Ordonnancement temps réel pour architectures hétérogènes reconfigurables basé sur des structures de réseaux de neurones

Antoine Eiche

N° d'ordre : 2012REN1E008 ANNÉE 2012

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Traitement du Signal

Ecole doctorale Matisse

présentée par

Antoine EICHE

préparée à l'unité de recherche IRISA / INRIA Rennes,
Institut de Recherche en Informatique et Systèmes Aléatoires
ENSSAT

Ordonnancement temps réel pour architectures hétérogènes reconfigurables basé sur des structures de réseaux de neurones

Thèse soutenue à l'ENSSAT le 14/09/2012

devant le jury composé de :

Eric MARTIN
Professeur des Universités, Université de Bretagne Sud / président

Yvon TRINQUET
Professeur des Universités, Université de Nantes / rapporteur

Olivier TEMAM
Directeur de Recherche, INRIA Saclay / rapporteur

Didier DEMIGNY
Professeur des Universités, Université de Rennes 1 / examinateur

Daniel CHILLET
Maitre de Conférences HDR, Université de Rennes 1 / directeur de thèse

Sébastien PILLEMENT
Professeur des Universités, Université de Nantes / co-directeur de thèse

Remerciements

Je remercie tout particulièrement Monsieur Daniel Chillet, Maître de Conférences à l'Universités de Rennes 1 pour avoir accepté de diriger cette thèse, et m'avoir fait bénéficier de son encadrement. Je remercie également Monsieur Olivier Sentieys, Professeur des Universités à l'Université de Rennes 1 et responsable de l'équipe CAIRN, pour m'avoir accueilli dans son équipe de recherche. Soyez assurés de ma gratitude.

Je tiens également à remercier Monsieur Eric Martin pour avoir accepté de présider le jury de soutenance, Messieurs Yvon Trinquet et Olivier Temam pour avoir accepté de rapporter sur ce manuscrit et Monsieur Didier Demigny pour avoir accepté d'examiner ces travaux.

Que tous les membres de l'équipe CAIRN et le personnel de l'ENSSAT acceptent mes remerciements pour avoir contribué à rendre cette expérience enrichissante aussi bien sur le plan professionnel que personnel.

Mes derniers remerciements, mais non des moindres, vont à Christine et Pierre pour avoir su me supporter durant ces trois dernières années, ainsi qu'à mes parents pour m'avoir permis d'étudier.

Je dédie l'ensemble de ce travail à mon père, François Eiche.

Table des matières

Introduction **7**

1 Ordonnancement spatio-temporel pour architectures reconfigurables 13
1.1 Architectures reconfigurables 13
1.1.1 Architectures configurables 14
1.1.2 Architectures reconfigurables 14
1.1.3 Architectures reconfigurables dynamiquement 15
1.1.3.1 Reconfiguration par colonne 15
1.1.3.2 Reconfiguration par région 16
1.1.4 Architectures reconfigurables hétérogènes 18
1.2 Hypothèses matérielles et logicielles de nos travaux 18
1.2.1 Préemption de programmes/tâches 18
1.2.1.1 Définition 18
1.2.1.2 La préemption sur une architecture reconfigurable . . . 19
1.2.2 *Relocation* 20
1.3 Ordonnancement de tâches 23
1.3.1 Définition générale 23
1.3.1.1 Hors-ligne versus en-ligne 23
1.3.1.2 Préemptif versus non-préemptif 24
1.3.2 Ordonnancement temps réel 24
1.3.2.1 Tâche temps réel 25
1.3.2.2 Implémentation des systèmes temps réel 26
1.3.2.3 Ordonnancement temps réel multiprocesseur 26
1.3.3 Ordonnancement pour architectures reconfigurables 27
1.4 Placement de tâches sur architectures reconfigurables 28
1.4.1 Présentation de l'ordonnancement spatial 28
1.4.2 Travaux relatifs à l'ordonnancement spatial 28
1.4.2.1 Placement par colonne 29
1.4.2.2 Approches utilisant la notion de *rectangle vide maximum* 30
1.4.2.3 Réduction de la fragmentation 31
1.4.2.4 Placement hétérogène 33
1.5 Synthèse 34

2 Réseau de neurones de Hopfield **35**
2.1 Réseau de neurones artificiels 35
2.1.1 Neurone biologique 35
2.1.2 Neurone formel 36
2.1.3 Perceptron, réseau multicouche et autres 38
2.2 Réseau de Hopfield : généralités 39
2.2.1 Présentation 39
2.2.2 Fonctionnement 39
2.3 Convergence des réseaux de Hopfield 42
2.3.1 Modes d'évaluation d'un réseau de Hopfield 42
2.3.1.1 Mode séquentiel 42
2.3.1.2 Mode synchrone 42
2.3.1.3 Mode parallèle 43
2.3.2 Fonction d'énergie 43
2.4 Réseau de Hopfield pour l'optimisation 46
2.4.1 Définition d'un codage du problème 46
2.4.2 Construction d'un réseau de Hopfield associé à un problème d'optimisation 46
2.4.3 Utilisation d'un réseau associé à un problème d'optimisation . . 47
2.5 Règles génériques de construction d'un réseau 48
2.5.1 Règle k-de-n 48
2.5.2 Règle au-plus-k-de-n 50
2.5.3 Règle 0-ou-1-de-n 51
2.5.4 Règle de k-consécutivité 51
2.6 Propriété d'additivité des réseaux de Hopfield 53
2.7 Conclusion 54

3 Ordonnancement temporel par réseaux de neurones de Hopfield **55**
3.1 Modélisation neuronale d'un problème d'ordonnancement 55
3.1.1 Architecture homogène 56
3.1.2 Architecture hétérogène 57
3.2 Simplification du réseau : neurones inhibiteurs 59
3.2.1 Définition d'un neurone inhibiteur 60
3.2.2 Application à l'ordonnancement 61
3.2.3 Convergence d'un réseau pourvu de neurones inhibiteurs 62
3.2.4 Réduction du nombre de neurones cachés 64
3.2.4.1 Réduction par les neurones inhibiteurs 64
3.2.4.2 Suppression des tâches fictives 64
3.3 Résultats 66
3.3.1 Comparaison avec des approches neuronales classiques 66
3.3.2 Comparaison avec l'algorithme Pfair 67
3.3.2.1 Résultats préliminaires 69
3.3.2.2 Comparaison avec des jeux de tâches aléatoirement générés 70

3.3.3 Résultats sur une architecture hétérogène 71
3.3.4 Résultats dans le contexte des reconfigurables 74
3.3.5 Complexité 75
3.4 Conclusion 77

4 Placement de tâches pour architectures reconfigurables hétérogènes 79
4.1 Placement d'un ensemble de tâches par un réseau de Hopfield 79
4.1.1 Modélisation d'une architecture reconfigurable hétérogène 80
4.1.2 Modélisation d'une application 81
4.1.3 Modélisation du problème 82
4.1.3.1 Contrôle du nombre d'instances activées 84
4.1.3.2 Éviter le chevauchement d'instances de tâches 85
4.1.3.3 Augmenter le nombre d'instances placées 86
4.1.3.4 Fusion des différentes fonctions 87
4.1.4 Modification de la dynamique du réseau 88
4.1.4.1 Choix d'une fonction **s** 91
4.1.4.2 Seuillage dynamique et nombre d'instances activées. . . 93
4.1.4.3 Implémentation du seuillage dynamique. 93
4.1.5 Résultats 93
4.1.5.1 Simulations du placement d'un ensemble de tâches . . . 94
4.1.5.2 Simulations sur plusieurs jeux de tâches ayant des caractéristiques identiques 94
4.1.5.3 Comparaisons avec l'algorithme *SUP Fit* 95
4.1.5.4 Comparaison en temps d'exécution 97
4.1.6 Utilisation du réseau de neurones dans un contexte réel 98
4.1.6.1 Placement d'un sous ensemble de tâches 98
4.1.6.2 Gestion de l'état de la zone reconfigurable 98
4.1.7 Conclusion 99
4.2 Placement en colonne 100
4.2.1 Organisation des ressources d'une architecture reconfigurable hétérogène 101
4.2.2 Modélisation d'une tâche 102
4.2.3 Algorithme de placement 104
4.2.3.1 Placement d'une tâche 104
4.2.3.2 Suppression d'une tâche 106
4.2.4 Expérimentations 107
4.2.4.1 Scénario d'exécution 107
4.2.4.2 Comparaison du nombre de tâches rejetées 108
4.2.4.3 Comparaison du nombre d'instances réelles 109
4.2.5 Conclusion 111

5 Optimisation et tolérance aux fautes des réseaux de neurones de Hopfield 113
5.1 Évaluation parallèle d'un réseau de neurones de Hopfield 113
5.1.1 Mode d'évaluation parallèle 114
5.1.1.1 Sur un exemple simple 114
5.1.1.2 Généralisation 116
5.1.2 Convergence dans le cas parallèle 117
5.1.2.1 Signe du produit $A_1 \times A_2$ de l'équation (5.17) 118
5.1.2.2 Signe du produit $B_1^T \times \mathbf{W}_{11} \times B_1$ de l'équation (5.17) . 118
5.1.3 Construction des paquets 119
5.1.3.1 Application à un problème d'ordonnancement 120
5.1.3.2 Construction des paquets : généralisation 120
5.1.4 Expérimentation 122
5.1.5 Conclusion 125
5.1.6 Perspectives 125
5.2 Tolérance aux fautes 127
5.2.1 Modèle de fautes au sein d'un réseau de neurones de Hopfield . . 127
5.2.2 Maintien de la convergence d'un réseau de Hopfield malgré les défaillances 128
5.2.3 Résultats 130
5.2.3.1 Nombre de fautes tolérées 130
5.2.3.2 Impact du nombre de fautes sur le temps de convergence 131
5.2.4 Conclusion 131

Conclusion 133

Glossaire 139

Bibliographie 141

Table des figures 149

Résumé 157

Introduction

Contexte

L'évolution constante des applications et des systèmes embarqués engendre le développement de nouvelles architectures matérielles. Les systèmes embarqués étant soumis à de fortes contraintes de surface et d'énergie, ils doivent être capables de traiter très efficacement les calculs qui leur sont affectés. Alors que dans un ordinateur classique, la majeure partie des traitements est effectuée par un ou plusieurs processeurs généralistes, les systèmes embarqués utilisent différentes ressources, chacune étant spécialisée pour un certain type de requêtes. Désormais, il est donc courant de rencontrer au sein de la même puce, des blocs d'opérateurs DSP (*Digital Signal Processing*), des accélérateurs matériels dédiés ou reconfigurables, des mémoires, ainsi que des processeurs généralistes. Nous parlons alors de « systèmes sur puce », également nommés SOC (*System On Chip*).

Le caractère hétérogène de ce type de systèmes permet d'utiliser des ressources adaptées à chaque type de requêtes. Lorsque des traitements lourds sont requis (par exemple un codage vidéo) des blocs matériels dédiés peuvent être utilisés tandis que le processeur généraliste est utilisé pour des traitements plus légers. Contrairement aux blocs matériels dédiés, le processeur généraliste présente l'avantage de pouvoir être programmé facilement rendant ainsi le système évolutif. Nous voyons donc apparaître une rivalité entre performance et évolutivité. Dans une certaine mesure, en proposant des instructions spécialisées performantes, les processeurs spécialisés, tels que les DSP, sont un compromis. En effet, ils permettent un traitement efficace de certaines requêtes, tout en conservant l'aspect programmable du processeur. Ils restent cependant dédiés à un type de traitements. Il serait préférable que le système puisse utiliser ses ressources pour réaliser des traitements fortement différents : on parle alors de *flexibilité*.

En termes de flexibilité, le vide architectural existant entre les processeurs et les circuits dédiés a été comblé par l'introduction d'architectures reconfigurables telles que les FPGA (*Field Programmable Gate Array*). Originellement, ces architectures ont été utilisées afin de prototyper rapidement des circuits dédiés. Elles sont constituées d'une matrice d'unités de traitement interconnectées par un réseau de communication flexible. Les unités de traitement peuvent être configurées en vue de réaliser une fonction logique simple. Elles sont reliées en configurant le réseau d'interconnexion de manière à réaliser une fonction logique plus complexe. Elles offrent ainsi des performances très proches d'un circuit dédié tout en étant flexibles. Cette flexibilité a encore été accrue

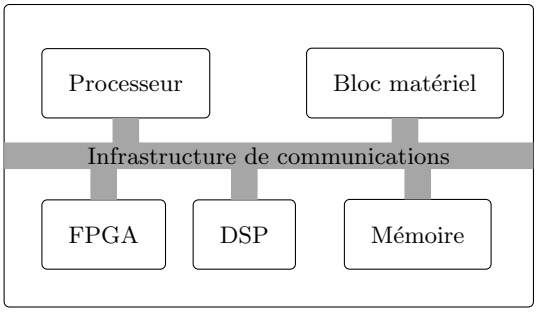


FIGURE 1 – Exemple de système sur puce composé de cinq ressources différentes ainsi que d'une infrastructure de communication.

par la technologie de reconfiguration dynamique qui permet de configurer une partie de l'architecture alors que des applications sont en cours d'exécution sur d'autres parties. En plus des unités de traitement logique nommées LUT (*Lookup Table*), les générations actuelles de FPGA intègrent également des unités de mémorisation et des DSP. À l'instar des SOC, l'introduction de ressources de différents types confère également aux FPGA un caractère hétérogène. La figure 1 présente le type d'architecture que nous considérons dans ce document. Cet exemple est un système sur puce composé de cinq ressources dont un FPGA.

