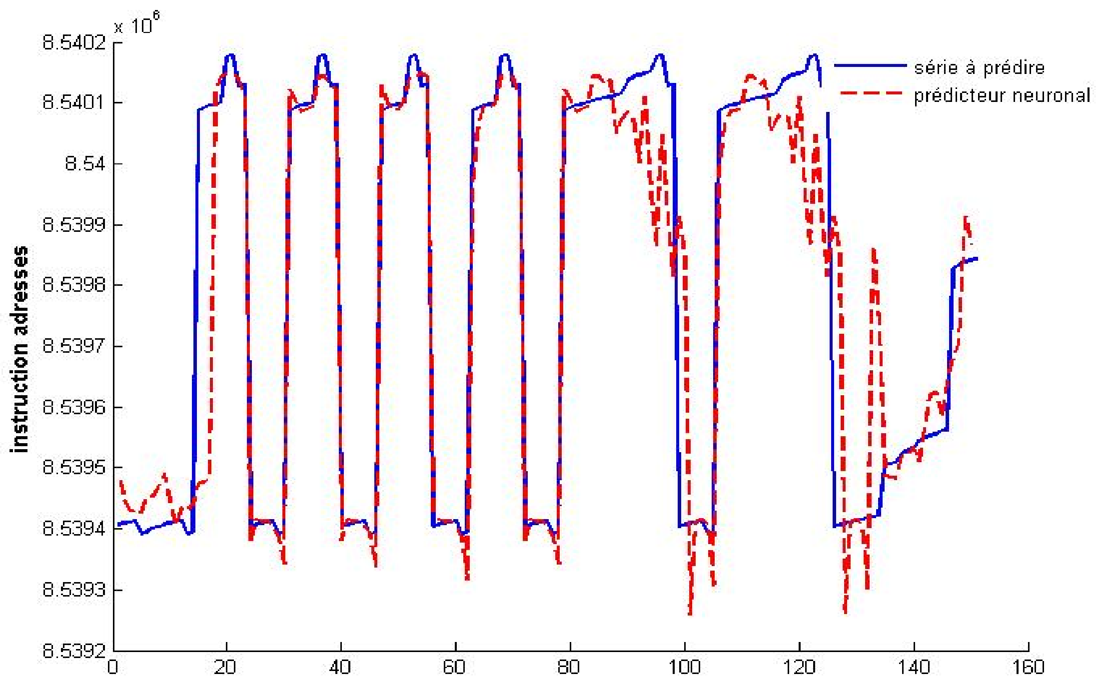
$$MSE = \sqrt{\sum_{i=1}^{l} \left[y(i) - \hat{y}(i) \right]^2} \qquad \text{(VI.1)}$$

Table. VI. 1 : Performances de prédiction en utilisant des bases de test de différentes tailles

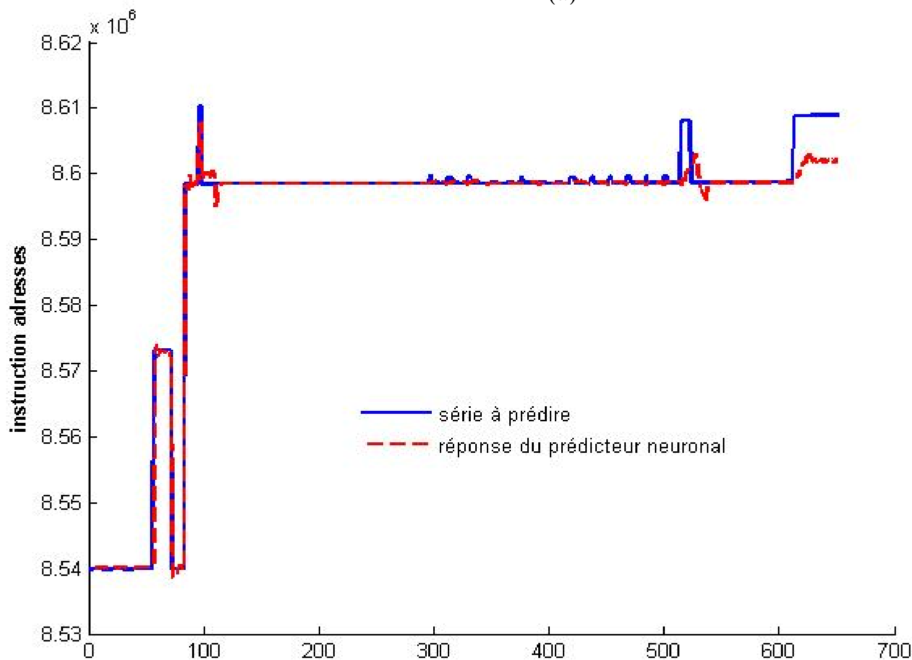
TAILLE DE LA BASE	500	5000	10000
Trace d'apprentissage (LS)	0,0075	0.0939	0.1483
Trace de test (CP)	0.0099	0.0956	0.1536
Trace de test (GZIP)	0,0075	0.0929	0.1550

Les réseaux de neurones récurrents utilisés dans cette expérimentation sont des structures à trois couches. La fixation de l'architecture d'un réseau de neurones se base sur les expérimentations effectuées dans le paragraphe V.6.3.1. Dans cette investigation, la couche d'entrée comporte 15 neurones, la couche de sortie comporte un neurone, la couche cachée est constituée de 5 neurones. Finalement, La fenêtre de prédiction est fixée à 3 pas.

La figure VI.2 expose les résultats d'apprentissage et de généralisation d'un réseau de neurones récurrent appliqué à la prédiction d'une série d'adresses d'instructions de la trace LS.



(a)



(b)

Fig. VI. 2 : Apprentissage et test de généralisation d'un réseau de neurones récurrents appliqué à la prédiction: (a) résultat d'apprentissage, (b) résultat de généralisation

La figure VI.2 (a) représente une partie de la série utilisée dans la phase d'apprentissage du réseau de neurones de prédiction. Cette partie comporte 150 coefficients. La capacité de généralisation du réseau de neurones est testée sur une partie de la série qui ne figure pas dans la base d'apprentissage. Le test de la capacité de généralisation du prédicteur neuronal est effectué sur une partie de la série comportant 650 coefficients.

Nous remarquons que les coefficients résultant de l'étape de test du prédicteur ne suivent pas exactement le même comportement de la série à prédire. Les caractéristiques statistiques de la série à prédire, telles que la variance et la taille, déterminent la performance d'apprentissage et de généralisation du prédicteur neuronal. L'accroissement des erreurs d'apprentissage et de généralisation en fonction des caractéristiques de la base d'apprentissage nous a incités à proposer un nouveau schéma de prédiction neuronale. Afin d'améliorer la capacité de prédiction des séries temporelles, nous proposons dans la section V.3.2, une nouvelle architecture de prédiction basée sur un modèle neuronal hybride.

VI.3.2. Modèle neuronal hybride appliqué à la prédiction

Le travail proposé dans cette section entre dans le cadre d'un axe de recherche visant l'optimisation des architectures des réseaux de neurones. Les travaux proposés dans la littérature focalisent sur l'introduction d'une certaine flexibilité au niveau de la couche d'entrée [2] et cachée [3], [4] d'un réseau de neurones.

Le schéma de prédiction proposé par la figure VI.3 est constitué d'un nombre de prédicteurs neuronaux. Ce nombre est déterminé pendant la phase d'apprentissage. Le nombre de prédicteurs implémentés dépend de la variabilité des coefficients de la série.

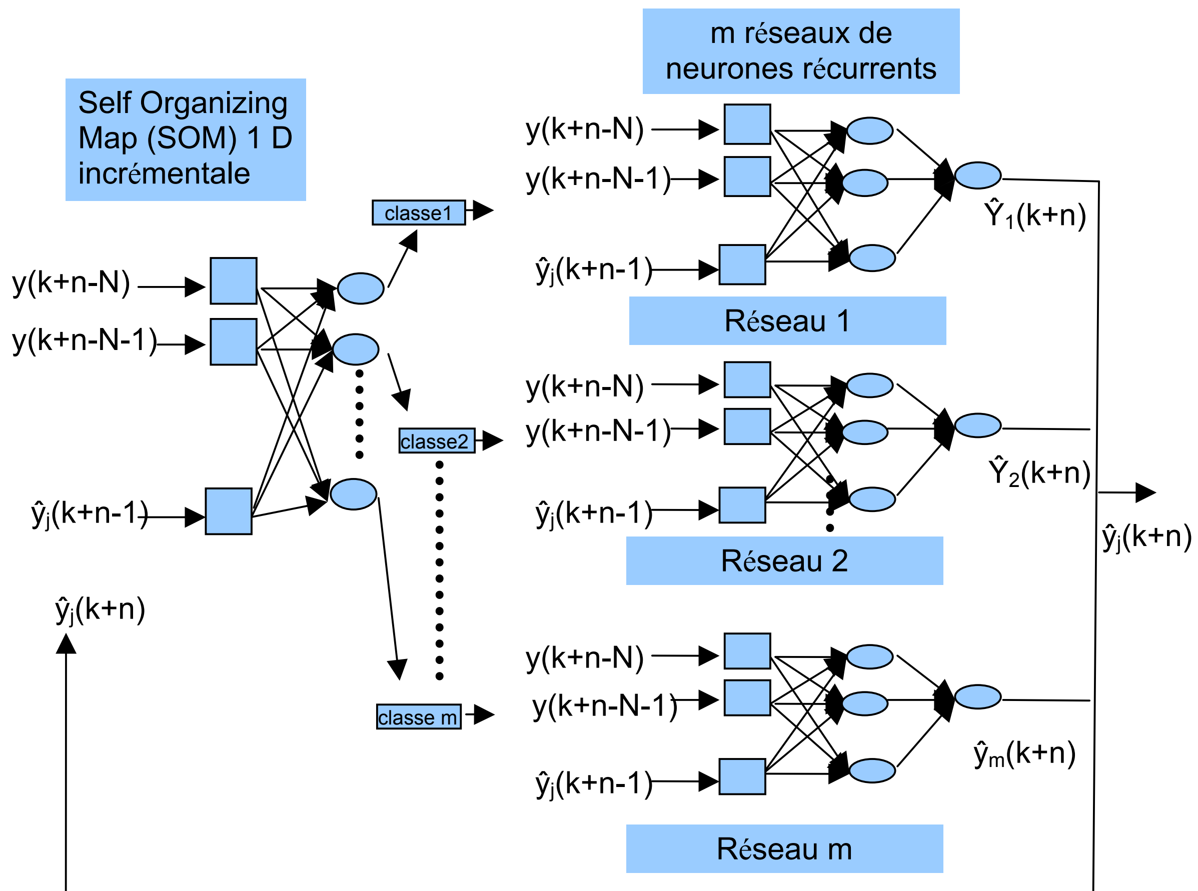


Fig. VI. 3 : Modèle neuronal hybride appliqué à la prédiction [7]

Le modèle proposé se compose de deux parties effectuant deux types de traitement différents. La première partie effectue la classification des coefficients de la série temporelle à apprendre. Cette phase est réalisée par un réseau de neurones de type SOM monodimensionnelle. La deuxième partie est une étape de prédiction effectuée par un ensemble de réseaux de neurones récurrent activés par les sorties du classifieur.

Ce schéma de prédiction se base sur l'hypothèse affirmant que la coopération de plusieurs systèmes de prédiction optimise la qualité de prédiction des coefficients d'une série temporelle [5]. L'idée de base qui caractérise ce schéma de prédiction consiste à associer chaque localité de la série à un prédicteur neuronal local. Ceci implique que ce modèle est typiquement utilisé pour prédire des séries comportant des localités multiples et variables (cas des séries d'adresses d'instructions). Le modèle neuronal hybride appliqué à la prédiction implémente un caractère adaptatif et évolutif durant la phase d'apprentissage. Les mécanismes d'adaptation sont implémentés dans deux niveaux différents :

- Au niveau de la couche cachée : l'algorithme d'apprentissage adapte le nombre de neurones d'une façon incrémentale [4]. Dans ce cas, le réseau de neurones démarre la phase d'apprentissage en utilisant un seul neurone caché. L'algorithme d'apprentissage ajoute un neurone caché si l'erreur reste supérieure à un seuil fixé à l'avance après un certain nombre d'itération.
- Au niveau du nombre de prédicteur : le nombre de prédicteurs utilisés pour construire le modèle hybride est généré automatiquement en ligne au cours de l'apprentissage. Ce qui implique que le nombre de prédicteurs dépend du nombre de classes existant dans la série utilisée pendant la phase d'apprentissage.