Nous avons décrit le type d'architectures considéré dans nos travaux. Ces architectures ayant vocation à exécuter des applications, il convient également d'en définir la nature. Ainsi, dans nos travaux, nous considérons des applications dynamiques ayant des contraintes temps réel. Une application est composée d'un ensemble de tâches, chacune de ces tâches réalisant un traitement spécifique. En vertu des contraintes temps réel, l'exécution des tâches doit respecter des délais imposés par le programmeur. De plus, le caractère dynamique des applications considérées implique que l'ordre d'exécution des tâches n'est pas connu avant l'exécution de l'application.

En résumé, nous considérons l'exécution d'applications dynamiques et temps réel sur des systèmes embarqués hétérogènes potentiellement pourvus d'une ressource de type FPGA.

Problématique et contributions

Lors de l'exécution d'une application sur un système sur puce, tel que celui présenté par la figure 1, il faut déterminer quelles ressources vont être utilisées et à quel moment. Cette problématique est connue sous le terme d'*ordonnancement*. L'ordonnancement d'une application sur une architecture composée de plusieurs ressources comprend alors deux dimensions :

– une dimension temporelle déterminant les instants auxquels sont exécutées les tâches, de manière à satisfaire les contraintes temps réel de l'application et

– une dimension spatiale (ou *placement*) déterminant les ressources exécutant ces tâches.

Dans le cadre d'un système hétérogène, l'ordonnanceur temporel doit gérer des tâches n'ayant pas des temps d'exécution identiques sur l'ensemble des ressources. Il s'agit donc de déterminer les ressources accueillant les tâches tout en respectant les contraintes temporelles. Nous voyons donc que ces deux dimensions sont fortement liées et qu'il paraît difficile de les traiter indépendamment. Ainsi, nous proposons un ordonnanceur pour SOC hétérogènes traitant ces deux dimensions simultanément.

L'ordonnancement spatial (ou placement) pour architectures reconfigurables de type FPGA s'avère quant à lui bien plus complexe. Alors que pour un SOC, l'ordonnancement spatial consiste à déterminer une ressource pour l'exécution d'une tâche, dans le cadre d'un FPGA, il est nécessaire de déterminer une zone accueillant une tâche. La figure 2 présente un exemple de modélisation d'une architecture reconfigurable composée de 25 ressources (cinq DSP et 20 LUT). Nous voyons également que deux tâches (τ_1 et τ_2) sont en cours d'exécution. Pour placer ces tâches, l'ordonnanceur leur a attribué un emplacement. Le choix de cet emplacement peut être effectué selon divers critères. L'un des critères les plus courants vise à favoriser le placement de futures tâches. Dans ce cas, le placement de tâches revient à placer un maximum de rectangles au sein d'un rectangle, ce qui peut être assimilé au problème *Bin Packing*, qui est prouvé NP Complet.

En fait, le placement de tâches sur une architecture reconfigurable moderne s'avère quelque peu différent. La présence de ressources de natures différentes restreint le placement des tâches à un sous ensemble des positions. Par exemple, sur la figure 2, la tâche τ_1 est placée à la position $(0, 3)$. En termes de ressources, elle pourrait également être placée à la position $(0, 2)$, tandis que la position $(1, 3)$ ne conviendrait pas car les ressources dont disposerait la tâche τ_1 ne seraient pas alignées avec les ressources requises. Nous voyons donc que le caractère hétérogène de l'architecture complexifie les opérations de placement de tâches.

De nombreux auteurs proposent des algorithmes de placement pour des architectures reconfigurables homogènes alors que la majorité des FPGA fabriqués sont hétérogènes. Leurs algorithmes s'avèrent donc inadaptés. Dans cette thèse, nous proposons des algorithmes de placement capables de gérer le caractère hétérogène de l'architecture.

Lors de la présentation du contexte de nos travaux, nous avons mentionné le dynamisme des applications. Ce dynamisme implique la nécessité d'effectuer l'ordonnancement en-ligne, c'est-à-dire, durant l'exécution de l'application. Ainsi, l'exécution des algorithmes d'ordonnancement doit être la plus efficace possible afin de ne pas pénaliser l'exécution de l'application. Dans cette thèse, nous proposons une résolution originale des problèmes d'optimisation par l'utilisation de réseaux de neurones de Hopfield.

Les réseaux de neurones de Hopfield sont des algorithmes inspirés du fonctionnement du cerveau humain. Un réseau de neurones de Hopfield est composé d'un ensemble de neurones, qui sont tous reliés les uns aux autres par des connexions valuées. Initialement, ce type de réseau a été développé afin de réaliser des mémoires associatives. Ensuite,

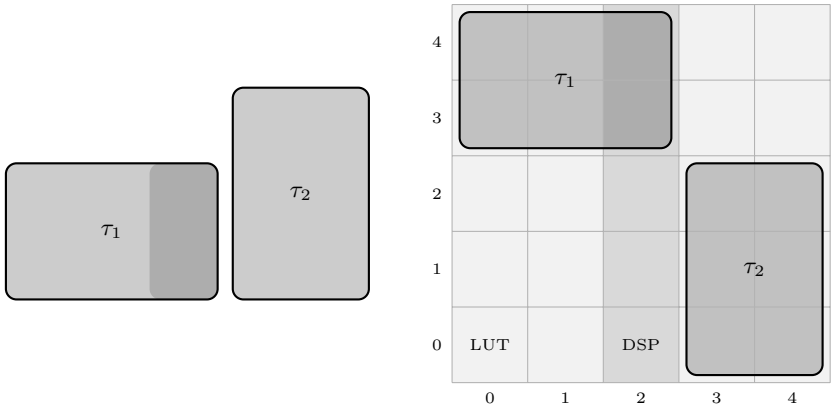


FIGURE 2 – Exemple de placement de deux tâches sur une architecture reconfigurable hétérogène de type FPGA. La tâche τ_1 requiert quatre LUT et deux DSP, la tâche τ_2 six LUT.

la propriété de convergence de ces réseaux a été utilisée pour résoudre des problèmes d'optimisation, tels que des problèmes d'ordonnancement.

L'évaluation d'un réseau de neurones de Hopfield est effectuée en évaluant successivement tous les neurones formant le réseau. L'évaluation d'un neurone consistant en une opération de multiplication-accumulation, il est possible d'obtenir une implémentation matérielle efficace. Cette caractéristique s'avère très intéressante dans notre contexte d'ordonnancement en-ligne sur une architecture reconfigurable. Afin de réduire au maximum le temps d'exécution de l'ordonnanceur, nous proposons également d'améliorer le temps d'évaluation d'un réseau de Hopfield en évaluant simultanément plusieurs neurones.

Dans cette thèse, nous traitons donc trois problématiques principales :

- l'ordonnancement temporel pour architectures hétérogènes,
- l'ordonnancement spatial pour architectures hétérogènes de type FPGA,
- l'exécution rapide des algorithmes d'ordonnancement à travers l'utilisation de réseaux de neurones de Hopfield.

Afin de répondre à ces problématiques, nous proposons dans ce manuscrit

- un ordonnanceur temporel pour architectures hétérogènes basé sur un réseau de Hopfield [13],
- un ordonnanceur spatial pour architectures hétérogènes reconfigurables également basé sur un réseau de Hopfield [1],
- un modèle de tâches permettant d'augmenter les emplacements utilisables par une tâche matérielle,
- une méthode d'évaluation parallèle de neurones réduisant le nombre d'évaluations de neurones d'un réseau de Hopfield [23],
- ainsi qu'une étude portant sur la tolérance aux fautes des réseaux de Hopfield.

Plan du mémoire

Ce document de thèse est articulé en cinq chapitres. Les deux premiers chapitres introduisent les notions nécessaires à la compréhension de nos travaux et les trois derniers chapitres présentent nos contributions et résultats.

Dans le premier chapitre, nous introduisons le contexte matériel sur lequel sont fondés nos travaux. Dans une première partie, nous présentons les architectures reconfigurables dynamiquement. Nous présentons ensuite l'ordonnancement temporel puis spatial. Concernant l'ordonnancement spatial, nous exposons les principaux travaux de placement de tâches sur une architecture reconfigurable.

Le second chapitre présente les réseaux de neurones de Hopfield. Parce que la majeure partie de nos travaux utilise de tels réseaux, il est indispensable d'en comprendre le fonctionnement en détail. Nous exposons donc leurs propriétés mathématiques ainsi que des techniques facilitant leur utilisation.

Le troisième chapitre présente un ordonnanceur pour architectures hétérogènes. Cet ordonnanceur est basé sur un réseau de neurones de Hopfield dont nous avons modifié le comportement afin d'améliorer son implémentation. Nous avons ainsi introduit les neurones inhibiteurs qui permettent de réduire de façon importante le nombre de neurones nécessaire à la résolution du problème. Cet ordonnanceur est comparé à l'algorithme d'ordonnancement multiprocesseur Pfair afin d'évaluer sa qualité.

Dans le quatrième chapitre, nous exposons nos travaux relatifs à l'ordonnancement spatial pour une architecture reconfigurable hétérogène de type FPGA. Nous présentons deux ordonnanceurs qui sont chacun basés sur des hypothèses matérielles différentes. Le premier ordonnanceur ne requiert aucun mécanisme de placement complexe. Il est donc utilisable avec les outils fournis par les fabricants de circuits. Le second ordonnanceur est plus avant-gardiste à cause des mécanismes de placement qu'il nécessite : nous considérons alors disposer d'un outil permettant de placer une tâche synthétisée en plusieurs endroits de la zone reconfigurable.

Finalement, le dernier chapitre expose des travaux relatifs aux réseaux de neurones de Hopfield. Dans une première partie, nous présentons une méthode d'évaluation parallèle d'un réseau de neurones de Hopfield permettant d'accélérer considérablement le temps d'obtention d'une solution. Une seconde partie décrit des propriétés de tolérance aux fautes de ce type de réseaux. Cette technique permet de définir des critères garantissant une évaluation correcte du réseau malgré l'apparition de fautes, telles que des collages de transistors, au sein de l'architecture exécutant le réseau de neurones.

Chapitre 1

Ordonnancement spatio-temporel pour architectures reconfigurables

Dans ce premier chapitre, nous exposons les notions nécessaires à la compréhension de nos travaux ainsi que des travaux connexes aux nôtres. Nous en profitons également pour poser les principales hypothèses sur lesquelles sont basés nos recherches et résultats.

La première partie de ce chapitre présente les architectures reconfigurables. Parce que ces architectures sont en perpétuelle évolution, il s'avère nécessaire de définir quels fonctionnalités et mécanismes nous considérons. Ainsi, la deuxième section décrit les hypothèses matérielles et logicielles communes à l'ensemble de nos travaux. Le reste du chapitre présente l'ordonnancement de tâches en ciblant les architectures hétérogènes reconfigurables dynamiquement. Ainsi, la troisième section présente l'ordonnancement temporel en indiquant les différentes composantes de cette problématique bien connue. Finalement, la dernière section présente le placement de tâches appliqué aux architectures reconfigurables. Après avoir introduit cette problématique, nous exposons les principaux travaux relatifs au placement de tâches sur ce type de composants.

1.1 Architectures reconfigurables

Cette section présente les architectures reconfigurables de manière chronologique. En commençant par les prémisses de ce type de composants, leurs différentes fonctionnalités et spécificités sont introduites au fur et à mesure pour finalement présenter les architectures hétérogènes reconfigurables dynamiquement qui correspondent aux circuits considérés dans nos travaux.

1.1.1 Architectures configurables

Les architectures configurables sont des architectures pouvant être adaptées à un algorithme précis, de manière flexible mais définitive. La première architecture de ce type est le PAL (*Programmable Array Logic*) présentée par *Monolithic Memories* en 1978. Ce type d'architecture offre un compromis entre un circuit dédié et un processeur. En effet, elle apporte une certaine flexibilité par rapport aux circuits dédiés et des performances accrues par rapport aux processeurs généralistes. Cependant, les architectures PAL présentent l'inconvénient de n'être configurables qu'une seule fois.

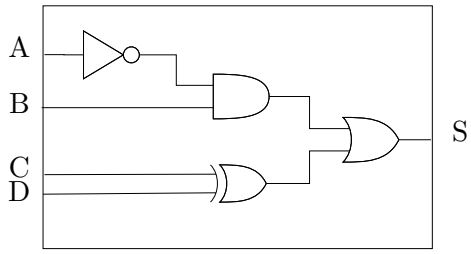
1.1.2 Architectures reconfigurables

Afin de pallier à cette contrainte de configuration définitive des architectures PAL, les fabricants ont développé de nouvelles technologies. Ainsi, *LATTICE* présente, en 1985, la technologie GAL (*Generic Array Logic*) permettant d'effectuer une configuration de l'architecture à chaque initialisation du circuit. La même année, la société *Xilinx* présente également un nouveau type d'architectures reconfigurables [16], les FPGA. Ce type d'architecture domine désormais le marché des architectures reconfigurables.