VI.3.2.1. Classification incrémentale

La première phase de traitement illustrée dans la figure VI.3 est une étape de classification. Le mécanisme de prédiction démarre par la mise en place d'un réseau SOM à N entrées (N représente le nombre de coefficients d'entrée prises en compte par le réseau SOM) et un seul neurone de sortie. Le neurone de sortie représente une localité de données unique. Nous attribuons à ce neurone un prédicteur unique. Le mécanisme d'incrémentation des neurones du SOM est appliqué sur la couche de sortie. Dès que la distance entre un vecteur d'entrée et les vecteurs poids de tous les neurones de la couche de sortie est supérieure à un seuil fixé τ par l'utilisateur, le SOM ajoute alors un neurone à sa couche de sortie. Le neurone ajouté représente une nouvelle classe de la série. Les poids du nouveau neurone de sortie sont initialisés alors par les coefficients du vecteur d'entrée. L'algorithme du SOM incrémental est implémenté comme suit:

1. Initialiser le vecteur poids w_1 du premier neurone par le premier vecteur d'apprentissage.
2. Initialisation : K = N et $ordre_classe = 1$
3. Calculer la distance euclidienne entre le vecteur présenté $\xi_k = (y(k-N+1),..., y(k))$ et le vecteur des poids de chaque neurone.
4. Si la distance euclidienne est supérieure à τ alors on incrémente par 1 le nombre de neurones de sortie :

   $$ordre_classe = ordre_classe + 1$$

   $$w_{ordre_classe} = \xi_k$$
5. Sélection du neurone gagnant : le neurone i ayant la plus petite distance

6. Les vecteurs poids du voisinage du neurone i de la carte de Kohonen sont alors mis à jour selon :

$$w_o = w_o + \eta \cdot h(o,i) \cdot (\xi - w_o)$$

h(o,i) est une fonction qui retourne 1 si le neurone o est inclus dans le voisinage du neurone i, et 0 si non.

7. Rependre l'apprentissage à partir de l'étape 3 et incrémentation de k.

Dans l'algorithme présenté ci-dessus, nous développons une méthode pour adapter le nombre de prédicteurs utilisés à la complexité de la série. Le choix de la valeur du seuil τ détermine le nombre de classes ($ordre_classe$) détecté à la fin de la phase d'apprentissage.

VI.3.2.2. Stratégie de prédiction neuronale hybride

Dans cette section, nous focalisons sur le fonctionnement du prédicteur neuronal hybride. La figure VI.3 expose l'organisation du système de prédiction complet. Le système complet possède N entrées et une seule sortie. A un instant précis t, le système déploie en mode apprentissage ou test un réseau de neurones récurrent unique sélectionné par le SOM. Le diagramme d'état de la figure V.4, analyse le fonctionnement de l'ensemble des composants du modèle hybride de prédiction proposé.

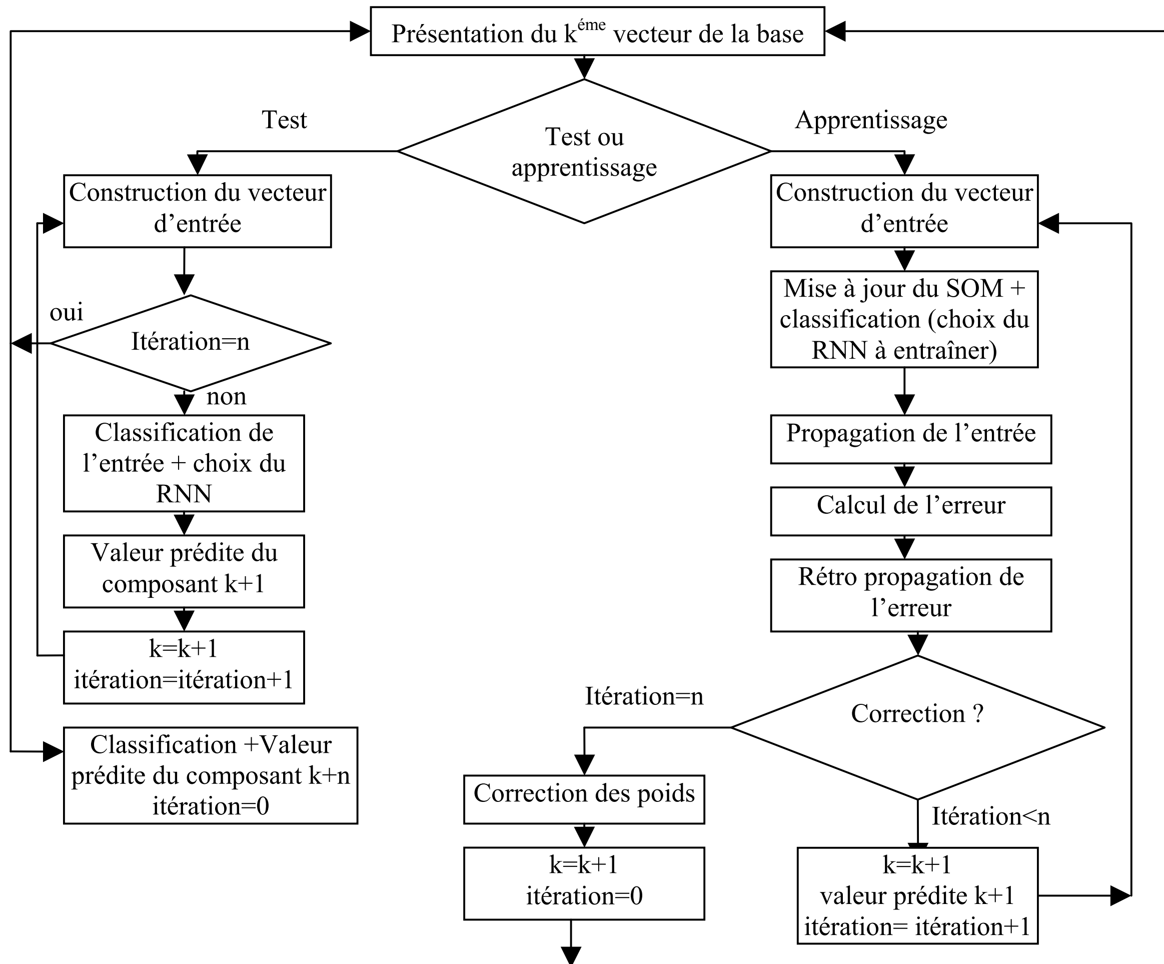


Fig. VI. 4 : Diagramme d'état du modèle neuronal hybride appliqué à la prédiction

La différence entre ce diagramme et celui proposé par la figure V.1 est l'insertion d'une nouvelle étape de classification et de sélection d'un unique réseau de neurones. Cette étape est essentielle dans le cas de test ou d'apprentissage du système de prédiction. Dans le schéma de prédiction développé, nous supposons que l'apprentissage du SOM est déjà effectué avant la phase d'apprentissage ou de test des réseaux de neurones récurrent. L'implémentation de cette méthode est basée sur l'approche objet qui a simplifié la manipulation des différentes entités du système.

VI.3.2.3. Résultats d'implémentation

Afin de tester les performances du prédicteur, nous l'avons développé pour prédire les adresses d'instructions de trois programmes différents. Dans le tableau Table.VI.2, nous présentons des résultats de prédiction en adoptant une base d'apprentissage (trace du programme LS) et de généralisation (trace des programmes CP et GZIP) comportant 100000 adresses d'instruction. Les résultats présentés par le MSE (Mean Square Error) sont calculés à partir des coefficients normalisés de la base d'apprentissage ou de test. Les réseaux de neurones récurrents adoptés sont tous identiques. Ils contiennent 15 neurones d'entrées, 10 neurones cachés, la fenêtre de prédiction est fixée à 5.

Table. VI. 2 : Résultats de prédiction en utilisant le modèle neuronal hybride

NOMBRE DE	MSE APPRENTISSAGE	MSE GENERALISATION	MSE GENERALISATION

CLASSES	(LS)	(CP)	(GZIP)
1	2.8662	10.7405	24.1532
3	2.8690	10.7196	21.4572
6	2.8691	9.2355	24.7295
10	3.1025	50.6963	15.9570

Nous montrons aussi par ce tableau, l'effet de la phase de classification (nombre de classes) sur les résultats de prédiction. Les résultats de généralisation dépendent des caractéristiques de la base à prédire. Les résultats au niveau de l'erreur de l'apprentissage ne montrent pas l'effet réel du modèle hybride. Nous illustrons cet effet dans la figure VI.5.

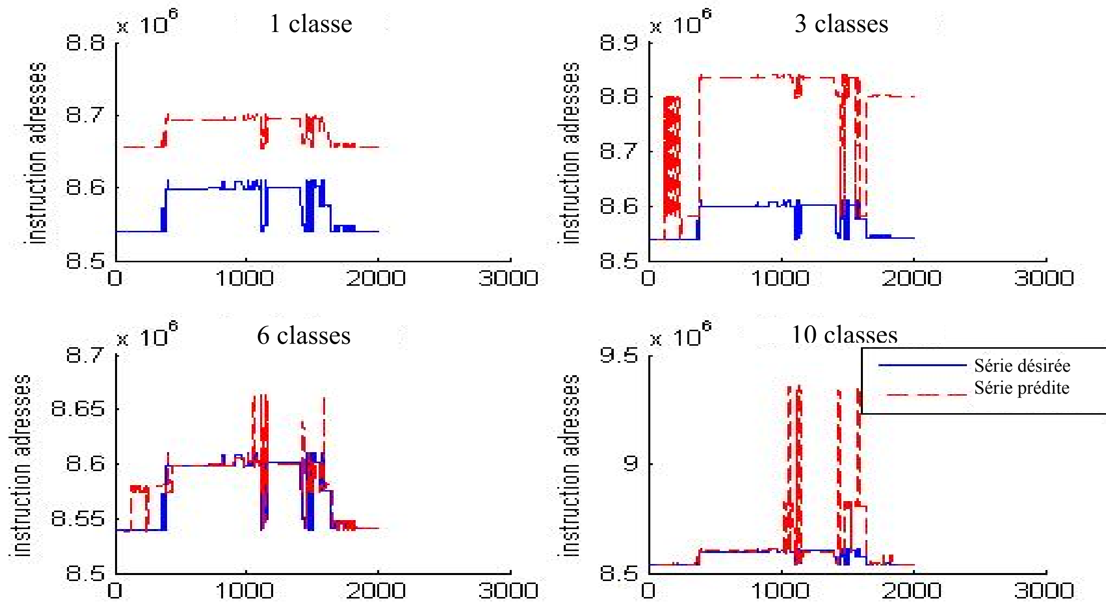


Fig. VI. 5 : Résultats d'apprentissage de la série LS en adoptant un SOM à une, trois, 6 et 10 sorties (classes)

Nous remarquons que la prédiction est localement améliorée. En augmentant le nombre de classes, la performance de prédiction au niveau de chaque réseau est meilleure. L'inconvénient dans cette approche est illustré essentiellement dans la figure qui traite le cas qui implémente 10 prédicteurs. Cette figure illustre des erreurs de prédiction importantes dans les zones transitoires entre les localités. Ces erreurs participent à l'augmentation du MSE global de prédiction. Afin de démontrer l'effet positif de cette approche sur la prédiction de la série, nous proposons de tester cette approche en augmentant le nombre de classes allouées et la taille des séries d'apprentissage et de généralisation. Le contrôle du nombre de classes est effectué en variant la valeur du seuil τ .

Le tableau VI.3 regroupe les erreurs normalisées et non normalisées d'apprentissage (sur 100000 coefficients) et de généralisation (sur 1 million de coefficients) du modèle hybride. Nous adoptons successivement 16, 32, 64 et 128 classes.

Table. VI. 3 : Résultats d'apprentissage et de généralisation de coefficients non normalisés d'une base d'apprentissage de grande taille [8]

Nombre de classes	MSE apprentissage LS (100000 coefficients)	NMSE apprentissage (100000 coefficients)	MSE généralisation LS (1M coefficients)	NMSE généralisation LS (1M coefficients)
16	1.1083e+009	2.0250e+005	1.9128e+010	1.0065e+005
32	1.0696 e+009	6.1225e+004	1.9176e+010	1.0254e+005
64	1.0472e+009	5.9303e+004	1.4992e+010	1.0956e+005

128	9.4274 e+008	5.7594e+004	1.8666e+010	1.1150e+005

Dans ce cas, nous remarquons une diminution considérable des erreurs d'apprentissage. Les erreurs d'apprentissage et de généralisation sont mesurées par le MSE et le NMSE (Normalized Mean Square Error). Le NMSE étant calculé à l'aide de l'équation VI.2 :

$$NMSE = \frac{\sum_{1 \le i \le l} \left[(y(i) - \hat{y}(i)^2)^{1/2} \right]}{\sum_{1 \le i \le l} \left[(y(i) - \overline{y})^2)^{1/2} \right]} \qquad \text{(VI.2)}$$

$\overline{y}$ étant la valeur moyenne de la série.

En conclusion, nous avons montré dans cette section l'effet positif du modèle hybride dans la prédiction des séries temporelles de grande taille. En adoptant cette approche nous optimisons localement la performance de prédiction. En revanche, nous remarquons que ce modèle présente des défaillances en traitant les zones de transition entre les différentes localités de la série.

VI.4. Système reconfigurable au niveau des applications

Dans cette section, nous allons présenter un cas d'étude traitant une stratégie de reconfiguration dynamique au niveau système. Le mécanisme de reconfiguration proposé agit sur des applications indépendantes de la plateforme afin de libérer une partie de la surface logique pour implémenter d'autres traitements. Le cas d'étude considéré est le codeur entropique de la chaîne de compression d'images fixes JPEG2000 [1]. Dans la première partie de cette section, nous présentons brièvement le standard JPEG2000. On s'intéresse particulièrement à la phase de codage entropique du standard. Dans la deuxième partie, nous présentons l'architecture SystemC du codeur à implémenter. Dans cette phase, nous analysons l'architecture parallèle implémentée du codeur entropique. Finalement, nous exposons la plateforme reconfigurable implémentée à base d'un réseau de neurones et plusieurs codeurs entropiques parallèles. Nous exposons également les résultats de reconfiguration sous la forme de mesures indiquant la composition de la plateforme au cours de l'exécution.