Les FPGA sont réalisés par un assemblage de trois fonctions logiques : des LUT, des bascules, et des multiplexeurs. Une LUT est une table à n variables où sont stockées les 2^n solutions possibles d'une équation logique à n entrées. En guise d'exemple, la figure 1.1 présente un schéma et la table de vérité de la fonction logique $(\bar{A} \cdot B) + (C \oplus D)$. Pour implémenter cette fonction logique, une LUT est configurée de manière à mémoriser la table de vérité de cette fonction logique. Les entrées de la LUT correspondent alors aux quatre variables A, B, C et D et permettent d'adresser les 2^4 solutions possibles.

Les LUT sont interconnectées par l'intermédiaire de multiplexeurs. Un multiplexeur est une ressource de connexion composée de n entrées et une sortie. Il est possible de connecter dynamiquement une entrée à la sortie au moyen d'un mot de configuration de taille $t = \lceil log_2(n) \rceil$.

Le réseau de LUT ainsi formé constitue un circuit combinatoire qui est rendu synchrone en plaçant des bascules en sortie des LUT.



A	B	C	D	S
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
⋮	⋮	⋮	⋮	⋮
1	1	1	0	1
1	1	1	1	0

FIGURE 1.1 – Schéma et table de vérité de la fonction logique $(\bar{A} \cdot B) + (C \oplus D)$.

Généralement, les fabricants regroupent plusieurs LUT, des multiplexeurs et des bascules dans un bloc appelé CLB (*Configurable Logic Block*). Ces CLB constituent les

éléments de base d'un FPGA, et la reconfiguration consiste alors à configurer ces CLB. Un réseau de communication permet de réaliser des connexions entre différents CLB.

Remarque. les FPGA sont des architectures reconfigurables dites à grains fins car la reconfiguration s'effectue au niveau bit. Il existe d'autres architectures reconfigurables, dites à grains épais travaillant sur des mots de plusieurs bits, tel que le processeur reconfigurable DART [52]. Une architecture à grains fins permet de spécialiser très finement l'architecture à l'algorithme ciblé, contrairement aux architectures à grains épais qui imposent, par exemple, une taille de mot fixe. En revanche, les architectures à grains épais bénéficient de données de configuration et de temps de configuration nettement moins importants.

1.1.3 Architectures reconfigurables dynamiquement

Nous avons vu que les architectures reconfigurables offrent la possibilité d'implémenter un algorithme différent à chaque initialisation. Par rapport à un processeur généraliste, les performances d'exécution sont nettement supérieures. Cependant, les FPGA ont des lacunes en termes de flexibilité à cause de la nécessité de redémarrer le circuit. Les processeurs généralistes sont pourvus d'une flexibilité bien plus importante. En effet, même sur un mono-processeur, il est possible d'exécuter plusieurs programmes simultanément, par multiplexage temporel ou préemption, mais surtout, un programme peut être modifié alors que d'autres programmes sont en cours d'exécution. Nous pouvons imaginer exécuter plusieurs algorithmes simultanément sur une architecture reconfigurable, en synthétisant tous ces algorithmes simultanément, cependant, il est impossible de modifier un de ces algorithmes sans ré-effectuer une synthèse globale, et donc un redémarrage du circuit.

À la fin des années 90, les fabricants ont commencé à concevoir des architectures reconfigurables [31, 4] dont des parties peuvent être reconfigurées alors que simultanément, les autres parties de la zone reconfigurable continuent de fonctionner normalement. Cette évolution technologique donna naissance aux architectures reconfigurables dynamiquement.

Actuellement, *Xilinx*, à travers sa gamme *Virtex*, propose les FPGA reconfigurables dynamiquement les plus aboutis du marché. En effet, l'architecture ainsi que les outils fournis par le fabricant permettent d'exploiter assez simplement la reconfiguration dynamique. Nos travaux se focalisent sur ce type d'architectures dont le fonctionnement est détaillé dans la suite de cette section.

1.1.3.1 Reconfiguration par colonne

Les premier FPGA *Xilinx* [1] étaient reconfigurables dynamiquement par colonne. Le FPGA était alors composé d'un ensemble de colonnes, qui pouvaient être reconfigurées séparément mais entièrement. La figure 1.8 représente un FPGA composé de quatre

1. *Virtex*, *Virtex*-II et *Virtex*-II Pro

colonnes. Un module M_1 est exécuté sur la colonne C_1. Il reste donc trois colonnes libres qui peuvent accueillir d'autres modules.

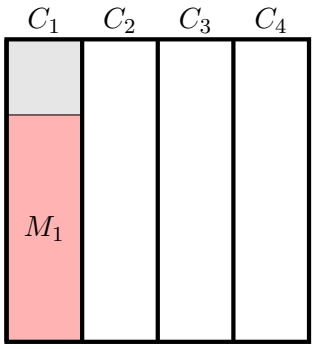


FIGURE 1.2 – FPGA reconfigurable dynamiquement par colonne. Un module M_1 est exécuté sur la première colonne.

Le module M_1 n'occupe pas toute la colonne du FPGA représenté par la figure 1.8. Comme les colonnes ne peuvent pas être reconfigurées partiellement, une partie de la colonne C_1 est perdue durant toute l'exécution du module M_1.

1.1.3.2 Reconfiguration par région

Afin de limiter les pertes de la reconfiguration par colonne, *Xilinx* a introduit la reconfiguration par région à partir des *Virtex*-4 [67].

La figure 1.3 illustre l'exécution de deux modules sur un FPGA reconfigurable par région. La sous figure 1.3.a présente le découpage en régions choisi. À l'initialisation du circuit, un découpage plus ou moins arbitraire est défini par l'utilisateur. La figure 1.3.b illustre l'exécution de deux modules. Chaque module occupe une région entière, laissant les autres régions exploitables ultérieurement par d'autres modules.

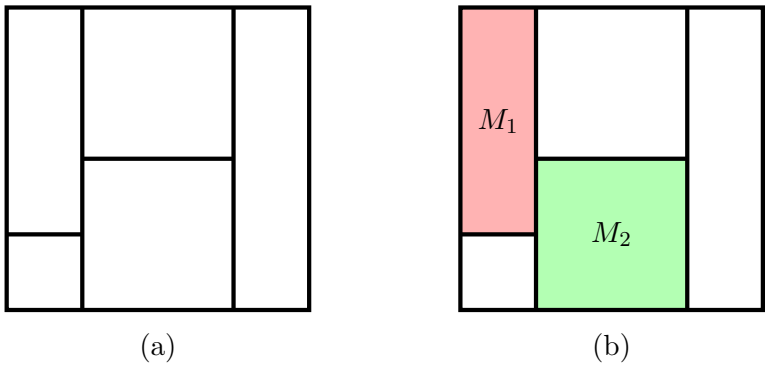


FIGURE 1.3 – Reconfiguration par région. La sous figure (a) présente un FPGA composé de cinq régions. La sous figure (b) illustre l'exécution de deux modules sur ce FPGA.

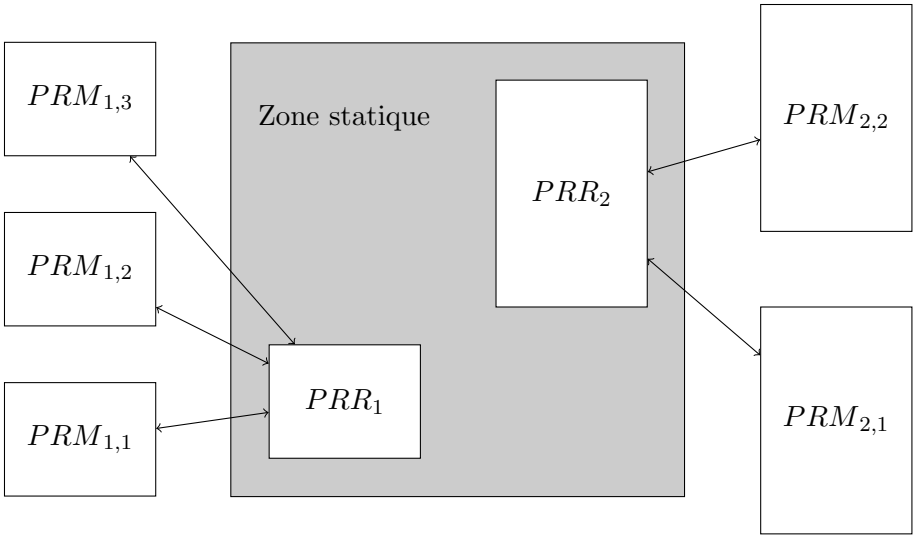


FIGURE 1.4 – Illustration de la reconfiguration par région. Deux PRR ont été instanciés sur la zone reconfigurable. Trois PRM différents peuvent être accueillis par le PRR_1 et deux PRM par le PRR_2.

Dans nos travaux, nous considérons l'utilisation de la méthode de reconfiguration partielle *module based* [66] permettant de procéder à des reconfigurations de zones définies à l'initialisation du circuit. Dans un tel système, le FPGA est partitionné en une zone statique et en une ou plusieurs régions reconfigurables appelées PRR (*Partially Reconfigurable Region*).

La zone statique est une partie du circuit où la logique n'est pas modifiée durant l'utilisation du FPGA. Cette zone est configurée à l'initialisation du circuit, en téléchargeant un *bitstream* implémentant les mécanismes nécessaires à la reconfiguration partielle ainsi qu'aux communications des PRR.

Les PRR contiennent la logique pouvant être reconfigurée indépendamment de la zone statique et des autres PRR. Ainsi, un PRR peut accueillir successivement plusieurs PRM (*Partially Reconfigurable Module*). Les PRM peuvent être conçus et implémentés indépendamment de la configuration statique. Cependant, ils sont associés à un PRR et doivent respecter les contraintes de connexions relatives à ce PRR. La figure 1.4 présente un zone reconfigurable où deux PRR ont été instanciés. Dans cet exemple, trois PRM ont été synthétisés pour le PRR_1 et deux PRM pour le PRR_2.

Un *bitstream* partiel est généré pour chaque PRM. En ayant une description précise d'un PRR, il est ainsi possible de synthétiser et d'exécuter des modules sur ce PRR alors que d'autres modules sont en cours d'exécution sur le FPGA. Ces *bitstream* sont chargés sur le FPGA via le port de reconfiguration ICAP (*Internal Access Configuration Port*) [68].

Finalement, un dernier élément dans le processus de reconfiguration dynamique des FPGA *Virtex* est le *Bus Macro* [66]. Le *Bus Macro* permet aux PRM de communiquer

avec la logique statique, assurant ainsi la communication des PRM avec le monde extérieur.

1.1.4 Architectures reconfigurables hétérogènes

Dans la section précédente, nous avons présenté les unités fondamentales d'un FPGA, à savoir, les CLB. Ces unités permettent de réaliser des fonctions logiques, mais peuvent également être utilisées en guise de mémoires. Cependant, les fabricants intègrent désormais des ressources optimisées pour la mémorisation, les BRam (*Block RAM*). Ces blocs mémoire sont répartis sur l'ensemble de la zone reconfigurable afin que l'ensemble des CLB puisse utiliser des BRam relativement proches.

Les FPGA issus de la famille *Virtex* sont également pourvus de blocs DSP, ainsi que de processeurs généralistes. Par exemple, certains modèles issus de la famille *Virtex* possèdent deux cœurs de processeurs *Power PC* au centre de la zone reconfigurable.

L'introduction de ressources de types différents au cœur de la zone reconfigurable lui fournit un caractère hétérogène dont il faut tenir compte lors de l'utilisation de ce type d'architecture.

1.2 Hypothèses matérielles et logicielles de nos travaux

1.2.1 Préemption de programmes/tâches

1.2.1.1 Définition

Initialement, les processeurs généralistes étaient utilisés dans des *mainframes* (ordinateurs centraux), réservés à des grandes firmes. L'opérateur avait alors la possibilité de soumettre un *job* (programme ou ensemble de programmes) à ces machines. Le processeur était alors monopolisé par ce *job* durant tout le temps requis à son exécution. Ce mode opératoire a immédiatement montré des limites notamment en terme de performances. Lorsqu'un programme en cours d'exécution utilisait des données provenant d'un périphérique lent (un disque dur par exemple), des cycles processeurs étaient perdus à attendre les données provenant de ce périphérique.

Afin d'exploiter pleinement la puissance de calcul d'un processeur, la multiprogrammation a été introduite. Cette technique permet de partager le temps processeur entre plusieurs programmes. Ainsi, lorsqu'un programme est en attente de données, le système peut exécuter un autre programme en attendant que les données soient prêtes.

La multiprogrammation requiert un mécanisme permettant de suspendre des programmes en cours d'exécution puis d'en reprendre l'exécution ultérieurement. Le mécanisme généralement utilisé est la préemption. La préemption consiste à sauvegarder le contexte d'exécution d'un programme, capturant ainsi l'état du programme au moment où il est préempté. Lorsque le programme est exécuté sur un processeur généraliste, ce contexte est principalement décrit par la valeur des registres ainsi que la valeur du compteur ordinal. Lorsqu'un programme est préempté, le système d'exploitation stoppe son exécution puis sauvegarde ces valeurs. Pour poursuivre l'exécution de ce programme, le contexte est alors restauré et l'exécution reprend là où elle avait été stoppée.

Bien qu'initialement créée afin d'améliorer le rendement d'un processeur, la multiprogrammation est devenue essentielle à l'utilisation d'un ordinateur moderne. Cette technique permet de donner l'illusion à l'utilisateur que plusieurs programmes sont exécutés simultanément. De même, les systèmes d'exploitation multi-utilisateurs exploitent pleinement la multiprogrammation.

Telle que décrite précédemment, cette technique peut être réalisée logiciellement, au prix d'une importante perte de performance, d'autant plus lorsque des politiques de sécurité sont appliquées. Afin d'améliorer les pénalités induites par la multiprogrammation, les fabricants ont introduit des mécanismes matériels, tels que la MMU (*Memory Management Unit*) par exemple.

1.2.1.2 La préemption sur une architecture reconfigurable

La multiprogrammation ayant permis d'améliorer le rendement des processeurs généralistes, il est naturel de souhaiter appliquer cette technique aux architectures reconfigurables dynamiquement. Comme il est possible de reconfigurer une partie de l'architecture sans altérer les exécutions en cours sur d'autres parties, il est envisageable de partager une partie entre plusieurs modules en multiplexant temporellement ces modules. Il est donc nécessaire d'avoir un mécanisme de préemption et donc de sauvegarde et restauration de contexte. Or, il est beaucoup plus complexe d'obtenir le contexte d'une application en cours d'exécution sur une architecture reconfigurable que sur un processeur généraliste. En effet, il serait nécessaire de sauvegarder l'état interne des données, or aucun mécanisme permettant un accès direct à ces données n'est actuellement fourni par les fabricants.

Néanmoins, il existe des travaux visant à fournir un service de sauvegarde et restauration de contexte sur FPGA. Bien que cela ne soit pas nécessaire à l'appréhension de nos travaux, nous exposons brièvement différentes stratégies permettant de réaliser un tel service. Une première méthode consiste à utiliser la fonctionnalité de *read-back* présente sur certain FPGA [42]. Cette fonctionnalité permet de relire le *bitstream* et d'obtenir l'état des registres. Le principal inconvénient de cette méthode est la quantité de données d'une sauvegarde car elle contient toutes les informations de configuration et pas uniquement l'état des registres.

Une seconde approche consiste à imposer au concepteur de respecter une interface exposant explicitement le contexte du module [19]. L'inconvénient majeur de cette approche est qu'elle complexifie grandement la tâche du concepteur qui doit gérer le contexte de son application.

Finalement, une autre approche consiste à modifier les bascules du *design* de manière à pouvoir obtenir leur état [27, 26]. Cette technique, nommée *scanpath*, est couramment utilisée dans le domaine du test de circuit. La logique ajoutée à chaque bascule induit une augmentation de la surface requise par le *design* et peut également réduire sa fréquence de fonctionnement maximale.

Nous avons présenté quelques techniques visant à introduire la préemption de tâches matérielles sur des architectures reconfigurables. Cependant, ces techniques sont encore

en cours d'étude et ne sont pas supportées par les fabricants. Dans nos travaux, nous ne considérons donc pas de préemption, ce qui a une implication directe sur les algorithmes d'ordonnancement.

1.2.2 *Relocation*

Lors de la présentation des architectures reconfigurables par région, il a été précisé qu'un PRM était associé à un PRR. Cette contrainte impose donc que l'exécution d'un PRM ait lieu sur un PRR en particulier. Ainsi, pour exécuter un module sur n PRR, il est nécessaire de générer n *bitstream*s puis de choisir à l'exécution le *bitstream* correspondant au PRR ciblé.

Le mécanisme de relocation [2] vise à offrir la possibilité d'exécuter sur plusieurs PRR un module synthétisé pour un seul PRR.

Relocation par colonne. Depuis plus de 10 ans, des travaux visant à reloger un *bitstream* sont menés. L'un des premiers résultats significatifs est l'outil PARBIT [36] permettant de générer un *bitstream* partiel à partir du *bitstream* complet d'un FPGA reconfigurable par colonne. Cette génération est effectuée logiciellement en amont du contrôleur de reconfiguration. Cependant, la génération de *bitstream* partiel est effectuée sur un processeur externe, ce qui compromet l'utilisation de PARBIT dans un contexte en-ligne.

Afin de réduire le temps de génération d'un *bitstream*, REPLICA [37, 38] agit comme un filtre, implémenté matériellement et manipulant le *bitstream* lors du processus de téléchargement. REPLICA supporte également un FPGA reconfigurable par colonne. Il permet donc d'exécuter un module sur une colonne déterminée lors de l'exécution. Afin de maximiser l'utilisation du FPGA, les modules exécutés doivent occuper la zone reconfigurable dans toute sa hauteur.

Utilisation de plusieurs colonnes par un module. Les applications exécutées sur le FPGA pouvant être de natures différentes, la surface qu'elles requièrent peut être également différente. Afin d'accroître la flexibilité du système, un module doit pouvoir occuper plusieurs colonnes, sans quoi, tous les modules exécutés disposent de la même surface. Or l'exécution d'un module sur plusieurs colonnes engendre des problèmes de communications.

Dans la section 1.1.3.2, nous avons présenté le *Bus Macro*, réalisant les communications entre PRR. Dans le cas d'une architecture reconfigurable par colonne, le même type de mécanisme est mis en place afin de permettre à un module exécuté sur une colonne de communiquer avec d'autres modules. Lorsqu'un module est exécuté sur plusieurs colonnes, il s'avère alors nécessaire d'utiliser ce type de mécanisme au sein du module. Cependant, les *Bus Macro* requiert un adressage spécifique déterminé lors du placement-routage d'un module. Il s'avère donc inadaptés au processus de relocation. Ainsi, pour permettre l'occupation de plusieurs colonnes par un module, REPLICA

2. Dans la suite du mémoire, nous francisons le terme *relocation*.

s'appuie sur une infrastructure de communication homogène, détaillée dans [39, 28], permettant à un module de communiquer indépendamment des colonnes utilisées.

Relocation par région. L'évolution technologique des FPGA a engendré des recherches sur la relocation de modules dans le cadre d'une architecture supportant la reconfiguration par région.

BiRF (*Bitstream Reconfiguration Filter*) est un filtre de relocation comparable à REPLICA disposant de performances accrues [25]. Il a été étendu à la relocation dans le cadre d'une architecture supportant la reconfiguration par région [17]. En effet, les auteurs de [17] exposent des résultats de relocation par région sur *Virtex* 4 et *Virtex* 5. De plus, ils y présentent des informations intéressantes quant à la structure du *bitstream* et l'organisation des ressources d'un FPGA *Virtex* 5. Cependant, l'infrastructure de communication n'est pas abordée par ces travaux.

Dans [55], des travaux plus complets de relocation par région sont présentés. En effet, cet article évoque la manipulation du *bitstream* ainsi que l'infrastructure de communication nécessaire à la relocation. La figure 1.5 décrit brièvement le modèle d'exécution utilisé dans [55]. La région statique contient le contrôleur de reconfiguration (incluant les mécanismes de manipulation du *bitstream*), ainsi que le contrôleur de bus. Chaque module implémente une interface de communication lui permettant d'accéder au bus. Lors de la relocation d'un module, le contrôleur de reconfiguration doit donc veiller à ce que le module soit « aligné »avec l'un des bus disponibles.

Hétérogénéité et relocation. Toutes les techniques de relocation présentées précédemment ont pour point commun de déplacer un *bitstream* sur des zones identiques. Or, la majorité des FPGA actuels dispose de ressources hétérogènes. Ils s'avèrent donc nécessaire que la technique de relocation soit en mesure de supporter cette hétérogénéité.

Une technique triviale consiste à synthétiser plusieurs *bitstream* pour une même application de manière à supporter des emplacements offrant des ressources différentes. Ainsi, pour a applications et r régions différentes, il est nécessaire de synthétiser et stocker $a \times r$ *bitstream*s.

Afin de réduire la quantité de mémoire nécessaire au stockage des *bitstream*s, les auteurs de [8] proposent de déterminer les ressources communes entre plusieurs régions différentes, puis de synthétiser l'application considérée en ne tenant compte que de ces ressources communes. Cette technique réduit le nombre de *bitstream*s à stocker en sacrifiant des ressources, ce qui peut mener à une sous utilisation du FPGA.

La relocation dans nos travaux. La relocation d'un *bitstream* est une opération complexe et fortement dépendante de l'architecture sous-jacente, nécessitant une connaissance précise de celle-ci. Or, les fabricants ne communiquent que très peu sur le fonctionnement interne, principalement pour des raisons de propriété intellectuelle. Il s'avère donc particulièrement délicat de développer ce genre de techniques.

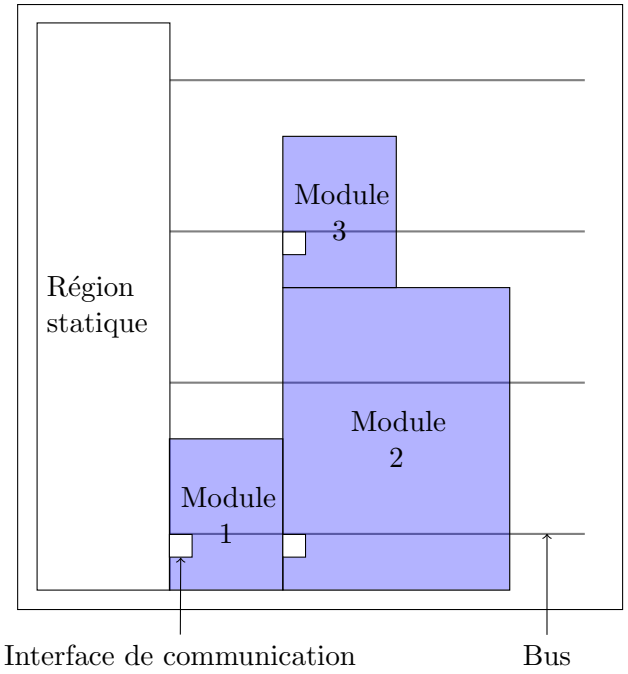


FIGURE 1.5 – Modèle d'exécution permettant la relocation de *bitstream* tel que présenté dans [55]. Trois modules sont en cours d'exécution et chacun de ces modules est pourvu d'une interface de communication. La région statique contient le contrôleur de reconfiguration (incluant les mécanismes de manipulation du *bitstream*), ainsi que le contrôleur de bus.

La majeure partie des travaux présentés dans cette thèse ne requiert pas de relocation afin d'être utilisable à travers le flot de conception fourni par les fabricants. Seuls les travaux exposés dans la section 4.2 exploitent la relocation, en la restreignant cependant au cas homogène.

1.3 Ordonnancement de tâches

1.3.1 Définition générale

L'ordonnancement de tâches consiste à définir un ordre d'exécution sur l'ensemble des tâches soumises à un système. Dans nos travaux, nous considérons une application composée d'un ensemble de tâches. Le flot d'exécution de l'application correspond alors à l'ordre d'exécution des tâches constituant l'application considérée. L'action d'ordonnancement est réalisé par un composant nommé ordonnanceur.

L'ordonnanceur est l'un des composants principaux d'un système d'exploitation [64]. Dans ce contexte, l'ordonnanceur choisit quel processus doit être exécuté à un instant donné. Afin de rester le plus généraliste possible, nous n'utilisons pas le vocabulaire spécifique aux systèmes d'exploitation. Ainsi, dans la suite de ce document, nous parlerons d'applications et de tâches.

En fonction des applications à ordonnancer et des architectures sous-jacentes, de nombreux types d'ordonnancements ont été proposés. En fonction de la nature de l'application à exécuter, l'ordonnancement peut être effectué avant son exécution (hors-ligne) ou pendant (en-ligne). Le matériel sur lequel est exécuté l'application peut également avoir une incidence sur les mécanismes utilisés par l'ordonnanceur. Ainsi, certaines architecture permettent de réaliser efficacement la préemption ou non.

1.3.1.1 Hors-ligne versus en-ligne

Lorsque le flot d'exécution de l'application est connu à l'avance, il est possible de prédéfinir l'ordonnancement des tâches composant cette application. Dans ce cas, l'ordonnancement est dit *hors-ligne*, la durée des temps de calcul engendrés par l'algorithme d'ordonnancement n'a pas réellement d'importance ce qui permet d'utiliser des algorithmes très sophistiqués. L'ordre d'exécution des tâches est alors fourni au système en même temps que les tâches. Ce type d'ordonnancement est généralement appliqué à des systèmes dédiés à une fonction bien particulière tels que des assistants à la conduite dans le domaine automobile par exemple [57].

Cependant, de nombreux systèmes ont à réagir à des événements extérieurs, provenant de l'environnement ou de l'utilisateur du système. Des actions sont alors effectuées en fonction de ces événements, il n'est plus possible de prédéfinir l'ordonnancement des tâches car le flot d'exécution de l'application n'est pas connu à l'avance. Dans ce contexte, les algorithmes d'ordonnancement sont exécutés *en-ligne* afin de déterminer quelle tâche doit être exécutée à un instant donné. Le temps d'exécution de l'ordonnanceur participant à la performance global du système, il convient d'utiliser

des algorithmes d'ordonnancement ayant un temps de réponse le plus bref possible, tout en consommant le moins de ressources possible.