VI.4.1. Chaîne de compression d'images JPEG2000

JPEG2000 est un nouveau standard ISO pour la compression des images fixes (ISO/IEC 15444-1 et ITU-T T.800) [6]. JPEG2000 est la seule norme de compression d'images à couvrant l'ensemble des besoins de compression d'images avec ou sans pertes.

En plus de sa capacité de fournir une meilleure performance en termes de taux de compression à qualité équivalente, JPEG2000 permet d'accéder à différentes représentations des images (résolution spatiale, région d'intérêt…), et permet de s'adapter aux conditions d'utilisation (capacité de traitement du terminal réalisant le décodage, capacité du canal de transmission…). L'ensemble des données nécessaires est contenu dans un fichier compressé unique. Cela est rendu possible grâce aux modes de codage de progressivité en résolution et en qualité ainsi qu'avec le concept de région d'intérêt. Un seul module de compression peut fournir plusieurs représentations de façon simultanée. Une seule fonction de compression peut fournir les deux représentations (pleine résolution, résolution intermédiaire, qualité intermédiaire) de façon simultanée, en utilisant un mode particulier du codage, appelé 'progressivité'. Cela permet de coder dans le même flux de données les deux représentations. La plus simple en terme de résolution est utilisée directement pour être émise et sert aussi

d'information de base pour, en complément de l'information d'enrichissement, former les images de pleine résolution.

Les caractéristiques principales de la chaîne de compression JPEG2000 représentée par la figure VI.6 sont les suivantes:

- Progressivité en qualité ou en résolution.
- Compression avec ou sans perte dans le même algorithme.
- Accès aléatoire aux données.
- Décompression partielle des données.

VI.4.1.1. Transformée en ondelettes

Le schéma de compression JPEG2000 est présenté par la figure VI.6 qui expose tous les blocs de traitement de JPEG2000.

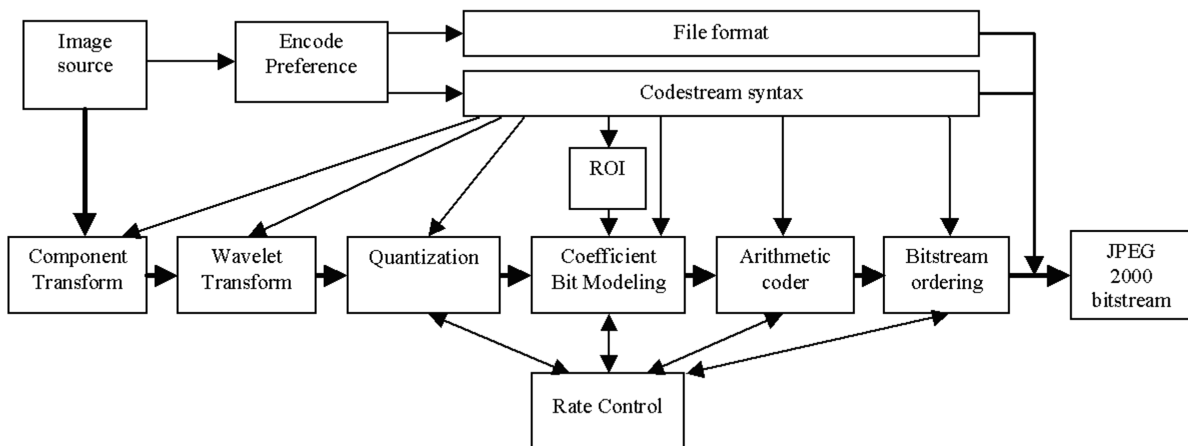


Fig. VI. 6 : Chaîne de compression JPEG2000

L'opération de compression d'une image fixe commence par la transformée en couleurs ou 'component transform'. Cette transformation s'obtient en faisant une combinaison linéaire des composantes RVB (Rouge, Vert, Bleu) de l'image. On obtient alors une nouvelle représentation de l'image appelée 'représentation chromatique' de type (Y, Cb, Cr). La deuxième étape du standard est la transformée en ondelettes utilisée pour l'analyse de l'image pour différents niveaux de décomposition ou de résolution. Les niveaux de décomposition contiennent un nombre de sous-bandes, composée chacune de coefficients qui décrivent les caractéristiques horizontales et verticales de l'image originale.

Le standard JPEG2000 traite seulement la décomposition en puissance de 2. On passe d'un niveau N-1 à N par un filtrage passe-bas et passe-haut sous échantillonnés d'un facteur 2 sur les lignes puis sur les colonnes. La figure VI.7 montre le mécanisme de décomposition de la transformée en ondelettes.

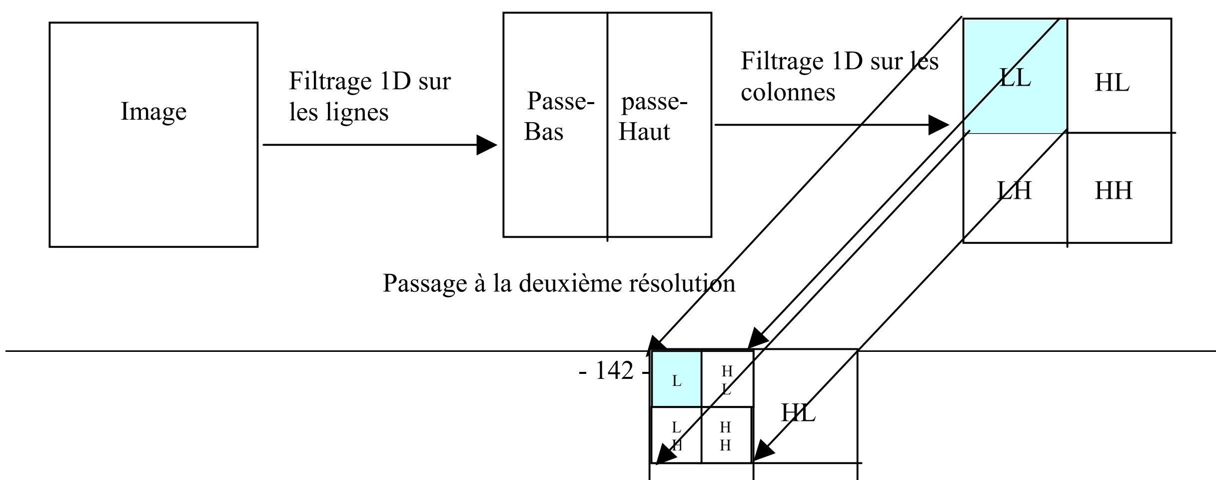


Fig. VI. 7 : Effet de la transformée en ondelettes sur une image

Les zones LL, LH, HL et HH représentent respectivement les coefficients qui ont subit séquentiellement un filtrage passe-bas/ passe-bas, passe-bas/ passe-haut, passe-haut / passe-bas et passe-haut/ passe-haut. A la suite de cette étape de transformée, les coefficients les plus significatifs de la transformée sont regroupés dans la zone LL de l'image transformée.

La troisième étape de la chaîne JPEG2000 est la quantification qui permet de réduire la dynamique des données. Cette étape est destructive. L'effet de la quantification des coefficients dépend du pas de quantification adopté. L'étape de quantification est suivie par une étape de sélection des régions d'intérêt (ROI[56]). Le codeur JPEG2000 permet de coder certaines régions de l'image avec plus de qualité.

VI.4.1.2. Codeur entropique

La dernière étape est le codage entropique qui regroupe plusieurs sous-ensembles de traitement :

- Découpage en blocs appelés code-blocs.
- Conversion de la représentation des valeurs des coefficients en valeur absolue/signée.
- Modélisation des données binaires (modélisation du contexte).
- Codage arithmétique.

Le découpage en blocs est l'étape qui suit la quantification. Les coefficients issus des différentes sous-bandes sont rangés en blocs, appelés code-blocs de forme rectangulaire et de taille paramétrable. Chaque code-bloc est codé indépendamment des autres. Cette indépendance offre la possibilité de définir des architectures parallèles effectuant le codage entropique.

Les code-blocs sont en fait des tableaux de coefficients qui peuvent être représentés par un tableau tridimensionnel binaire constitué de plusieurs bit-plans. La figure VI.8 montre la disposition des bit-plans constituant un code-bloc.

[56] ROI : Region Of Interest

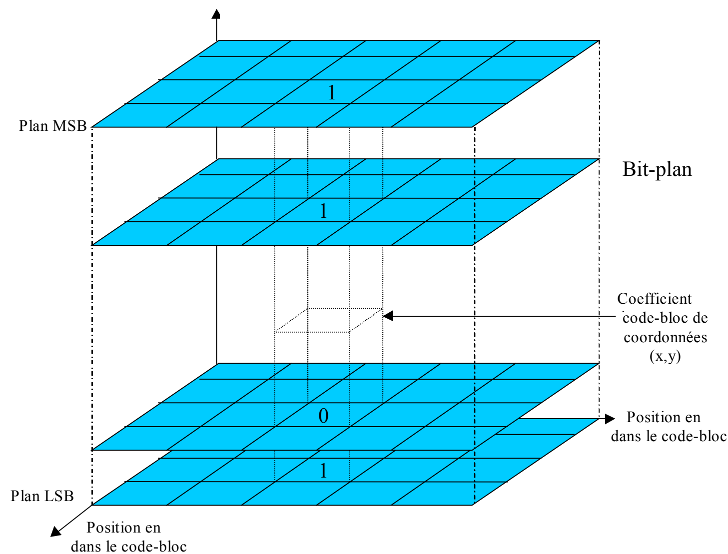


Fig. VI. 8 : Représentation en bit-plan d'un code-block

Les échantillons d'un code-bloc sont rangés en bit-plans. Ces plans démarrent par le bit de poids fort (MSB). Les différents bit-plans sont codés par le bloc de traitement 'Coefficient Bit Modeling' représenté dans la chaîne de compression JPEG2000 de la figure VI.6. Le codage se décompose en trois étapes appelées aussi passes : la passe de signification (Significance Pass), la passe d'affinage (Refinement Pass) et la passe de nettoyage (Cleanup Pass). L'objectif de ces passes est de préparer la phase de codage arithmétique.

Avant de décrire ces trois passes, indiquons qu'un échantillon est dit significatif si le bit de poids le plus fort de ce coefficient a déjà été codé. A chaque coefficient d'un code-bloc est associée une variable d'état appelée signification. Cette variable devient vraie (coefficient significatif) quand le premier bit à 1 est le MSB du coefficient. Pour chaque coefficient, on définit également un vecteur de contexte qui se réfère à l'état de ses huit voisins. Tous les coefficients voisins à l'extérieur d'un code-bloc sont considérés non significatifs. Le nombre de contextes possibles est cependant réduit à 19 (0 à 18). Les règles d'assignement du contexte sont dépendantes du type de passe. Les valeurs de contexte et les valeurs binaires issues de chaque passe sont ensuite envoyées au codeur arithmétique.

1. Passe de signification: codage des bits (0 ou 1) des échantillons non significatifs ayant au moins un voisin significatif, plus éventuellement transformation d'un échantillon non significatif en significatif dans le cas du codage d'un bit à 1 (il s'agit alors du premier bit à 1 pour cet échantillon).
2. Passe d'affinage : codage des bits (0 ou 1) des échantillons significatifs non encore codés.
3. Passe de nettoyage : codage du reste des bits des échantillons non significatifs (0 ou 1). Si on code un bit à 1 (premier bit à 1 pour cet échantillon), on fait alors passer l'échantillon à l'état significatif. Cette passe contient aussi un mécanisme de codage par plage permettant de coder les séquences consécutives de zéros (Run-length coding).

Finalement, la chaîne de compression effectue un codage arithmétique sur le flux binaire constitué du contexte et de la valeur binaire du bit codé (CX, D). Dans la troisième section, nous allons présenter l'architecture matérielle du codeur entropique de la chaîne JPEG2000. La description de l'architecture est effectuée en utilisant la bibliothèque SystemC.

VI.4.2. Architecture SystemC du codeur entropique

Les travaux effectués dans le cadre de la modélisation SystemC du codeur entropique focalisent sur les aspects architecturaux de conception au niveau système. L'évaluation de la performance de cette architecture est effectuée en mesurant le nombre de cycles d'exécution.

Le codeur entropique du standard JPEG 2000 est l'étape la plus complexe en termes de traitement. En effet, cette phase utilise une grande quantité d'information durant l'exécution de plusieurs opérations présentant un caractère séquentiel. L'aspect séquentiel des traitements complique la tâche qui vise la modélisation d'une architecture parallèle du codeur. Notre but se résume à concevoir une architecture présentant un maximum de parallélisme tout en respectant les contraintes de dépendance qui existent entre les différentes opérations effectuées par le codeur. La validation de cette architecture passe par trois phases qui sont la conception, l'implémentation et le test.

VI.4.2.1. Conception

La phase de conception démarre par une description fonctionnelle détaillée de l'application. Nous commençons l'étape de spécification par la définition d'une entité regroupant un ensemble de bits appartenant à un seul bit-plan. Cette entité appelée 'stripe' regroupe tous les bits utilisés par une passe à un instant fixé. Un 'stripe' est composé à la fois par quatre lignes de coefficients appartenant à un seul bit-plan. Nous définissons une nouvelle entité appelée 'stripe étendu' (Fig. VI.9) composée d'un 'stripe' entouré par son voisinage de bits. Le voisinage gauche et droit du stripe est constitué en appliquant le principe du miroir. En revanche, le voisinage bas ainsi que haut est constitué à partir des 'stripes' voisins. L'étape d'insertion des voisinages est essentielle pour coder les bits appartenant aux bords gauche, droit, haut et bas du 'stripe'. Le 'stripe étendu' constitue le paquet de communication de base entre les différentes passes.

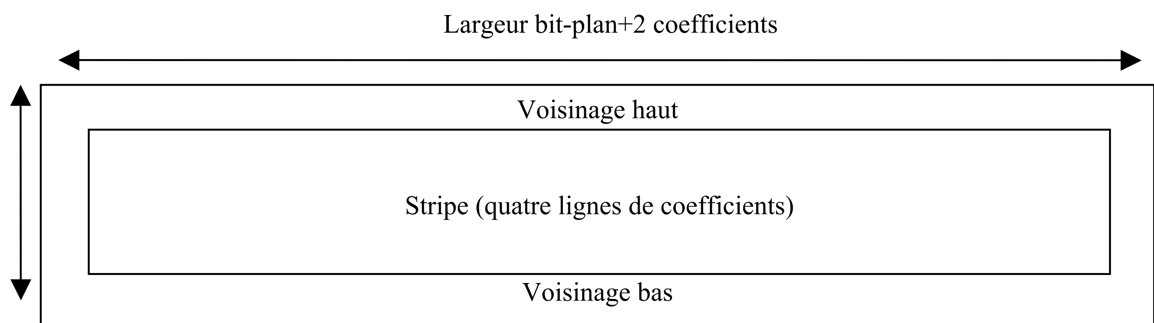


Fig. VI. 9 : Composition d'un 'stripe étendu'

Le principe de codage entropique associe à chaque bit appartenant au 'stripe' traité trois indicateurs appelés (Pi, Sigma et Sigma bar). En plus des différents 'stripes' de données, les modules effectuant les trois passes chargent dans leurs mémoires internes les coefficients associés à chaque bit du 'stripe' qui sont le Pi, Sigma, Sigma bar et le signe. Les coefficients Pi, Sigma et Sigma bar sont initialisés à zéro. Les valeurs de ces indicateurs changent après l'exécution de chaque passe.

Nous notons que le traitement des passes sur un 'stripe' ne peut être effectué qu'après le chargement des cinq entités (Sigma, Pi, Sigma bar, bit et signe) dans la mémoire interne du module effectuant la passe. L'étape de mise à jour des différents indicateurs est effectuée après l'exécution des trois passes. La figure VI.10 expose le diagramme d'état des traitements effectués par le bloc 'Coefficients Bit Modeling'.

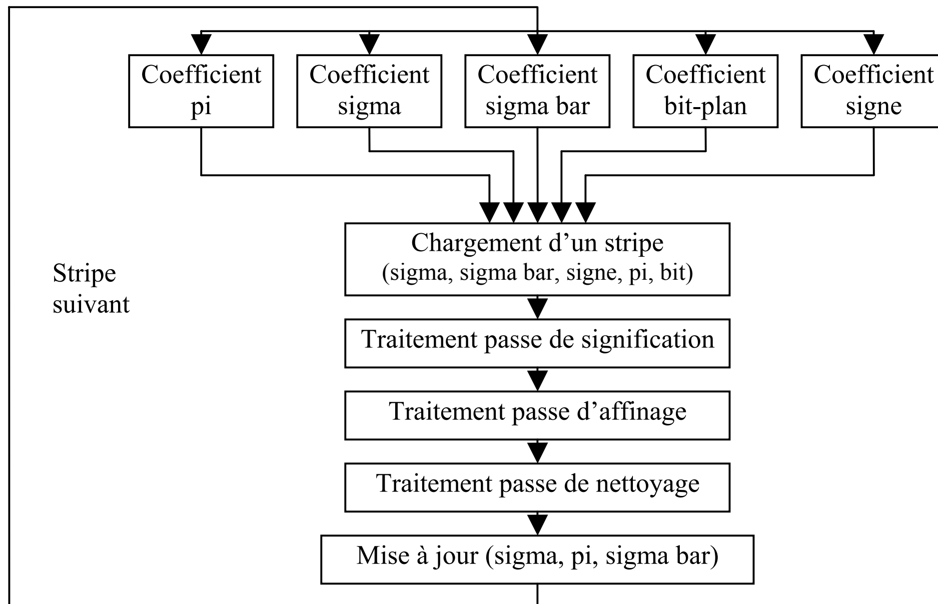


Fig. VI. 10 : Traitement de 'Coefficient Bit Modeling'

L'étape déterminante dans la phase de conception est l'étude de dépendance entre les différentes tâches qui composent l'architecture du codeur. Cette étude a permis l'extraction des traitements parallèles de l'application. Nous proposons alors de développer une architecture fonctionnant selon le principe du pipeline. Le codage commence par la première passe (passe de signification). Le module traitant cette passe commence par le chargement du premier stripe avec son voisinage (stripe étendu). Après l'achèvement du chargement, le module commence alors les traitements sur les coefficients du premier stripe. Ces traitements génèrent des sorties qui sont interceptées par le module traitant la passe d'affinage. Une fois que le module effectuant la passe de signification ait terminé ses traitements, alors il passe directement au chargement du deuxième stripe. En même temps le module passe d'affinage, qui a terminé le chargement des sorties du premier passe, passe au traitement. Chaque module effectuant soit la passe de signification (passe0), la passe d'affinage (passe1) ou la passe de nettoyage (passe2) effectue deux opérations successives qui sont le chargement (notée chstri) et le traitement (notée trstri) des coefficients du $i^{éme}$ stripe étendu à traiter. Le traitement en pipeline qui caractérise l'architecture développée est représenté par la figure VI.11:

Passe0	chstr1	trstr1	chstr2	trstr2	chstr3	trstr3	chstr4	trstr4	chstr1	trstr1	chstr2	trstr2	chstr3	trstr3	.....
Passe1		chstr1	trstr1	chstr2	trstr2	chstr3	trstr3	chstr4	trstr4	chstr1	trstr1	chstr2	trstr2	chstr3	trstr3
Passe2			chstr1	trstr1	chstr2	trstr2	chstr3	trstr3	chstr4	trstr4	chstr1	trstr1	chstr2	trstr2	chstr3

Fig. VI. 11 : Pipeline préliminaire

En considérant ce pipeline, on remarque la présence de deux dépendances non respectées:

- La première dépendance se résume à l'exécution de l'opération trstr1 de la passe d'affinage qui s'exécute, dans ce cas, avant l'opération de trstr2 de la passe de nettoyage. En effet le traitement d'un stripe par la passe d'affinage (respectivement par la passe de nettoyage) nécessite le chargement de deux stripes déjà traités par la passe de signification (respectivement par la passe d'affinage). Le deuxième stripe traité sert à mettre à jour le voisinage bas du stripe à traiter.

- La deuxième dépendance commence avec le traitement du deuxième bit plan. En effet, il faut synchroniser le fonctionnement de ce pipeline de façon à ce que l'on ne commence le traitement du deuxième bit-plan qu'après le traitement de deux stripes du premier bit plan par le passe2. Les informations de sortie du module passe2 servent à mettre à jour les données des coefficients Sigma, Pi et Sigma bar.

Pour rectifier le comportement du pipeline, il faut ajouter une certaine synchronisation entre les tâches indiquées dans la figure VI.11 afin de satisfaire les dépendances de données entre les différentes passes. Cette synchronisation est effectuée en ajoutant des phases d'attente. Le pipeline final déployé est présenté par la figure VI.12:

Passe0	chstr1	trstr1	chstr2	trstr2	chstr3	trstr3	chstr4	trstr4			chstr1	trstr1	chstr2	trstr2	....
Passe1		chstr1		chstr2	trstr1	chstr3	trstr2	chstr4	trstr3			chstr1	trstr4	chstr2	trstr1
Passe2					chstr1		chstr2	trstr1	chstr3	trstr2			chstr4	trstr3	chstr1

Fig. VI. 12 : Pipeline respectant la dépendance des données

Ce pipeline montre l'aspect parallèle qui caractérise le fonctionnement du module de codage 'Coefficient Bit Modeling'. En effet, dans cette version, toutes les passes fonctionnent simultanément soit en chargement ou en traitement. Le deuxième avantage de cette version c'est que l'on n'est pas obligé d'attendre la fin de traitement du premier bit plan pour commencer le traitement du deuxième. En revanche, le fonctionnement simultané des modules effectuant la passe de signification, la passe d'affinage et la passe de nettoyage entraîne un désordre au niveau des sorties du 'Coefficient Bit Modeling'. Ce qui implique la nécessité d'ajouter un module supplémentaire qui sert à stocker et réorganiser ces sorties.

Finalement, nous exposons le fonctionnement global des différentes passes (Fig.VI.13) pour charger, traiter et mettre à jour les coefficients des bit-plans d'une image.

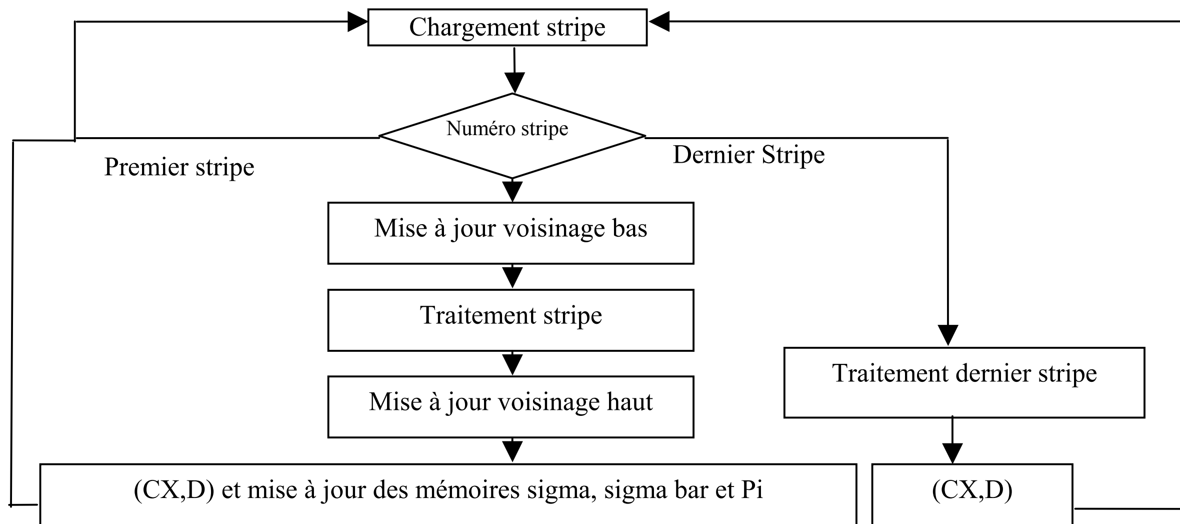


Fig. VI. 13 : Diagramme d'état des différentes passes

Le fonctionnement global des différentes passes commence par le chargement d'un 'stripe étendu' à partir des différentes mémoires. Le fonctionnement de la passe est fixé selon l'ordre du stripe chargé. Pendant l'étape de mise à jour du voisinage bas, nous remplaçons les coefficients du voisinage du premier 'stripe étendu' par la première ligne du deuxième stripe. Pendant l'étape de mise à jour du voisinage haut, nous remplaçons les coefficients du voisinage du deuxième 'stripe étendu' par la dernière ligne du premier stripe. Finalement, le

module effectuant la passe remplace le premier stripe par le deuxième. La sortie de l'opération de codage est un ensemble de couples (CX, D). Le symbole CX représente le contexte relatif à chaque donnée codée D. Les couples (CX, D) représentent les entrées de l'étape du codage arithmétique.

VI.4.2.2. Implémentation en SystemC

L'architecture du 'Coefficient Bit Modeling', effectuant le codage du bit-plan pour servir le codeur arithmétique, est implémentée en utilisant la bibliothèque SystemC [9]. La description est effectuée au niveau comportemental. Ce choix est dû à la richesse de la bibliothèque SystemC au niveau des structures dédiées à la description haut niveau d'architectures embarquées. L'architecture du 'Coefficient Bit Modeling' est exposée dans la figure VI.14.