1.3.1.2 Préemptif versus non-préemptif

La section 1.2.1 a présenté le concept de préemption d'un point de vue architectural et système. Nous rappelons simplement que la préemption permet une exécution fragmentée d'une tâche. L'ordonnanceur peut donc utiliser la préemption pour partager temporellement des ressources entre plusieurs tâches. Les algorithmes d'ordonnancements sont généralement conçus en fonction du support ou non de la préemption.

La figure 1.6 présente deux scénarios d'ordonnancement produits par deux algorithmes différents, à savoir, les algorithmes FIFO (*First In First Out*) et *Round Robin*. L'application considérée est composée de trois tâches (τ_1, τ_2 et τ_3) nécessitant chacune trois *slots* d'exécution [3]. La figure 1.6.a présente l'ordonnancement produit par l'algorithme FIFO non préemptif. Cet algorithme exécute les tâches entièrement en fonction de leur ordre d'arrivée. La figure 1.6.b présente l'ordonnancement produit par l'algorithme *Round Robin*. Cet algorithme exécute tour à tour les tâches en leur attribuant un *slot*. Il présente l'avantage de distribuer équitablement la ressource d'exécution entre les tâches mais nécessite le support de la préemption par l'architecture.

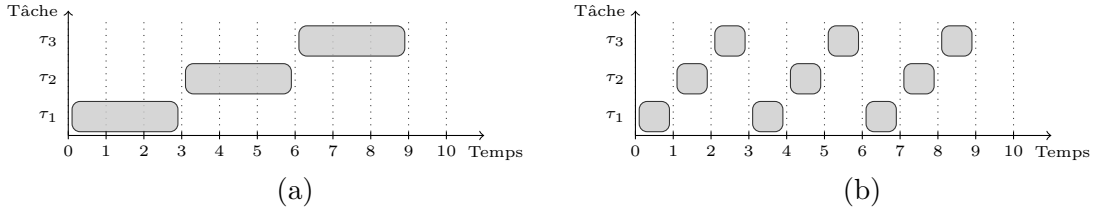


FIGURE 1.6 – Exemple d'ordonnancement de trois tâches sur une architecture supportant la préemption et sur une qui ne la supporte pas. La sous figure (a) présente un ordonnancement FIFO. La sous figure (b) présente un ordonnancement *Round Robin*.

Nous avons présenté une classification généraliste des algorithmes d'ordonnancement. Les choix relatifs à la nature de l'ordonnancement sont effectués en fonction des applications visées, notamment en ce qui concerne l'exécution hors-ligne ou en-ligne, ainsi que de l'architecture en ce qui concerne la prise en compte de la préemption. Dans la suite, nous présentons une autre caractéristique importante de l'ordonnancement, la gestion des applications ayant des contraintes temps réel.

1.3.2 Ordonnancement temps réel

De nombreuses applications sont soumises à des contraintes temporelles qui doivent être respectées par l'ordonnanceur. L'exécution de ces applications requiert un système

3. Temps d'exécution entre deux préemptions.

dit *temps réel*. La correction de l'application dépend du résultat produit mais également de son temps d'exécution.

Il convient de distinguer deux types d'applications différents. Les applications dont le non-respect des contraintes temporelles n'engendre qu'une perte de qualité de service peuvent être exécutées sur un système *temps réel souple*. A contrario, lorsque le non-respect des contraintes temporelles engendre une faute grave de l'application, un système *temps réel dur* doit être utilisé. Pour cette dernière catégorie, nous pouvons par exemple citer les applications de pilotage automatique d'avion ou de contrôle des réacteurs d'une centrale nucléaire.

Dans cette thèse, nous considérons des applications peu sensibles, se contentant d'un système temps réel souple. Ce type d'application représente la majeure partie des systèmes temps réel déployés. En effet, de très nombreuses applications possèdent des contraintes temps réel souples. Nous pouvons par exemple citer les applications de visioconférences, de téléphonie, ou de musique assistée par ordinateur.

1.3.2.1 Tâche temps réel

Dans cette section, nous introduisons les caractéristiques d'une tâche temps réel. De manière générale, une tâche temps réel τ_i est caractérisée par
- une DDPT (Date de Démarage au Plus Tôt), notée S_i ;
- un WCET (*Worst Case Execution Time*) , noté C_i ;
- une DFPT (Date de Fin au Plus Tard), notée D_i ;
- une période, notée P_i.

Ces paramètres sont utilisés par l'ordonnanceur afin de fournir à l'utilisateur les contraintes temporelles requises par l'application.

La figure 1.7 présente l'exécution d'une tâche τ_i possédant comme caractéristiques
- une DDPT $S_i = 0$,
- un WCET $C_i = 3$,
- une DFPT $D_i = 4$ et
- une période $P_i = 5$.

L'ordonnanceur dispose alors d'une fenêtre pour lancer l'exécution de la tâche τ_i. Dans cet exemple, lors de la première période, la tâche τ_i a été exécutée immédiatement et son exécution se termine une unité temporelle plus tôt que la DFPT. Durant la seconde période, l'ordonnanceur a choisi d'exécuter la tâche au deuxième *slot* temporel de la fenêtre, ce qui suffit cependant à respecter la DFPT. Cette marge peut permettre à l'ordonnanceur d'adapter les ressources utilisées par une tâche en fonction du flot d'exécution de l'application.

De nombreux algorithmes d'ordonnancement temps réel ont été développés. Le plus connu d'entre eux est certainement l'algorithme EDF (*Earliest Deadline First*) [69]. À chaque *tick* [4], l'algorithme EDF sélectionne la tâche ayant la DFPT la plus proche. Sur

4. Instant où le système d'exploitation peut préempter une tâche.

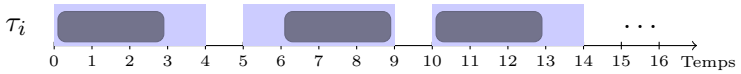


FIGURE 1.7 – Exemple d'exécution d'une tâche temps réel τ_i ayant une DDPT $S_i = 0$, une période $P_i = 5$, une DFPT $D_i = 4$, et un WCET $C_i = 3$.

une architecture monoprocesseur, cet algorithme produit un ordonnancement prouvé optimal. Cependant, son implémentation requiert des calculs complexes limitant son utilisation dans l'industrie.

1.3.2.2 Implémentation des systèmes temps réel

Dans la section précédente, nous avons présenté un modèle de tâches permettant au développeur de fournir des informations relatives au comportement temporel des tâches. En réalité, peu de systèmes temps réel offrent aux développeurs une API (*Application Programming Interface*) permettant de spécifier ces paramètres, ce qui complexifie l'utilisation d'algorithmes d'ordonnancement tels que EDF. Il existe cependant quelques systèmes temps réel implémentant directement l'algorithme EDF tels que MaRTE OS [5] ou RTLinux.

La majeure partie des systèmes temps réel utilise des priorités en lieu et place des contraintes temporelles. Le développeur attribue des priorités aux tâches, et l'ordonnanceur sélectionnera la tâche ayant la priorité la plus importante. Il convient de distinguer deux classes d'ordonnancements fonctionnant avec des priorités [47]. La première, dite à *priorités fixes*, offre au développeur la possibilité d'associer à une tâche une priorité qui, durant toute l'exécution de la tâche, ne peut plus être modifiée. L'algorithme RMS (*Rate Monotonic Scheduling*) [46] exploite ce type de priorités. La seconde classe introduit des *priorités dynamiques* aux tâches. La priorité donnée à une tâche peut alors évoluer durant son exécution, ce qui permet par exemple d'émuler l'algorithme EDF. Dans ce cas, le développeur doit faire évoluer la priorité des tâches en fonction de la proximité de leur DFPT respectives. Le système d'exploitation VxWorks [6] permet l'utilisation de priorités dynamiques.

1.3.2.3 Ordonnancement temps réel multiprocesseur

L'intérêt croissant pour les systèmes multiprocesseur n'a pas épargné le domaine du temps réel. Les algorithmes d'ordonnancement ont donc dû être repensés en tenant compte de ce type d'architectures qui introduit des changements importants. Par exemple, l'algorithme EDF n'est plus optimal sur une architecture multiprocesseur [21]. De nouveaux algorithmes ont donc été développés afin de s'adapter aux architectures multiprocesseur. Ainsi, l'algorithme Pfair a été prouvé optimal sur une architecture multiprocesseur homogène [6].

5. http ://marte.unican.es/
6. http ://www.windriver.com/products/vxworks/

L'ordonnancement pour des architectures composées de plusieurs unités d'exécution introduit une dimension supplémentaire, la *dimension spatiale*. Dans le cas d'une architecture monoprocesseur, l'ordonnancement consiste à déterminer quand une tâche doit être exécutée. Lorsque plusieurs unités d'exécution sont disponibles, l'ordonnanceur doit également déterminer la ressource sur laquelle la tâche va être exécutée.

Dans le cadre des architectures multiprocesseur, deux stratégies ont été proposées. La première consiste à effectuer un ordonnancement local à chaque processeur. Dans ce cas, des tâches sont attribuées à un processeur donné et un ordonnancement local à ce processeur est appliqué. Cette stratégie présente l'avantage d'être implémentable facilement en utilisant des algorithmes d'ordonnancement monoprocesseur, tels que EDF, sur chaque processeur. Cependant, l'association d'une tâche à un processeur est une contrainte pouvant mener à une réduction de la qualité d'ordonnancement. Par exemple, un processeur pourrait être surexploité alors qu'un autre processeur serait sous-exploité.

Pour pallier à ce problème, une autre stratégie consiste à réaliser un ordonnancement global. Dans ce cas, l'ordonnanceur assigne une date de démarrage ainsi qu'un processeur à une tâche au fur et à mesure de l'exécution de l'application. L'ordonnanceur dispose alors d'une vue globale du système et peut ainsi répartir la charge de manière optimisée. L'algorithme Pfair est un algorithme d'ordonnancement global. Ce type d'algorithme nécessite généralement un mécanisme de migration de tâches. La migration d'une tâche consiste à déplacer un tâche d'une ressource à une autre suite à une préemption, ce qui s'avère problématique sur une architecture hétérogène.

1.3.3 Ordonnancement pour architectures reconfigurables

Au même titre qu'un multiprocesseur, une architecture reconfigurable dispose de plusieurs unités d'exécution. La section 1.1 évoquait la possibilité d'exécuter plusieurs tâches sur la zone reconfigurable. L'ordonnancement sur une architecture reconfigurable possède donc deux dimensions, une dimension temporelle et une dimension spatiale. Cependant, la dimension spatiale est plus complexe que dans le cadre d'une architecture multiprocesseur. Dans la section 1.1.3, nous avons vu que des zones de formes et tailles plus ou moins arbitraires peuvent être créées afin d'accueillir des tâches. L'opération de création de ces zones peut alors s'avérer très complexe, notamment lorsque le but est de maximiser le nombre de tâches placées. Dans ce contexte, ce problème est assimilable au problème *Bin Packing*, qui est NP-complet [15].

Les sections suivantes présentent en détail l'ordonnancement spatial, également nommé *placement de tâches*, en exposant notamment des travaux relatifs à cette problématique.

1.4 Placement de tâches sur architectures reconfigurables

1.4.1 Présentation de l'ordonnancement spatial

L'ordonnancement spatial, ou placement, consiste à placer une tâche sur la zone reconfigurable. L'opération de placement d'une tâche est constituée de deux étapes. La première consiste à trouver un emplacement, la seconde consiste à placer physiquement la tâche à cet emplacement. Cette seconde étape a été évoquée dans la section 1.1.3 et est classiquement réalisée via le port de reconfiguration ICAP. Dans la suite, nous nous focalisons donc sur la première étape qui concerne directement les algorithmes de placement.

La zone reconfigurable est une surface rectangulaire. Dans nos travaux, nous considérons que les tâches sont également rectangulaires. Dans la section 1.3.2.1, nous avions présenté un ensemble de variables (DFPT, DDPT, etc.) caractérisant une tâche temps réel. Afin de décrire la dimension spatiale d'une tâche, il est nécessaire d'ajouter à cet ensemble

- un couple de coordonnées, noté (x, y),
- une largeur, notée l_i, et
- une hauteur, notée h_i.

Le placement d'une tâche consiste à déterminer les coordonnées (x, y) d'une zone libre de largeur l_i et de hauteur h_i. Le placement consiste donc à agencer des rectangles dans un conteneur rectangulaire. Lorsque le but est de maximiser le nombre de tâches à placer, ce problème est connu sous le nom de *bin-packing 2D* et est NP-complet [15]. Il paraît donc ambitieux de tenter une résolution optimale de ce problème, d'autant plus que la résolution a lieu pendant l'exécution des tâches. Le surcoût d'exécution de l'ordonnanceur devant être négligeable face aux charges induites par l'exécution des tâches, des heuristiques sont utilisées afin de minimiser le surcoût de l'algorithme de placement. Un objectif généralement visé est de maximiser le nombre de tâches placées, mais d'autres objectifs peuvent être envisagés tels que minimiser la distance entre des tâches communicantes.

Des travaux visent à supporter des tâches ayant une forme de polygone n'ayant que des angles droits [29]. Cependant, afin de simplifier les algorithmes de placement ainsi que le placement physique d'une tâche, la majorité des travaux relatifs à l'ordonnancement spatial considère des tâches ayant une forme rectangulaire. Ainsi, nous ne considérons que des tâches rectangulaires.

Notons enfin que dans nos travaux, nous ne considérons que des tâches matérielles. Il existe cependant des recherches portant sur l'ordonnancement (et donc le placement) de tâches logicielles et matérielles [2].

1.4.2 Travaux relatifs à l'ordonnancement spatial

Dans cette section, les principaux travaux relatifs à l'ordonnancement spatial sont présentés. Notre contexte étant l'ordonnancement en-ligne, nous n'évoquons que très brièvement le placement hors-ligne. De même, nos travaux se focalisant sur les architectures reconfigurables par région, nous ne présentons que sommairement le placement

sur des architectures reconfigurables par colonne.

Placement hors-ligne. Lorsque toutes les tâches d'une application ainsi que le flot d'exécution sont connus lors de la synthèse [24], il est possible de pré-calculer le placement des différentes tâches en fonction des flots d'exécution possibles. Dans ce contexte, cette méthode est très efficace car elle n'engendre pas de surcoût lors de l'exécution. De plus, elle présente l'avantage d'être implémentable très facilement. Cependant, lorsqu'une application possède un caractère dynamique, c'est à dire que son flot d'exécution n'est pas connu lors de la phase de synthèse, cette méthode ne peut être utilisée. Il est alors nécessaire de déterminer le placement lors de l'exécution.

Placement en-ligne. Le placement en-ligne a pour but de déterminer la position d'une tâche lors de l'exécution de l'application. Le contexte d'utilisation est donc le même que dans le cas de l'ordonnancement temporel en-ligne, présenté dans la section 1.3.1.1. Parce que les algorithmes de placement en-ligne s'avèrent généralement complexes à implémenter, un processeur généraliste intégré à la zone reconfigurable est souvent utilisé [10].

1.4.2.1 Placement par colonne

Les algorithmes de placement doivent donc déterminer dans quelle(s) colonne(s) une nouvelle tâche doit être placée. La figure 1.8 représente une zone reconfigurable composée de quatre colonnes. L'espace libre restant sur la zone reconfigurable est la colonne C_2. L'algorithme *1D Horizon*, présenté dans [61], est un exemple d'algorithme de placement pour architectures reconfigurables par colonne. L'avantage principal de cet algorithme est de gérer l'espace libre grâce à une structure de donnée d'implémentation simple ne nécessitant que peu d'espace mémoire.

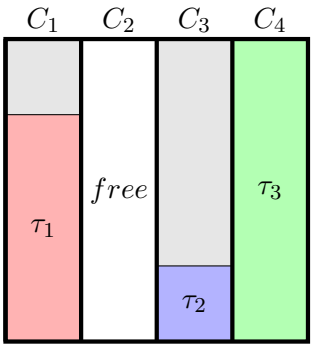


FIGURE 1.8 – FPGA reconfigurable dynamiquement par colonne. Trois tâches sont en cours d'exécution sur le FPGA. L'espace libre restant est la colonne C_2.

1.4.2.2 Approches utilisant la notion de *rectangle vide maximum*

De nombreux travaux de placement sur une architecture reconfigurable par région utilisent la notion de MER (*Maximum Empty Rectangle*) afin de modéliser les parties non utilisées de la zone reconfigurable et de déterminer un emplacement libre [7]. La zone reconfigurable est découpée en rectangles vides maximum, nommés MER, de telle façon qu'il n'existe aucun rectangle contenant une tâche préalablement placée et qui soit entièrement contenu dans un autre rectangle vide. La figure 1.9 représente le placement de trois tâches (τ_1, τ_2 et τ_3) ainsi que les deux MER (M_1 et M_2) associés à ce placement.

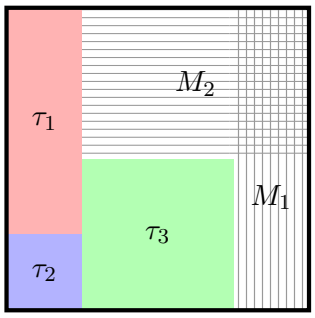


FIGURE 1.9 – Un exemple de placement et les MER M_1 et M_2 associés à ce placement. La zone barrée verticalement correspond au MER M_1, celle barrée horizontalement au MER M_2.

Un algorithme de placement de tâches, basé sur les MER, est proposé dans [7]. Il est constitué de trois principales étapes. La première consiste à déterminer les MER présents sur la zone reconfigurable. La seconde étape consiste à déterminer les MER pouvant accueillir la tâche devant être placée. Enfin, une heuristique est utilisée pour déterminer le MER dans lequel la tâche va être placée. La liste des MER est mise à jour à chaque ajout et suppression de tâches. Afin de simplifier ces opérations, les auteurs proposent de ne considérer que des MER disjoints tels que présentés par la figure 1.10.

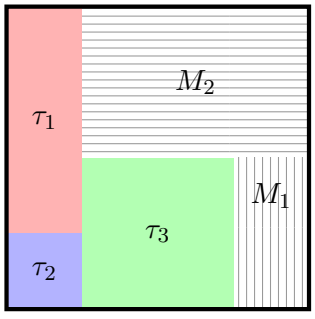


FIGURE 1.10 – Exemple de placement identique à la figure 1.9 où seuls des MER disjoints sont considérés [7].

L'opération d'ajout (resp. suppression) consiste à diviser (resp. fusionner) les rec-

tangles obtenus. Une heuristique est utilisée pour décider de quelle manière est effectuée la division (verticale ou horizontale) lors de l'insertion d'une tâche dans un MER. Une autre technique consiste à retarder la division d'un MER, celle-ci étant alors effectuée lors de l'utilisation d'un sous rectangle [65]. Cette technique permet de choisir le type de division en fonction de la tâche placée ultérieurement afin d'adapter au mieux cette division.

Dans [7], le temps de recherche d'un MER pouvant accueillir une tâche est linéaire par rapport au nombre de tâches déjà placées. Cependant, il est possible d'utiliser le temps durant lequel le placeur est inactif (pendant l'exécution des tâches) pour construire une structure de données permettant de trouver un rectangle en temps constant. Ainsi, les auteurs de [65] utilisent une table de hachage. Cependant, ces algorithmes nécessitent des structures de données particulièrement complexes, qu'il est difficile d'implémenter matériellement.

Les MER permettent de modéliser efficacement l'état de la zone reconfigurable afin de faciliter les ajouts/suppressions de tâches. Des heuristiques s'avèrent nécessaires pour choisir le MER le plus propice - selon certaines contraintes - lors de la soumission d'une tâche. Dans la section suivante, nous présentons des techniques visant à réduire la fragmentation de la zone reconfigurable. Certaines sont utilisées pour sélectionner un MER, d'autres constituent un algorithme de placement à part entière.

1.4.2.3 Réduction de la fragmentation

La zone reconfigurable est fragmentée lorsqu'il existe des ressources utilisées non contiguës. La fragmentation est un problème récurrent que l'on retrouve notamment dans la gestion de la mémoire. La fragmentation d'une zone reconfigurable peut être classée en quatre types [30].

- La fragmentation interne est causée par l'hypothèse que les tâches soient de forme rectangulaire alors que l'outil de synthèse peut créer des tâches ayant des formes plus complexes.
- La fragmentation virtuelle est engendrée par un algorithme de placement qui ne serait pas capable de trouver tous les emplacements vides.
- La fragmentation de partition se produit lorsque la zone reconfigurable est découpée en régions. Lorsqu'une seule tâche peut être placée dans une région, cette région peut posséder de l'espace libre inutilisable.
- La fragmentation externe est due aux insertions et suppressions répétées des tâches. Il apparaît alors des régions libres non contiguës.

La fragmentation interne peut être contrôlée uniquement en utilisant un algorithme de placement supportant des tâches ayant une forme plus complexe qu'un rectangle. La perte de surface engendrée par ce type de fragmentation restant négligeable, nous ne considérons dans nos travaux que des tâches de forme rectangulaire afin de simplifier les algorithmes de placement.

Le modèle de reconfiguration imposé par les architectures reconfigurables engendre nécessairement de la fragmentation de partition. Afin de la minimiser, il conviendrait de

créer des régions de petites tailles. Cependant, les ressources requises par l'implémentation d'un mécanisme de communication entre les régions croissent avec l'augmentation du nombre de régions. Il s'agit donc de trouver un compromis entre la fragmentation de partition et la surface utilisée par le mécanisme de communication.

Finalement, les algorithmes de placement présentés dans la suite de cette section visent à minimiser la fragmentation externe de la zone reconfigurable.

Dans [30], les auteurs définissent une métrique, nommée TF (*Total Fragmentation*), permettant d'évaluer la fragmentation à l'échelle d'un MER. La TF d'une région est d'autant plus petite qu'elle est entourée de régions inutilisées. La TF d'un MER est la moyenne des TF des régions appartenant à ce MER. Cette métrique permet donc de guider le choix d'un MER lors du placement d'une tâche.

Dans [20], les auteurs présentent un algorithme de placement non basé sur les MER permettant de réduire la fragmentation. Lorsqu'une tâche τ doit être placée, l'algorithme construit des *régions interdites*. Une région interdite est une zone dans laquelle la tâche τ chevaucherait une tâche déjà placée. Donc, pour chaque tâche placée, une région interdite est créée. De plus, une autre région interdite est créée de façon à garantir que la tâche τ soit entièrement placée sur la zone reconfigurable. Afin de minimiser la fragmentation, une métrique de densité est créée à partir de ces informations. Cette métrique est basée sur l'assertion suivante : plus un emplacement possède de voisins directs appartenant à des régions interdites, meilleur cet emplacement est. Cette assertion permet donc de favoriser le placement d'une tâche à un emplacement entouré de tâches déjà placées. Notons également que les auteurs étendent cette méthode en considérant le temps. Les tâches ayant des durées de vie, la densité d'une case de la zone reconfigurable évolue dans le temps. Pour placer une tâche τ, la densité d'un emplacement est alors calculée sur toute la durée de vie de la tâche τ.

Tabero et al. proposent une méthode de placement basée sur une *Vertex List* [62]. La figure 1.11 présente un schéma représentant l'état de la zone reconfigurable par une liste de sommets. Si la zone contient plusieurs emplacements libres non adjacents, plusieurs listes de sommets sont créées. Les sommets grisés sont les sommets pouvant accueillir une tâche. Les auteurs proposent ensuite deux heuristiques visant à réduire la fragmentation. La première consiste simplement à évaluer l'adjacence d'un emplacement avec le bord de la zone ou avec des tâches en cours d'exécution. La seconde permet de réduire les placements entraînant la création d'emplacements libres ayant des formes complexes et de petites tailles. La mise à jour des listes lors d'ajouts et de suppressions nécessite des opérations complexes impliquant un contrôle important, limitant ainsi une implémentation matérielle efficace. Cette méthode présente cependant l'avantage de fonctionner avec des formes plus complexes que des rectangles.

Une méthode utilisant un profil des tâches soumises au système est développée par Cui et al. [18]. Ils utilisent la probabilité que la future tâche devant être placée soit d'une certaine taille. En fonction de cette probabilité, ils choisissent dans quel MER va être placée une tâche. Cette méthode est alors étendue afin de considérer la durée de vie des tâches. Cependant, une méthode nécessitant une connaissance a priori des

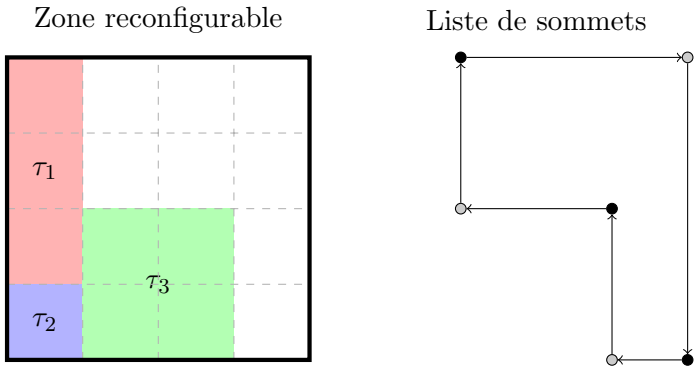


FIGURE 1.11 – Représentation de l'état de la zone reconfigurable et d'une liste de sommets décrivant l'espace libre.

tâches ne peut être utilisée dans le cas général [7].

Les méthodes de placement présentées jusqu'à présent ne tiennent pas compte de l'hétérogénéité de la zone reconfigurable. Or, ce facteur remet en cause bon nombre d'algorithmes, en particulier les algorithmes basés sur les MER.

1.4.2.4 Placement hétérogène

Les algorithmes de placement sur zones reconfigurables hétérogènes doivent prendre en compte la présence des différentes ressources. Considérant qu'une tâche puisse être placée n'importe où, les algorithmes de placement basés sur les MER s'avèrent inadaptés à ce type d'architectures.

Lors de la synthèse, une tâche se voit allouer un ensemble de ressources, ainsi qu'une position sur la zone reconfigurable. Comme évoqué dans la section 1.2.2, les outils de relocation de *bitstream* étant actuellement limités, il paraît intéressant de disposer d'algorithmes de placement hétérogène n'utilisant pas de mécanismes de relocation.

Afin de fournir de la flexibilité au système, il est indispensable de pouvoir placer une tâche en plusieurs endroits de la zone reconfigurable. Une solution consiste à synthétiser plusieurs *bitstream*s d'une même tâche. Ces différents *bitstream*s sont nommés *instances* de tâche. Ainsi, plusieurs instances sont synthétisées pour une tâche donnée, permettant de choisir un emplacement adapté lors de l'exécution de l'application.