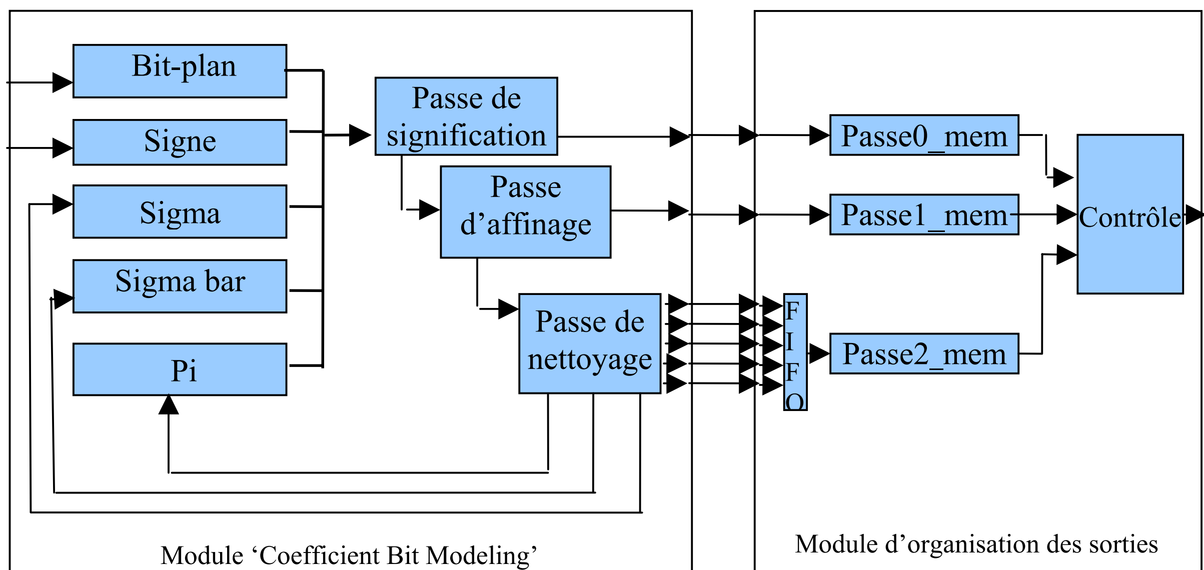


Fig. VI. 14 : Architecture du module 'Coefficient Bit Modeling '

L'architecture qui implémente cette version du 'Coefficient Bit Modeling' du codeur entropique présente deux grands modules:

- Un module représentant les différentes passes du 'Coefficient Bit Modeling'. Ce sous module comporte huit modules (5 mémoires et 3 modules de traitement). Les mémoires 'Bit-plan' et 'Signe' stockent respectivement les bit-plans des données et les signes du flux à coder. Les mémoires 'Sigma', 'Sigma bar' et 'Pi' stockent des résultats intermédiaires de codage.
- Un module d'organisation des sorties du 'Coefficient Bit Modeling' qui se compose de 4 modules (3 mémoires et un module d'arbitrage et de contrôle qui organise le fonctionnement des 3 mémoires). En effet, l'aspect pipeline de l'architecture proposée engendre une perturbation au niveau du flux de sortie binaire de l'étape de codage. Ce module stocke et organise les sorties des différentes passes.

VI.4.3. Codeur entropique parallèle et reconfigurable en ligne

L'architecture parallèle se base sur un ensemble de codeurs entropiques. Le caractère indépendant des traitements appliqués sur plusieurs code-blocs permet d'implémenter un système de codage basé sur plusieurs codeurs. Chaque codeur effectue le codage d'un seul code-bloc à la fois. L'architecture proposée se base sur un mécanisme de prédiction neuronale qui estime le nombre de cycles nécessaires pour coder les prochains code-blocs. Un système de décision, intégré dans l'architecture, se base sur les valeurs de performances prédites afin

d'effectuer le contrôle, l'allocation et l'organisation des différents codeurs. L'objectif est de garder une performance de codage respectant les contraintes de traitement temps réel tout en réduisant le nombre de codeurs entropique [10]. Cette réduction implique l'optimisation de l'implémentation de l'architecture en termes de surface et de consommation d'énergie. Cette méthodologie est applicable sur tous les systèmes à charge de travail variable et qui réalisent un traitement indépendant. Le schéma de la figure VI.15 représente l'architecture reconfigurable développée.

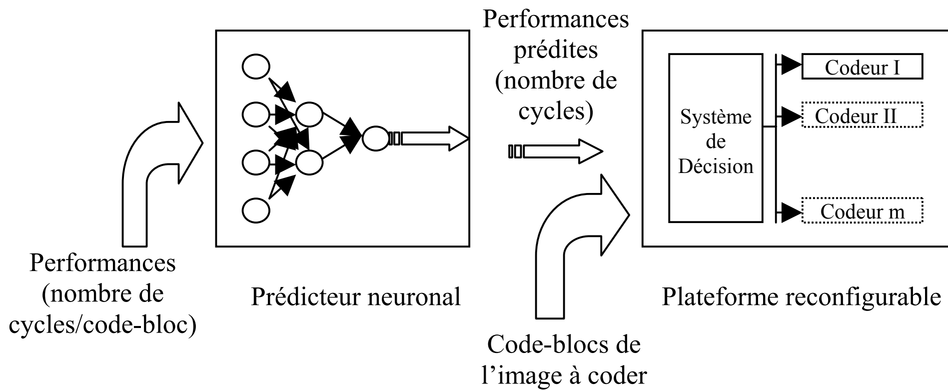


Fig. VI. 15 : Architecture parallèle et reconfigurable de codage entropique

En se basant sur le schéma de base de la figure IV.14 adopté pour implémenter un système reconfigurable au niveau système, nous présentons le diagramme (Fig.VI.16) d'état qui détermine l'évolution du système. Il est essentiel de définir d'abord la notion d'époque de reconfiguration. Une époque de reconfiguration est le délai nécessaire pour achever le codage d'un nombre constant n de code-blocs en utilisant un nombre ω ($\omega \leq n$) variable de codeurs.

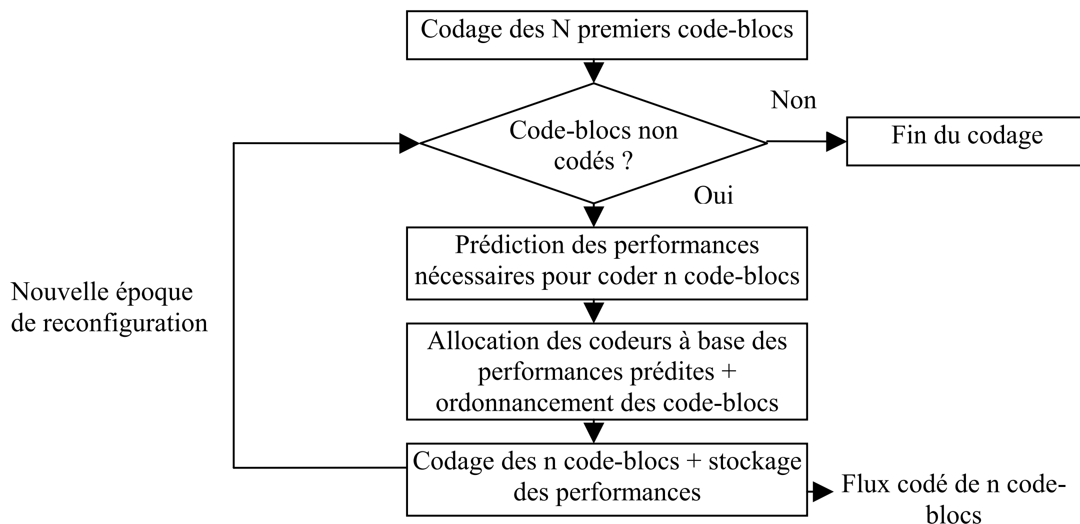


Fig. VI. 16 : Diagramme d'état de l'architecture reconfiguarble du codeur entropique parallèle.

L'application démarre par le codage de N code-blocs de l'image. N représente le nombre de performances requis pour démarrer le processus de prédiction neuronale. L'architecture proposée effectue le codage de n code-blocs en utilisant ω codeurs ($\omega \leq n$). Tant qu'il y aura d'autres code-blocs à coder, l'architecture prédit les performances de codage des n code-blocs prochains. En se basant sur ces valeurs prédites, l'application décide sur le nombre de codeurs à implémenter. Dans ce contexte, un codeur peut traiter un ou plusieurs code-blocs. Selon la charge de travail de chaque codeur, l'application décide alors d'affecter un ou plusieurs code-

blocs à un seul codeur. Nous notons alors la présence de deux étapes critiques dans ce schéma de reconfiguration effectuant la prédiction neuronale et l'affectation d'un code-bloc au codeur traitant.

VI.4.3.1. Prédiction neuronale

La prédiction des performances de codage des différents code-blocs est effectuée par un réseau de neurones unique. L'apprentissage est effectué à l'aide d'une série de performances générée à la suite du codage de l'image 'Lena.ppm'. Le test de généralisation est effectué à l'aide de trois séries générées en codant les images 'Barbara.ppm', 'baboon.ppm' et 'fruits.ppm'. Les séries d'apprentissage et de généralisation se composent de 777 coefficients qui représentent les performances de codage de 777 code-blocs. Dans le tableau VI.4, nous présentons les performances d'apprentissage et de généralisation du réseau de neurones récurrent appliqué à la prédiction des performances de codage. Les performances sont présentées en MSE.

Table. VI. 4 : Résultats d'apprentissage et de généralisation relatifs àla prédiction des séries de performances

	apprentissage Lena.ppm	généralisation Baboon.ppm	généralisation Barbara.ppm	généralisation Fruit.ppm
MSE	2.2146	3.7609	4.6890	3.7554

Les figures VI.17, VI.18, VI.19 et VI.20 exposent les résultats de prédiction des performances de codage des différents code-blocs en utilisant un réseau de neurones récurrent. Ces figures représentent également une sorte de zoom sur les résultats obtenus.

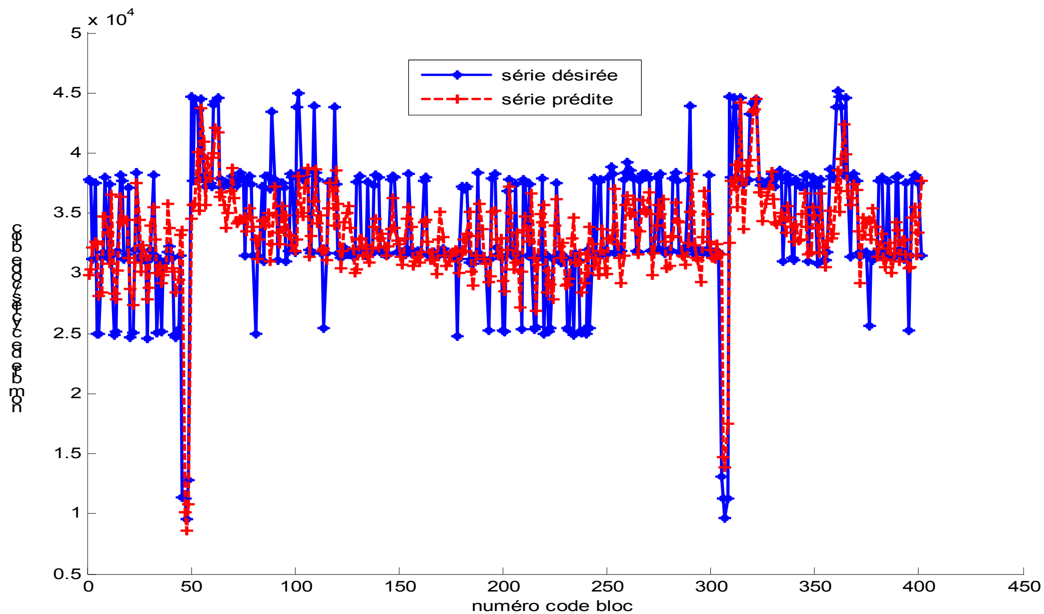


Fig. VI. 17 : Résultats de prédiction des performances de codage d'une partie des code-blocs de 'Lena.ppm' (apprentissage)

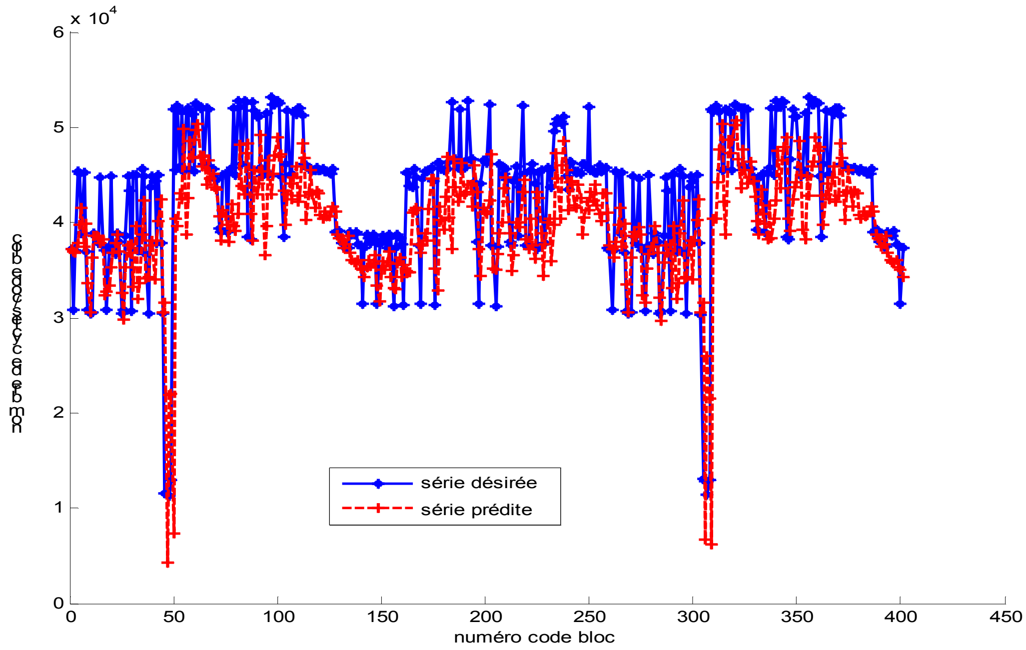


Fig. VI. 18: Résultats de prédiction des performances de codage d'une partie des code-blocs de 'baboon.ppm' (généralisation)