Koester et al. proposent un algorithme de placement sur zone hétérogène, utilisant des instances de tâche [44]. Le choix de l'instance est basé sur le degré d'utilisation des régions par les tâches actuellement soumises au système. Pour chaque région, l'algorithme détermine le nombre d'instances utilisant cette région. Lorsqu'une tâche doit être placée, l'instance sélectionnée est l'instance utilisant les régions les moins utilisées. Nous revenons sur cette heuristique dans la section 4.1.3.3, où nous la détaillons

7. Les auteurs justifient la faisabilité de cette méthode en exposant le fait que les FPGA sont généralement utilisés dans des systèmes embarqués, et que la nature des applications soumises à ce type de système est souvent connue à l'avance.

davantage. Deux versions de l'algorithme sont proposées [44]. Une première version, dite statique et nommée SUP Fit (*Static Utilization Probability Fit*) est proposée dans laquelle les degrés d'utilisation des régions sont calculés avant l'exécution de l'application. Si toutes les tâches de l'application ne sont pas connues avant son exécution, une seconde version, dite dynamique et nommée RUP Fit (*Run-time Utilization Probability Fit*), recalcule les degrés d'utilisation des régions à chaque soumission de tâches. Ainsi, lorsqu'une tâche est soumise au système, le degré d'utilisation de chaque région occupée par les différentes instances de cette tâche est mis à jour. Cette deuxième version de l'algorithme nécessite des calculs supplémentaires mais permet de gérer des applications dynamiques.

Les instances de tâche permettent d'offrir de la flexibilité à des applications exécutées sur des architectures reconfigurables sans recourir à des mécanismes de relocation.

Afin d'augmenter la flexibilité du système, il convient d'augmenter le nombre d'instances de tâche. Cependant, les instances étant des *bitstream*s, la place requise pour leur stockage peut devenir problématique. Afin de diminuer le nombre d'instances stockées, Koester et al. proposent de n'en sélectionner qu'un sous ensemble [43]. Ce sous ensemble contient alors des instances utilisant des régions différentes afin de minimiser le nombre instances non plaçables simultanément.

1.5 Synthèse

Nous avons présenté les principaux travaux relatifs au placement de tâches sur architectures reconfigurables dynamiquement. De nombreux travaux considèrent une architecture homogène, et développent des algorithmes pour ce type de composant. Dans ce cadre, nous avons cité les algorithmes basés sur des MER, ainsi que sur des *Vertex List*. Cependant, la majorité des architectures reconfigurables fabriquées sont hétérogènes et cette hétérogénéité rend inexploitable bon nombre de ces algorithmes. En effet, même en considérant l'existence d'un outil de relocation, il reste nécessaire de partitionner la zone reconfigurable en parties de nature différente à cause de la présence des différentes ressources utilisables par une tâche.

À notre connaissance, seuls les travaux de Koester et al. [44] s'adressent aux architectures hétérogènes à travers la notion d'instances de tâches. Dans le chapitre 4, nous reviendrons sur leurs travaux et nous nous en inspirerons pour construire un algorithme basé sur un réseau de neurones de Hopfield. Nous proposerons également un algorithme mixte, utilisant à la fois les instances de tâche et la relocation : les instances sont utilisées pour exécuter une tâche sur des zones pourvues de ressources différentes et la relocation est utilisée afin de déplacer une instance sur des régions disposant d'exactement les mêmes ressources.

Chapitre 2

Réseau de neurones de Hopfield

Les réseaux de neurones artificiels sont des algorithmes bio-inspirés, visant à reproduire le fonctionnement du cerveau. Dans les années 1950, des études neurologiques ont permis d'appréhender le fonctionnement du cerveau animal et d'en exhiber un nouveau modèle de calcul. De ces études sont nées de nombreuses variantes exploitant le concept du neurone. Dans nos travaux, nous nous concentrons sur le modèle développé par John Hopfield en 1982 [34].

La première section de ce chapitre présente des concepts communs aux réseaux de neurones artificiels. Ces concepts permettent d'appréhender la première utilisation des réseaux de Hopfield, présentée dans la deuxième section, à savoir, la réalisation de mémoires associatives. Ces réseaux étant soumis à des problèmes de convergence, nous montrerons, dans la troisième section, sous quelles conditions ils évoluent vers un état stable. Pour prouver cette convergence, une technique inspirée de la seconde méthode de Lyapunov est utilisée. Cette technique consiste à prouver la décroissance d'une fonction, nommée *fonction d'énergie*, associée au réseau. Dans la quatrième partie, nous verrons comment la décroissance de cette fonction est utilisée pour résoudre des problèmes d'optimisations à l'aide de réseaux de Hopfield. Les deux dernières parties du chapitre présentent des méthodes permettant de simplifier la construction d'un réseau de Hopfield associé à un problème donné.

2.1 Réseau de neurones artificiels

Cette section introduit de manière générale les réseaux de neurones artificiels. Des concepts communs à tous les types de réseaux de neurones, tels que la définition d'un neurone formel, y sont présentés.

2.1.1 Neurone biologique

Le neurone est l'unité fonctionnelle de base du système nerveux. La figure 2.1 est une photographie microscopique de neurones biologiques. Nous pouvons observer que les neurones sont connectés les uns aux autres. Les biologistes distinguent deux types de prolongements neuronaux réalisant les connexions : les *dendrites* et les *axones*. De

nombreuses dendrites amènent des stimuli au corps cellulaire qui intègre et propage ou non un stimulus à son unique axone. Ces stimuli peuvent être de nature chimique ou électrique et leur transmission par les prolongements neuronaux est effectué par des *synapses*. Les neurones forment donc un réseau.

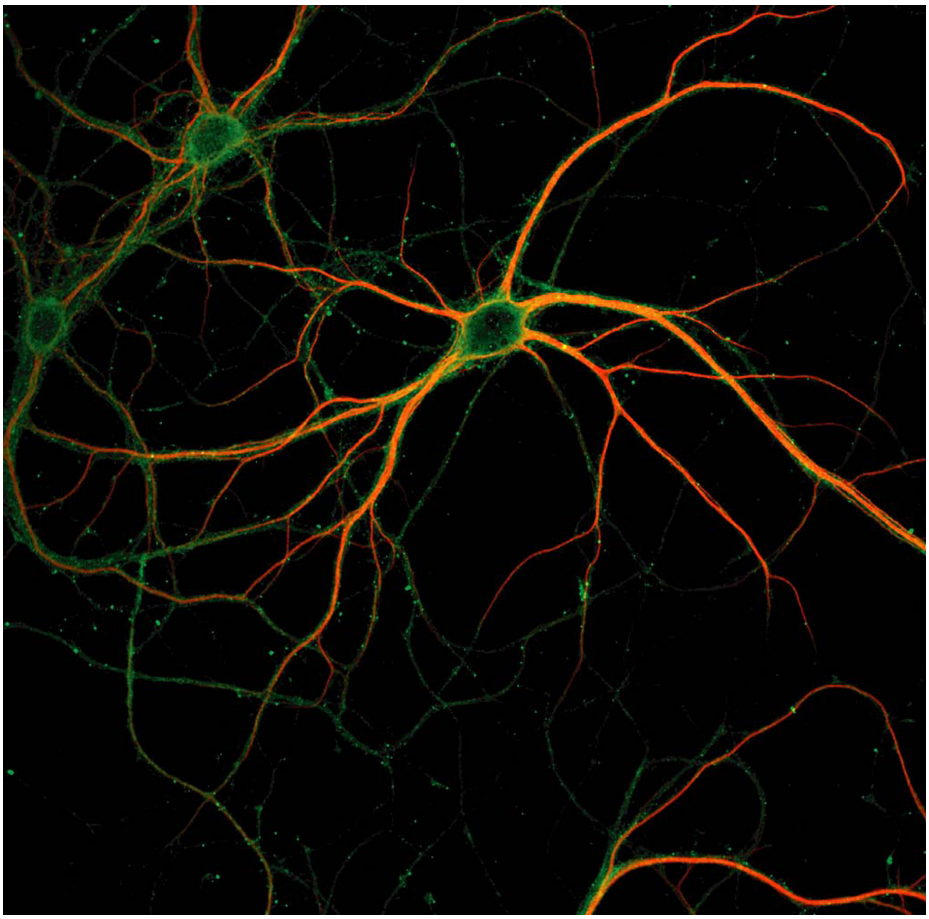


FIGURE 2.1 – Photographie microscopique de neurones (agrandie 650x).

Dans la suite, nous nous focaliserons sur les neurones formels qui sont un modèle simplifié des neurones biologiques. Ainsi, les dendrites seront nommées *entrées*, l'axone, *sortie* et les connexions simplifiées.

2.1.2 Neurone formel

Un neurone formel est une représentation mathématique d'un neurone biologique. Le premier modèle de neurone formel a été proposé par McCulloch et Pitts [50] en 1943, et est illustré par la figure 2.2. Dans cette formulation, un neurone possède un état binaire : il est actif ou inactif (respectivement 0 ou 1).

La figue 2.2 illustre le fonctionnement d'un neurone x_i recevant des stimuli de n

neurones. Ces stimuli sont représentés par un poids de connexion $w_{ji} \in \mathbb{R}$ du neurone j au neurone i. L'évaluation d'un neurone permet de déterminer son état. Elle peut être décomposée en deux principales étapes incarnées par une fonction de combinaison et par une fonction de seuillage.

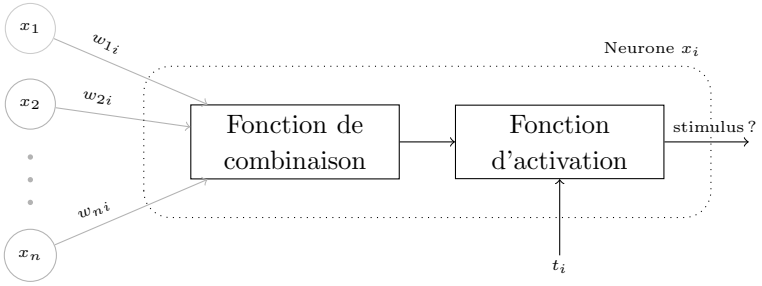


FIGURE 2.2 – Illustration du fonctionnement d'un neurone formel à n entrées.

Fonction de combinaison. La *fonction de combinaison* permet de réduire le vecteur d'entrée d'un neurone à un scalaire. Les valeurs du vecteur d'entrées sont multipliées par l'état du neurone associé à une entrée. Cette fonction est classiquement une simple somme des poids des connexions. Ainsi, pour un neurone i, le terme $\sum_{j=1}^{n} x_j \times w_{ji}$ produit un scalaire qui peut être soumis à la fonction d'activation.

Fonction d'activation. La valeur obtenue suite à la combinaison des entrées est soumise à une *fonction d'activation*. Dans le modèle originel de McCulloch et Pitts [50], cette fonction est la fonction *heaviside* H définie telle que

$$\forall x \in \mathbb{R}, H(x) = \begin{cases} 0 & \text{si} \quad x < 0 \\ 1 & \text{si} \quad x \geq 0 \end{cases} . \tag{2.1}$$

Lorsque la valeur retournée par la fonction *heaviside* est égale à un, l'état du neurone est actif, sinon il est inactif. De nombreuses autres fonctions de seuillage ont été proposées et utilisées telles que la fonction sigmoïde [**?**].

L'état du neurone est donc déterminé par le résultat de la fonction d'activation. Dans le modèle de McCulloch et Pitts, la valeur de l'impulsion et la valeur de l'état du neurone sont identiques et appartiennent à l'ensemble $\{0, 1\}$. Des modèles neuronaux plus complexes permettent, lorsque le neurone est actif, de propager des valeurs réelles issues de la fonction d'activation.

En utilisant la fonction *heaviside* (2.1), la valeur du seuil est implicitement égale à zéro, il peut cependant être intéressant de paramétrer cette valeur. Le paramètre t_i (*threshold*) permet ainsi de modifier la valeur de seuil du neurone x_i. Chaque neurone peut donc posséder une valeur de seuil qui lui est propre. De plus, certains modèles neuronaux modifient cette valeur pendant l'évaluation du réseau. Cette technique est très proche de l'évolution de la température dans l'algorithme du recuit simulé.

Le modèle neuronal de McCulloch et Pitts comporte de nombreuses approximations ou simplifications vis-à-vis du neurone biologique. Il a cependant été montré qu'un tel modèle permettait d'obtenir une puissance de calcul équivalente à celle d'une machine de Turing [60].

En résumé, un neurone formel dispose :
- d'un ensemble d'entrées,
- d'une valeur de seuil et
- d'une sortie associée à l'état du neurone.

Ces valeurs permettent de définir l'équation d'évaluation associée au modèle neuronal de McCulloch et Pitts

$$x_i = H(\sum_{j=1}^{n} x_j \times w_{ji} + t_i), \tag{2.2}$$

où w_{ji} est la valeur de la connexion du neurone x_j vers le neurone x_i. Nous utiliserons principalement cette définition d'un neurone dans nos travaux.