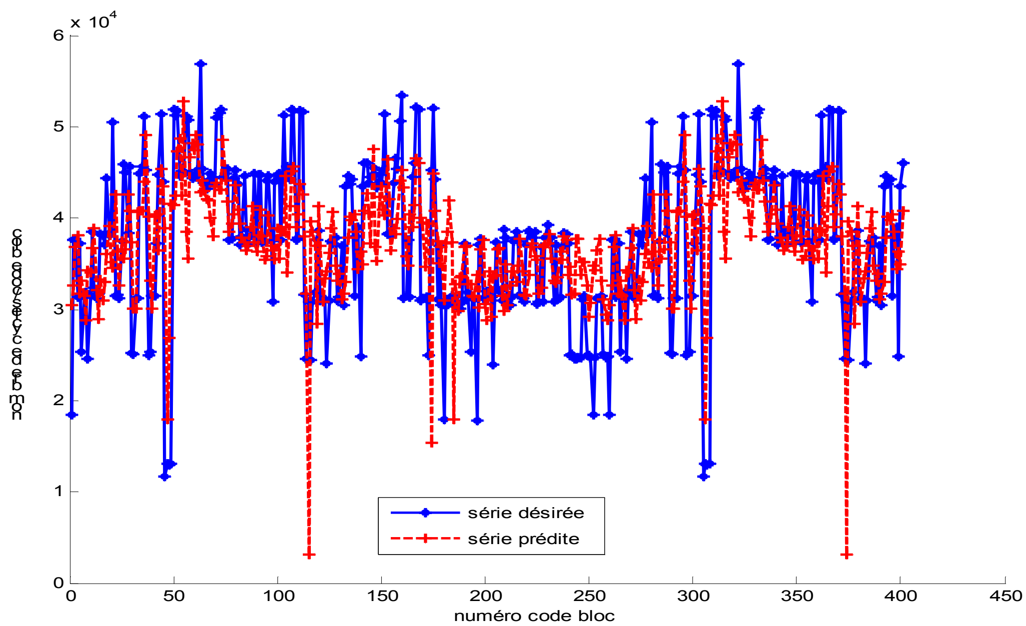


Fig. VI. 19 : Résultats de prédiction des performances de codage d'une partie des code-blocs de 'Barabara.ppm' (généralisation)

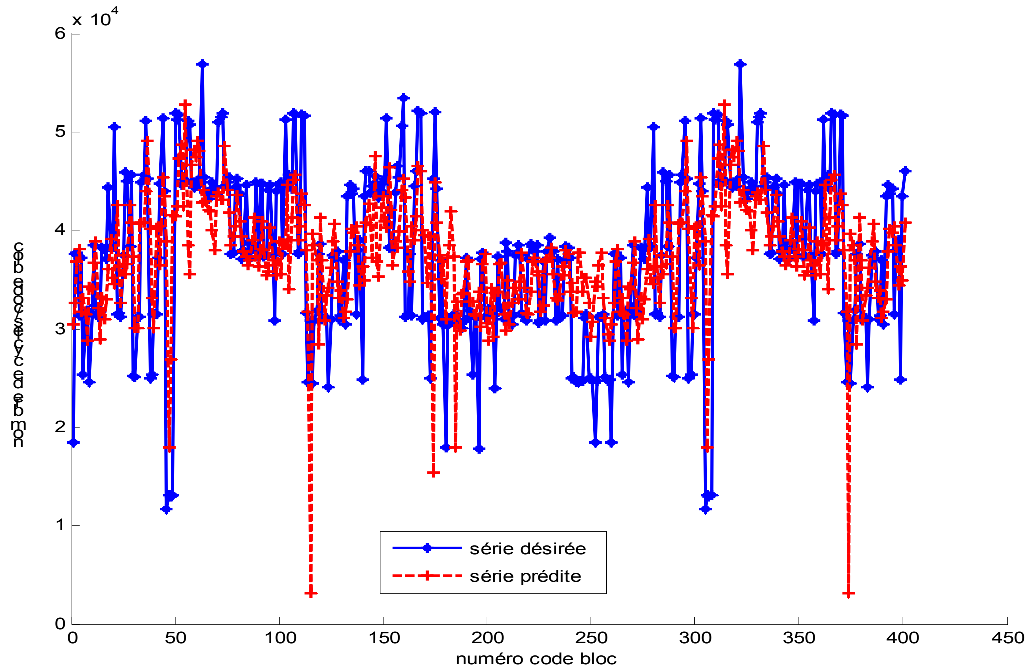


Fig. VI. 20 : Résultat de prédiction des performances de codage des code-blocs de 'fruit.ppm'

VI.4.3.2. Système de décision

Le système de décision associe chaque code-bloc d'une image à son codeur traitant. Le système de décision organise le codage de sorte que le traitement des n code-blocs ne dépasse pas un délai fixé à l'avance. Ce système de décision peut être considéré comme un système d'ordonnancement. Nous détectons la présence de deux approches d'ordonnancement possibles. La première approche optimise l'affectation des code-blocs en tenant compte d'une contrainte de délai locale (relative à l'époque de reconfigration actuelle). La deuxième approche fixe l'ordre d'affectation des code-blocs en tenant compte d'une contrainte de délai globale (relative à toutes les époques de reconfiguration).

a) Approche d'ordonnancement avec une contrainte locale de délai

Le délai de temps occupé par une seule reconfiguration est décidé en déterminant le maximum prédit de cycles d'exécution par code-bloc. En outre, l'époque de reconfiguration s'étend chaque fois sur un délai équivalent à la valeur maximale $y_{\max}$ incluse dans le vecteur $(\hat{y}(k),...,\hat{y}(k+n))$. L'algorithme d'ordonnancement affecte en premier lieu le code-bloc qui exige $y_{\max}$ cycles au premier codeur P_1. En deuxième lieu, l'algorithme explore la totalité de combinaisons possibles. La phase d'affectation des code-blocs est exécutée de sorte que la somme des délais de codage des code-blocs traités par le codeur j ne dépasse pas $y_{\max}$ ($\sum_{i=1}^{h} \hat{y}_j(i) \leq y_{\max}$, avec h est le nombre de code-blocs traités par le codeur j). La solution sélectionnée est celle qui utilise le moins de codeurs. La figure VI.21 présente un exemple qui explique l'effet de cette approche dans le cas où n est égale à 5. Dans la figure VI.21, le terme C-Bi désigne le code-bloc d'ordre i.

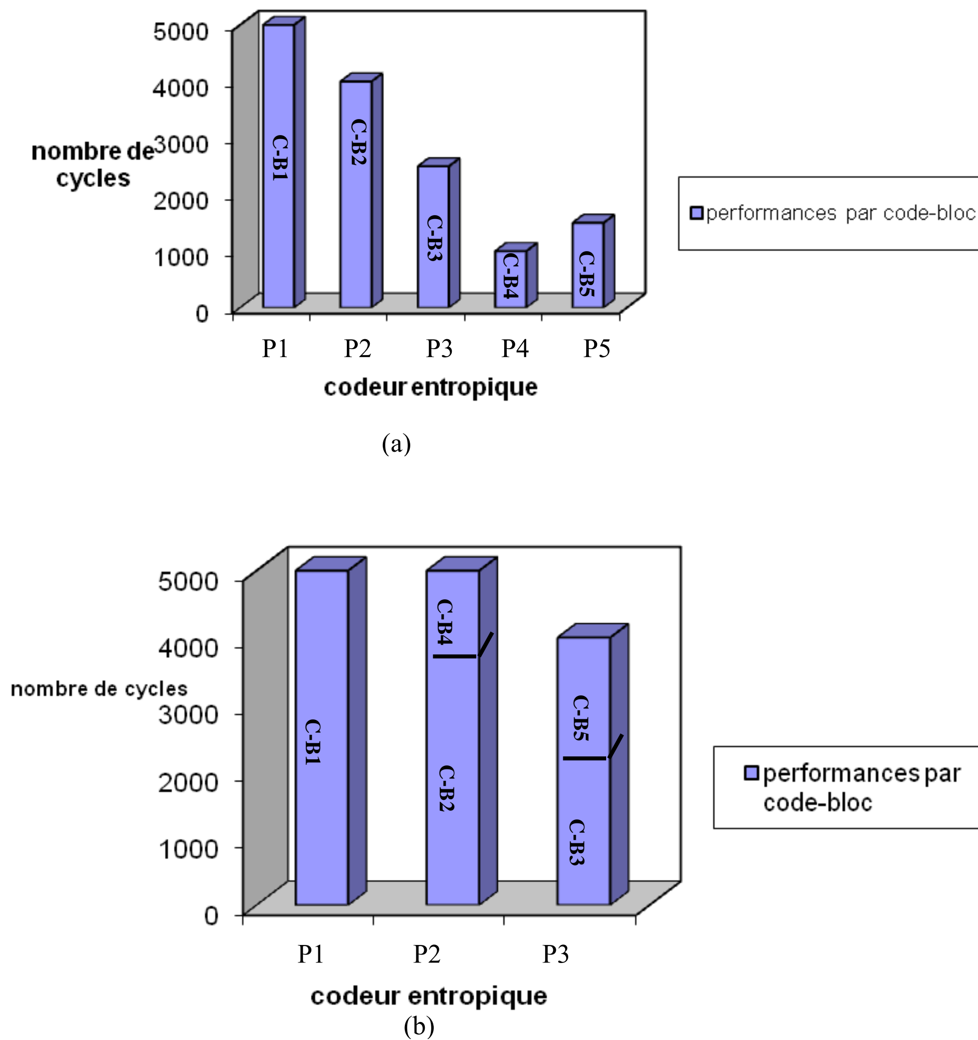


Fig. VI. 21 : Effet de l'ordonnancement selon l'approche locale : (a) distribution initiale des code-blocs, (b) distribution adoptée par le système de décision

En appliquant cette approche sur la plateforme reconfigurable effectuant le codage entropique parallèle et en fixant n à 5, nous aurons la variation du nombre des codeurs par époque de reconfiguration exposée par la figure VI.22.

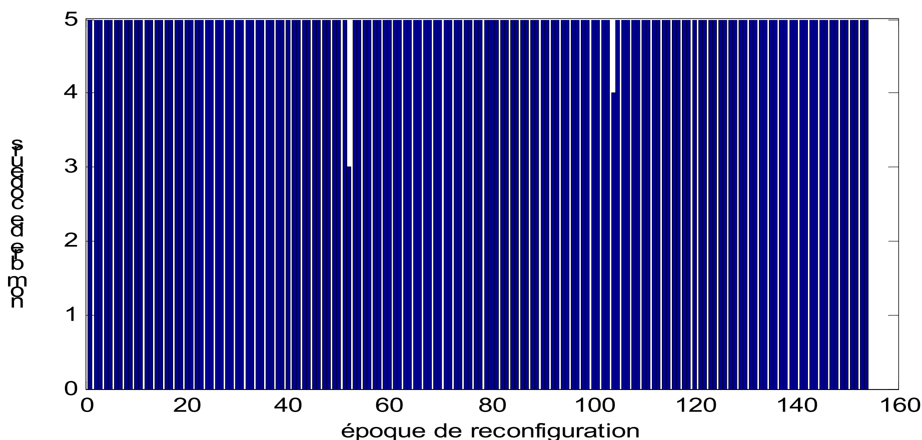
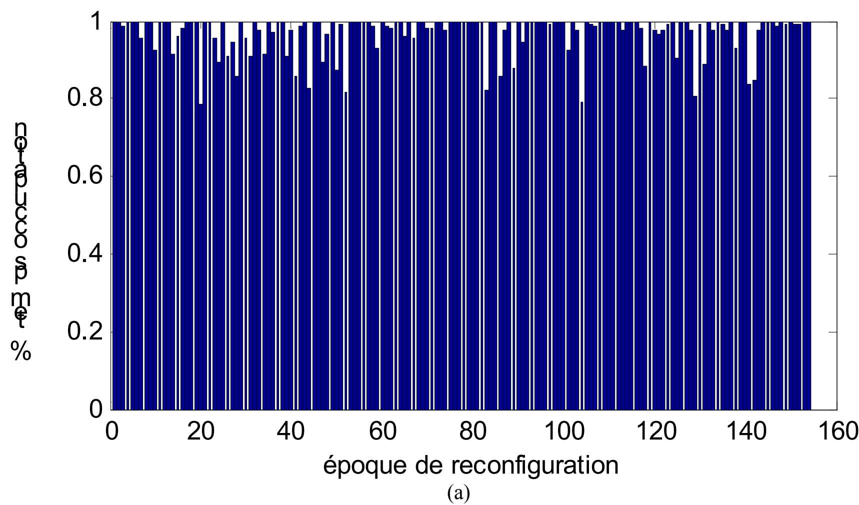


Fig. VI. 22 : Nombre de codeurs par époque de reconfiguration

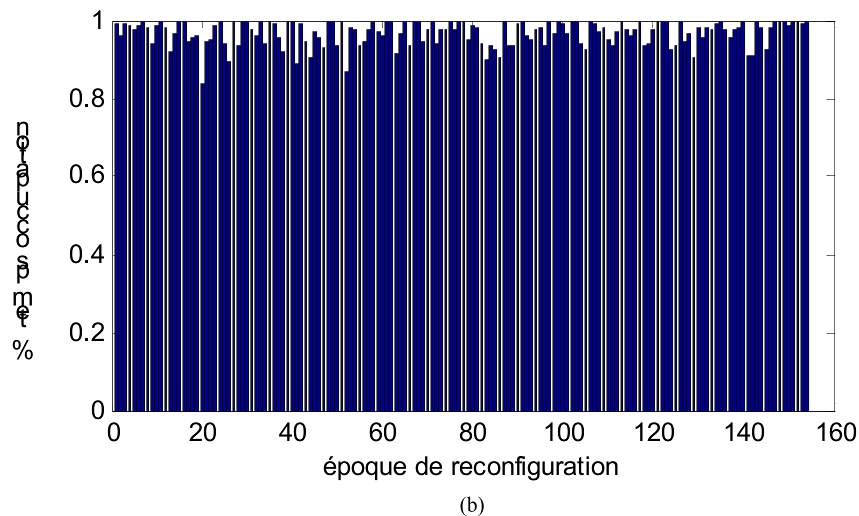
Nous remarquons que la variation du nombre de codeurs est négligeable. En effet, nous arrivons à varier le nombre de codeurs deux fois uniquement. La première variation est apparue au niveau de l'époque 52 où on déploie 3 codeurs. La deuxième variation est apparue au niveau de l'époque 103 où on déploie 4 codeurs. Nous définissons le taux d'occupation τ_j relatif à chaque codeur unique d'ordre j selon l'équation VI.3.

$$\tau_j = \frac{\sum_{i=1}^{h} \hat{y}(i)}{y_{\max}} \qquad \text{(VI.3)}$$

h représente le nombre de code-blocks traités par le codeur j. Dans la figure VI.23, nous exposons la variation du taux d'occupation de chaque codeur en fonction des époques de reconfiguration (cas n=5).



(a)



(b)

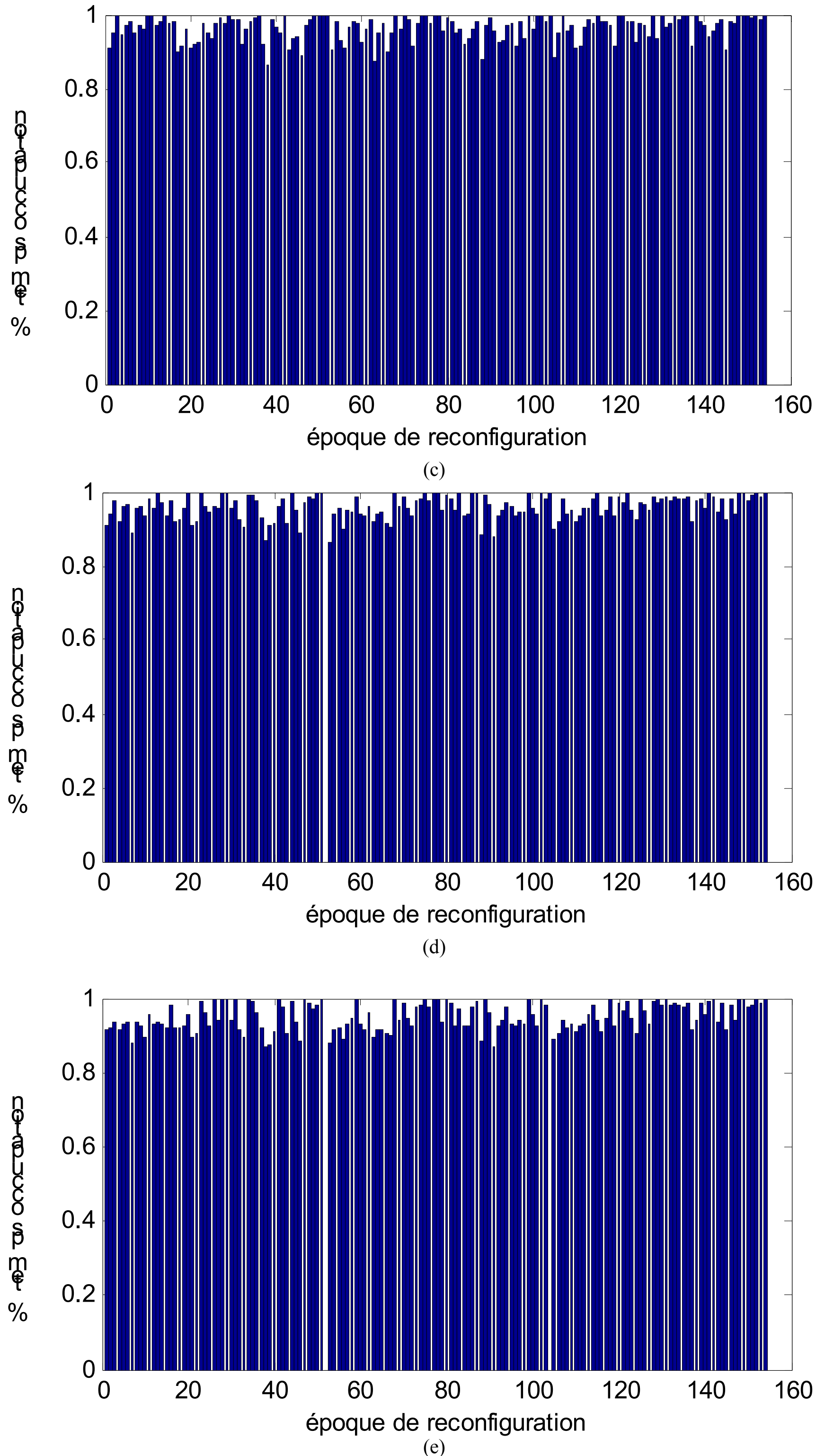
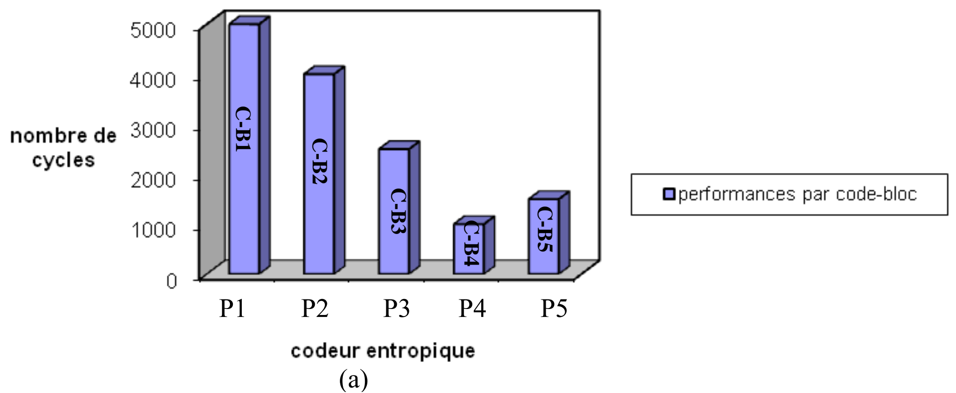


Fig. VI. 23 : Taux d'occupation des différents codeurs : (a) codeur P1, (b) codeur P2, (c) codeur P3, (d) codeur P4, (e) codeur P5.

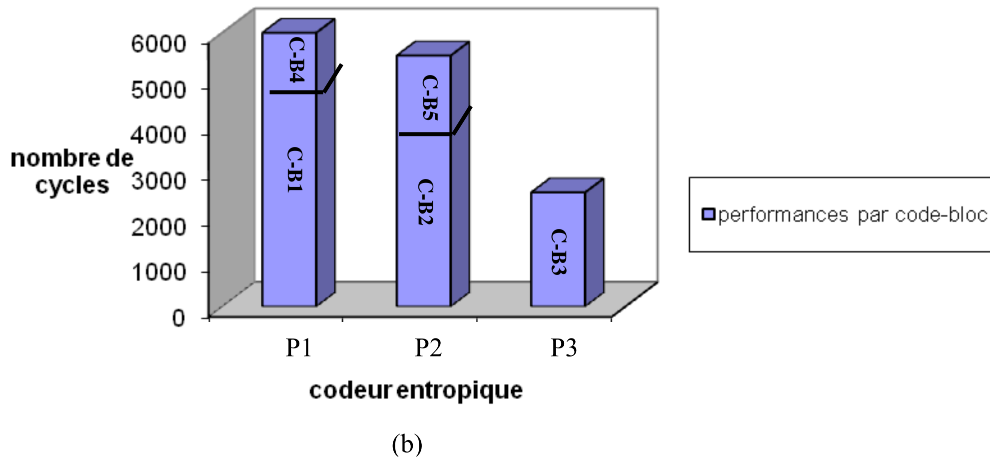
En analysant les valeurs numériques des taux d'occupation, nous remarquons que les différents codeurs sont en exécution plus que 80% du temps. Cette approche d'ordonnancement utilise efficacement les différents codeurs afin d'augmenter la performance de codage (délai minimal). Nous proposons par la suite une deuxième approche qui défavorise la performance de codage en allouant un nombre minimal de codeurs.

b) Approche d'ordonnancement avec contrainte globale de délai

Le délai de temps occupé par une seule reconfiguration est décidé en faisant référence à la contrainte de performance imposé par le caractère de traitement temps réel de l'application. En outre, l'époque de reconfiguration s'étend chaque fois sur un délai ρ ($\rho \geq \max((\hat{y}(k),...,\hat{y}(k+n)))$) fixé par l'utilisateur. L'algorithme d'ordonnancement démarre par l'exploration de la totalité de combinaisons d'affectation possibles. La phase d'affectation des code-blocs est exécutée de sorte que la somme des délais de codage des code-blocs traités par le codeur j ne dépasse pas ρ ($\sum_{i=1}^{h} \hat{y}_j(i) \leq \rho$, avec h est le nombre de code-blocs traités par le codeur j). La solution sélectionnée est celle qui utilise le moins de codeurs. La figure VI.24 expose un exemple qui explique l'effet de cette approche dans le cas où n est égal à 5 et $\rho = 6000\, cycles$. Dans la figure VI.24, le terme C-Bi désigne le code-bloc d'ordre i.



(a)



(b)

Fig. VI. 24 : Effet de l'ordonnancement selon l'approche globale : (a) distribution initiale des code-blocs, (b) distribution adoptée par le système de décision

En appliquant cette approche sur la plateforme reconfigurable effectuant le codage entropique parallèle et en fixant n à 5, nous aurons la variation du nombre de codeurs par époque de reconfiguration exposée par la figure VI.25.

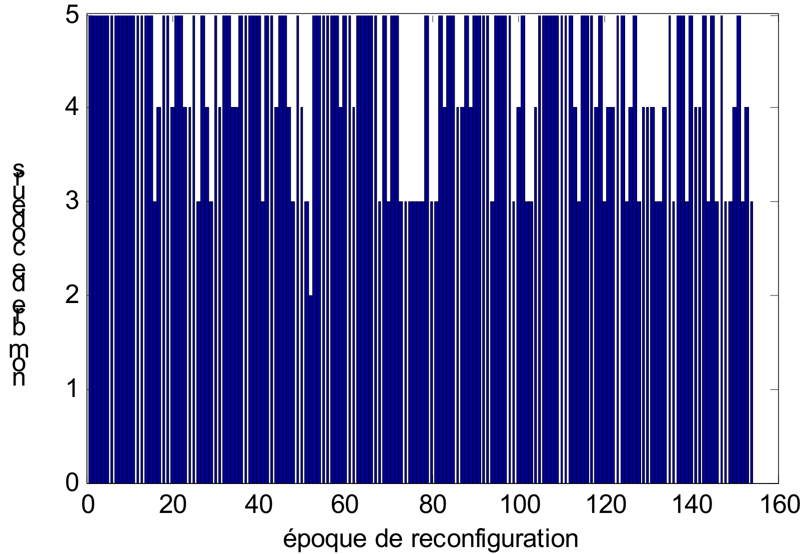
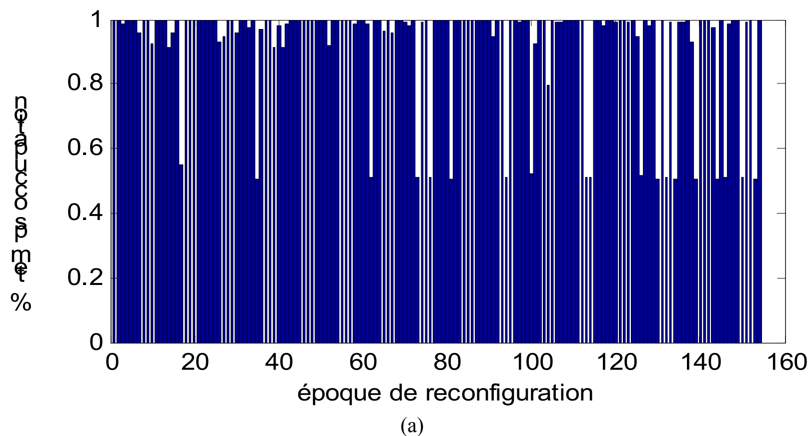


Fig. VI. 25 : Nombre de codeurs par époque de reconfiguration

Nous remarquons que la variation du nombre de codeurs est considérable par rapport à la première approche. En effet, nous arrivons à varier le nombre pendant plusieurs époques de reconfiguration. Nous arrivons par exemple à réduire le nombre de codeur à deux pendant la 51éme époque en gardant la même performance. Dans ce cas, nous définissons le taux d'occupation τ_j relatif à un codeur unique d'ordre j selon l'équation VI.4.

$$\tau_j = \frac{\sum_{i=1}^{h} \hat{y}(i)}{\rho} \qquad \text{(VI.4)}$$

Dans la figure VI.26, nous exposons la variation des taux d'occupation de chaque codeur en fonction des époques de reconfiguration (cas n=5 et $\rho = 6000\, cycles$).



(a)

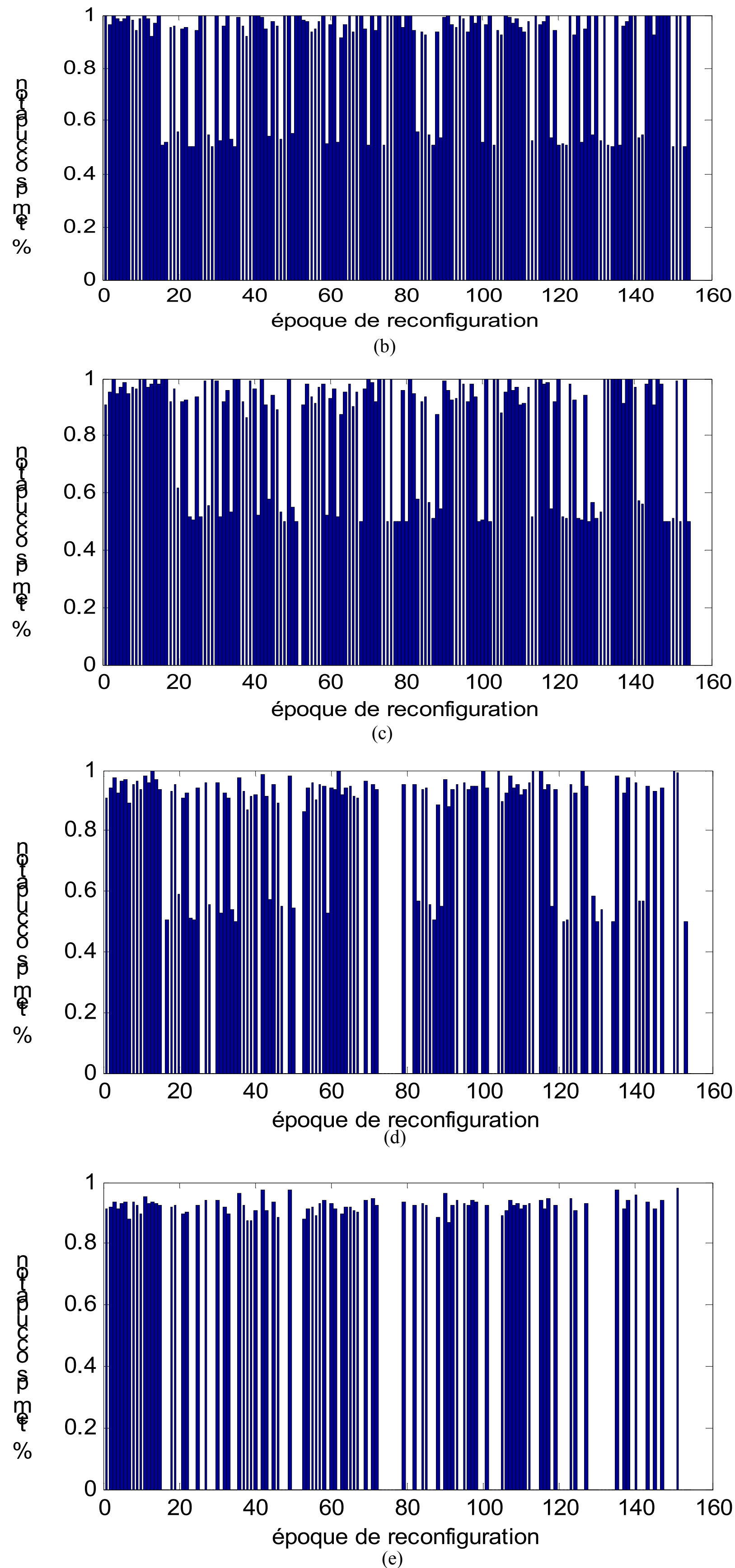


Fig. VI. 26 : Taux d'occupation selon l'approche globale des différents codeurs : (a) codeur P1, (b) codeur P2, (c) codeur P3, (d) codeur P4, (e) codeur P5.

En analysant les valeurs numériques des taux d'occupation, nous remarquons que ces taux varient entre 50% et 100% du temps d'exécution. En comparant les deux approches proposées, nous remarquons que la distribution des code-blocs selon la première approche est plus efficace (taux d'occupation plus grand). En revanche, la deuxième approche utilise moins de codeurs durant la phase de codage d'une image.

VI.6. Conclusion

Dans ce chapitre, nous avons exposé les travaux traitant la reconfiguration basée sur des métriques extraites à partir du niveau des instructions et des applications.

Les métriques extraites à partir d'un niveau instruction sont représentées par les adresses mémoire des instructions. Ces métriques sont caractérisées par leur variabilité. En outre, la série temporelle qui regroupe ces adresses est caractérisée par sa grande taille. Afin d'optimiser la capacité de prédiction appliquée sur une série de caractéristique pareil, nous avons proposé un modèle neuronal hybride appliqué à la prédiction. Ce modèle est composé par un bloc de classification est un autre de prédiction. Dans la première partie de ce chapitre, nous avons traité le problème de prédiction en montrant l'efficacité de l'approche hybride.

Dans la deuxième partie de ce chapitre, nous avons présenté un cas d'étude traitant un système reconfigurable implémenté au niveau système. Ce cas d'étude est représenté par un système de codage entropique parallèle regroupant plusieurs codeurs. En premier lieu, nous avons exposé le standard JPEG2000 ainsi que l'architecture SystemC mono-codeur développée. En deuxième lieu, nous avons exposé l'architecture multi-codeurs. Nous avons également présenté les stratégies de prédiction et d'ordonnancement dynamique. Cette stratégie organise l'affectation des code-blocs de l'image aux différents codeurs, en se basant sur les performances de codage par code-bloc estimées. L'estimation de ces performances est effectuée à l'aide d'un prédicteur neuronal.

En testant cette stratégie de conception de circuits réactifs, nous arrivons à diminuer le nombre de codeurs entropique implémentés durant la phase de codage en gardant la même performance. La stratégie de conception reconfigurable implémentée est applicable sur tous les systèmes qui comportent au moins une tâche (par exemple le codage entropique) appliquée itérativement sur des vecteurs d'entrées indépendants (code-blocks dans le cas de l'étude présentée dans ce chapitre). L'apport de cette approche est garanti si la performance de cette tâche varie en fonction du vecteur d'entrée traité. Ce type de tâche est notamment présent dans les applications multimédia.

VI.6. Références

[1] Taubman. D, M.W.Marcellin, “JPEG2000 – Image Compression Fundamentals, Standards and Practice “Kluwer Academic Publishers”, Nov. 2001.

[2] Y. Chen, B. Yang, J. Dong, A. Abraham, “Time series forecasting using flexible neural tree model”, Information sciences 174, 219-235, 2005.

[3] Stephan K. Chalup, Alan D. Blair, “Incremental training of first order recurrent neural networks to predict a context-sensitive language”, Neural Networks 16 (2003) 955–972.

[4] Derong. L, Tsu-Shuan. C, and Yi. Z, “A Constructive Algorithm For Feedforward Neural Networks With Incremental Training”, IEEE Transactions on circuits and systems-I: fundamental theory and applications, Vol. 49, No. 12, 2002.

[5] Jorg D. Wichard, Maciej Ogorzalek, “Time Series Prediction with Ensemble Models”, International Joint Conference on Neural Network, Budapest, 2004.

[6] JPEG 2000 Part I with Cor.1 Cor.2, Cor.3 and DCor.4, Amd.1, FPDAM.2, ISO/IEC JTC1/SC29/WG1 N2513R2, May 2002.

[7] S.Chtourou, M.Chtourou and O.Hammami, “Performance Evaluation of Neural Network Prediction for Data Prefetching in Embedded Applications”, Proceedings of the International Conference on Computer Science ICCS'06 Vienna, Vol. 12, Mars 29-31, 2006, ISBN 975-00803-1-9, Austria.

[8] S. Chtourou, M. Chtourou, O. Hammami, “A Hybrid Approach for Training Recurrent Neural Networks: Application to Multi-Step-Ahead Prediction of Noisy and Large Data Sets”, Accepté pour parution dans le journal Neural Computing & Applications (paru en ligne).

[9] S.Chtourou and O.Hammami, “SystemC Modeling of JPEG-2000 components : Discrete Wavelet Transform and Entropy coder”, 3rd International Symposium on Image/Video communications over fixed and mobile networks, 13-15 September, Tunisia.

[10] S.Chtourou, O.Hammami and M.Chtourou, “JPEG-2000 Workload Prediction for Adaptive System on Chip Entropy Coders Architecture”, WCCI 2006 IEEE International Joint Conference in Neural Network, July 16-21, Vancouver, Canada.

VII. Conclusion générale

Le travail présenté dans cette thèse traite des méthodologies de conception de systèmes sur puce exécutant des charges de travail présentant une variabilité significative. Cette variabilité significative empêche l'utilisation des flots de conception classiques. Généralement, cette variabilité se traduit par un surdimensionnement inacceptable des plateformes intégrant ces applications. Ce surdimensionnement à pour conséquence des coûts, des délais de simulation et de consommation d'énergie incompatibles avec les contraintes considérables existant dans l'industrie des semi-conducteurs.

La constatation qui a été établie en analysant plusieurs fonctions est que cette variabilité est multiple, à grains variables, composée de différents niveaux de la hiérarchie système et statiquement imprévisible.

Comment, à partir de cela, concevoir un système et sur quelle philosophie de conception doit on baser des propositions de nouveaux outils de conception ?

La démarche suivie dans ce travail est basée sur l'introduction d'un mécanisme unique de prédiction dynamique de charge de travail couplé avec un reconditionnement dynamique des ressources matérielles. Cette philosophie englobe tous les niveaux de variabilité qui existent dans le système et de ce fait ouvre des pistes nouvelles dans la recherche des systèmes adaptatifs sur SOPC.

Ce travail diffère des travaux sur la reconfigurabilité dynamique où le problème est celui d'une exploitation optimale de ressources sous contraintes de surface en gardant une performance optimale. En effet, dans ce cadre la, le concepteur conserve la maîtrise du flot de conception à partir d'une connaissance statique parfaite de son application. Les temps d'exécution des différentes fonctions, la surface nécessaire et le placement sont connus. Le cadre de travail de cette thèse est fondamentalement différent car le rôle du concepteur n'est plus de décider statiquement de l'allocation dynamique des ressources et l'ordonnancement des tâches mais de concevoir un contrôleur basé sur la prédiction dynamique de charge capable de prendre dynamiquement et de manière autonome ces décisions.

Dans cette thèse, nous avons mis en évidence un flot de conception basé sur cette philosophie.

Des analyses comparatives, extraites de la littérature relative la prédiction, nous ont permis de choisir des techniques à base de réseaux de neurones pour le mécanisme de prédiction. La question qui devait alors être traitée était le surcoût en surface et en consommation d'énergie d'une telle technique. Nous avons donc proposé un flot de conception multi objectifs de réseaux de neurones qui nous a permis de conclure que l'insertion d'un tel réseau de neurones ne présente aucune surcharge au niveau de la plateforme.

De nombreuses problématiques nouvelles ont été traitées. Nous pouvons citer en particulier, les prédictions de séries de temps de grande taille, l'inclusion d'un predicteur de charge variable au sein d'un système sur puce, des flots de conception multi objectifs des réseaux de neurones et l'ordonnancement post prédiction pour un nombre d'IPs variable. Ce travail a ses limites car il ne modifie pas la variabilité et la considère comme une donnée d'entrée non modifiable. On pourrait envisager la fusion ou la décomposition de variabilités.

La pluridisciplinarité et la transversalité de cette thèse ont permis d'établir des bases pour de nombreuses autres perspectives. En particulier, la question d'échantillonnage des séries de temps de grande taille suscite la recherche d'un nouveau concept de signature système. En effet, s'il est possible d'échantillonner une charge de travail variable, il est alors possible d'obtenir une projection d'un système dans un sous espace de systèmes. La notion de plateforme actuellement répandue peut être considérée comme un dérivatif de cette signature. Il s'ouvre alors un nouveau axe de recherche traitant la création de système comme une

combinaison de sous systèmes représentant des bases d'un 'espace vectoriel' de systèmes. La signature peut être alors vue comme la décomposition en ses éléments dans la base. Fondamentalement, la conception d'un système sur puce rentre dans une phase plus théorique ayant plus de fondements mathématiques.