Remarque. L'état des neurones peut être également défini sur l'ensemble $\{-1, 1\}$. Dans ce cas, la fonction de seuil H est remplacée par la fonction SGN

$$\forall x \in \mathbb{R}, SGN(x) = \begin{cases} -1 & \text{si} \quad x < 0 \\ 1 & \text{si} \quad x \geq 0 \end{cases} . \tag{2.3}$$

Cette définition peut permettre de simplifier les valeurs des connexions et des seuils dans certains cas tels que celui présenté dans la section 2.2.2. L'équation d'évaluation d'un neurone devient alors

$$x_i = SGN(\sum_{j=1}^{n} x_j \times w_{ji} + t_i), \tag{2.4}$$

Dans nos travaux, nous utilisons principalement les valeurs $\{0, 1\}$ car elles permettent une implémentation matérielle immédiate, contrairement aux valeurs $\{-1, 1\}$ qui nécessitent une interprétation binaire.

2.1.3 Perceptron, réseau multicouche et autres

La section précédente a présenté le concept du neurone formel. Afin de compléter cette présentation, nous évoquons rapidement leur utilisation au sein d'un réseau très simple proposé par Rosenblatt en 1957 : le perceptron [54]. La figure 2.3 présente un exemple ayant deux entrées. En fournissant aux unités d'entrées deux variables booléennes sur un bit, ce réseau peut par exemple réaliser la fonction logique *et*. Cette fonction est réalisée en paramétrant les poids w_1 et w_2 des connexions. Le neurone utilise l'équation (2.2) pour produire un résultat en sortie.

Malheureusement, le perceptron a très vite montré ses limites car il n'était pas capable de réaliser une fonction non linéaire tel que le *ou exclusif*. Ce problème a été

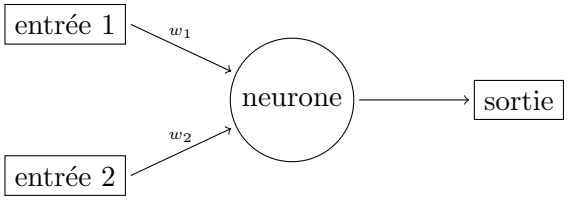


FIGURE 2.3 – Illustration d'un perceptron à deux entrées.

résolu en ajoutant des couches cachées de neurones, c'est à dire des neurones qui ne sont plus directement connectés aux entrées ou sorties du réseau. Ce type de réseau, nommé réseau multicouche, permet de réaliser une fonction non linéaire [32].

2.2 Réseau de Hopfield : généralités

2.2.1 Présentation

En 1982, le physicien John Hopfield [34] proposa un modèle de réseau de neurones récursif. Ce type de réseau a originellement été proposé pour réaliser une mémoire addressable par contenu. Un ensemble de motifs est préalablement mémorisé dans le réseau. En fournissant le fragment d'un motif au réseau, celui ci est alors capable de trouver le motif correspondant au mieux à ce fragment. On parle alors de mémoire.

La figure 2.4 présente un exemple de réseau de Hopfield contenant trois neurones. Il est important d'observer le caractère récursif du réseau : ce type de réseau est modélisé par un graphe orienté complet. Ainsi, un neurone du réseau est connecté à tous les autres neurones du réseau. Les connexions d'un neurone x_i vers un neurone x_j sont notées w_{ij}, et la valeur de seuil d'un neurone x_i est notée t_i.

Par rapport aux réseaux évoqués dans la section 2.1, le caractère récursif des réseaux de Hopfield modifie leur configuration et leur utilisation, parce qu'il n'y a pas d'entrées et de sorties explicites. La section suivante présente brièvement le fonctionnement d'un réseau réalisant une mémoire associative.

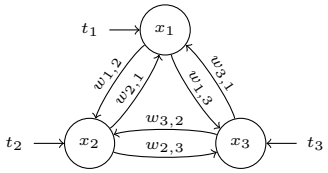


FIGURE 2.4 – Exemple d'un réseau de neurones de Hopfield possédant trois neurones.

2.2.2 Fonctionnement

Pour illustrer simplement le fonctionnement d'un réseau de Hopfield, nous exposons un exemple de mémoire associative. Cet exemple permet de comprendre comment les

données d'un problème sont exposées au réseau, et comment le réseau présente une solution.

La figure 2.5 présente un ensemble de motifs en guise d'exemple. Chaque motif est une image de taille 3×3 pixels, un motif est donc composé de neuf pixels, et chaque pixel est une variable binaire. Le réseau nécessaire pour mémoriser ces motifs possèdent neuf neurones, chaque neurone représentant un pixel. Quand l'état d'un neurone est actif, le pixel lui étant associé est allumé et inversement.

Apprentissage. Pour stocker des motifs dans le réseau, les poids des connexions sont paramétrés de façon à ce que les neurones correspondant à des pixels allumés s'activent simultanément. Cette notion est directement issu de la loi de Hebb [33] stipulant que les connexions entre deux neurones activés simultanément ont tendance à être renforcées afin que l'activation d'un neurone facilite l'activation des autres. Les motifs sont appris en utilisant la relation

$$w_{ij} = \sum_{k=1}^{p} x_i^{(k)} x_j^{(k)}, \tag{2.5}$$

où p est le nombre de motifs et $x_i^{(k)}$ l'état du neurone i appartenant au motif k.

Dans cet exemple, l'état des pixels et des neurones est codé sur l'ensemble $\{-1, 1\}$ où la valeur -1 signifie que le pixel/neurone est désactivé. Afin de définir la matrice de connexions, la loi de Hebb est appliquée sur les motifs présentés par la figure 2.5,

$$\forall i, j \in [1, 9], w_{ij} = \sum_{k=1}^{p=3} x_i^{(k)} x_j^{(k)}, \tag{2.6}$$

où i et j désignent des numéros de pixels, k un numéro de motif, et $x_i^{(k)}$ l'état du pixel i appartenant au motif k. Cette relation permet de définir la matrice de connexions

$$\mathbf{W} = \begin{pmatrix} 0 & -1 & -3 & 1 & -1 & -1 & -1 & 1 & 3 \\ -1 & 0 & 1 & 1 & -1 & 3 & -1 & 1 & -1 \\ -3 & 1 & 0 & -1 & 1 & 1 & 1 & -1 & -3 \\ 1 & 1 & -1 & 0 & -3 & 1 & 1 & 3 & 1 \\ -1 & -1 & 1 & -3 & 0 & -1 & -1 & -3 & -1 \\ -1 & 3 & 1 & 1 & -1 & 0 & -1 & 1 & -1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 0 & 1 & -1 \\ 1 & 1 & -1 & 3 & -3 & 1 & 1 & 0 & 1 \\ 3 & -1 & -3 & 1 & -1 & -1 & -1 & 1 & 0 \end{pmatrix}.$$

Notons que dans cette application, les seuils des neurones sont tous implicitement égaux à zéro parce que nous avons choisi l'ensemble $\{-1, 1\}$ comme ensemble de définition de l'état des neurones.

Utilisation. La matrice $\mathbf{W}$ est utilisée pour paramétrer les connexions du réseau. Des motifs incomplets ou erronés peuvent alors être soumis au réseau, ce dernier tachera de retrouver le motif mémorisé correspondant.

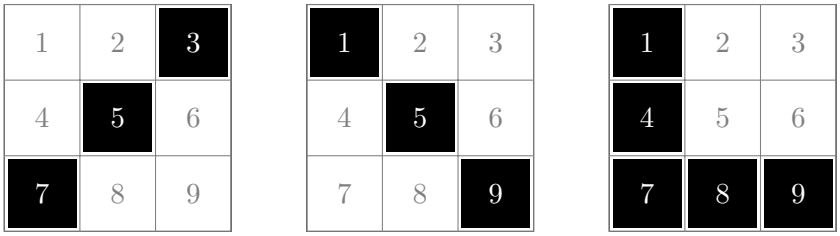


FIGURE 2.5 – Exemple de motifs soumis au réseau. Les pixels de chaque motif sont numérotés de un à neuf et les motifs sont numérotés de un à trois.

Le motif (a) de la figure 2.6 est soumis au réseau de Hopfield, c'est à dire, l'état des neurones est initialisé en fonction des pixels allumés du motif. Sur cet exemple, les neurones x_1 et x_9 sont initialement activés, les autres étant désactivés.

Tous les neurones sont alors évalués séquentiellement au moyen de l'équation (2.4). Lorsque l'état des neurones n'évolue plus au fil des évaluations, le réseau est dit *stable*. Le réseau a ainsi convergé et l'état des neurones code le motif correspondant le plus au motif soumis. Pour reprendre l'exemple précédent, le motif (b) de la figure 2.6 est le motif obtenu par le réseau.

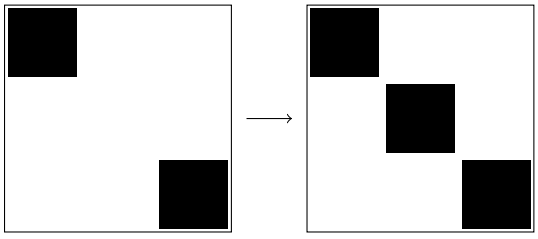


FIGURE 2.6 – Un motif partiel est soumis au réseau. Le réseau répond le motif correspondant.

Cette section a présenté de manière succinte les réseaux de Hopfield permettant de réaliser une mémoire associative. L'apprentissage des motifs, effectué à l'aide de la loi de Hebb, est spécifique aux mémoires associatives et n'est donc pas utilisé pour résoudre des problèmes d'optimisations.

En revanche, le fonctionnement du réseau est commun aux différentes applications. Les neurones sont évalués jusqu'à ce que le réseau soit stable. La convergence du réseau est essentielle et il est nécessaire de s'assurer de sa validité. La section suivante présente une preuve de cette convergence.

2.3 Convergence des réseaux de Hopfield

Dans les parties précédentes, la notion de convergence du réseau a été évoquée. Cette convergence est une caractéristique très intéressante des réseaux de Hopfield car elle participe au fonctionnement simple et autonome de ce type d'algorithme. En effet, il n'est pas nécessaire d'évaluer la solution pour arrêter l'algorithme, celui-ci s'arrête lorsque les neurones n'évoluent plus et cela, indépendamment du problème traité.

Dans cette section, la démonstration de la convergence du réseau est rappelée sous certaines contraintes. Ces contraintes devront donc être respectées lors de la construction du réseau.

Jusqu'à présent, le fonctionnement du réseau a été abordé de manière intuitive. Afin de mener à terme la démonstration, il est nécessaire de décrire un réseau de Hopfield plus formellement. Ainsi, le mode d'évaluation du réseau, c'est-à-dire la façon dont les neurones sont évalués, doit être défini.

Parce que le réseau est récursif, l'évaluation d'un neurone peut potentiellement utiliser l'état de tous les autres neurones du réseau. Il convient donc de distinguer les états des neurones au fil des évaluations. Ainsi, dans la suite, la notation $x_i(t)$ désigne l'état du neurone i suite à l'itération t, une itération correspondant à une évaluation de tous les neurones du réseau.

2.3.1 Modes d'évaluation d'un réseau de Hopfield

2.3.1.1 Mode séquentiel

Le mode le plus utilisé est le *mode séquentiel*. Dans ce cas, tous les neurones sont évalués les uns après les autres. L'ordre d'évaluation des neurones n'a pas d'incidence sur la convergence du réseau. En revanche, il influera sur les solutions générées. Dans le cas d'un ordre prédéfini tel que celui basé sur la numérotation des neurones, les solutions générées seront déterministes : avec les mêmes entrées, plusieurs évaluations du réseau mèneront à la même solution. Pour fournir au réseau un caractère non déterministe, il convient d'évaluer les neurones de manière aléatoire.

L'équation (2.7) exprime l'évaluation d'un neurone en utilisant le mode séquentiel et un ordre d'évaluation correspondant à la numérotation des neurones. L'évaluation du neurone i à l'itération $t+1$ utilise l'état des neurones 1 à $i-1$ calculé à l'itération $t+1$ et l'état des neurones i à n calculé à l'itération t.

$$x_i(t+1) = H(\sum_{j=1}^{i-1} x_j(t+1) \times w_{ji} + \sum_{j=i}^{n} x_j(t) \times w_{ji} + t_i),\ \forall i \in [1,n] \qquad (2.7)$$

2.3.1.2 Mode synchrone

Ce mode d'évaluation consiste à évaluer simultanément tous les neurones. L'intérêt majeur de ce mode est qu'il permet de paralléliser l'évaluation du réseau. En revanche, la qualité des solutions obtenues à travers ce mode est largement détériorée. Ce mode présente donc un intérêt limité. De plus, la convergence d'un réseau utilisant ce mode

n'est plus garantie. Il a en effet été démontré que des « 2-cycles » apparaissent lorqu'un réseau est évalué en utilisant le mode synchrone [41], les « 2-cycles » étant caractérisés par la relation

$$\exists t, X(t) = X(t+k), \forall k \in [2, 4, 6, \ldots, \infty[. \tag{2.8}$$

L'équation (2.10) exprime l'évaluation du vecteur d'état des neurones X. Parce que tous les neurones sont évalués simultanément, l'évaluation d'un neurone à l'itération $t+1$ utilise l'état des neurones à l'itération t

$$x_i(t+1) = H\left(\sum_{j=1}^{n} x_j(t) \times w_{ji} + t_i\right), \forall i, \tag{2.9}$$

ce qui est équivalent, sous forme matricielle à

$$X(t+1)^T = H(X(t)^T \times \mathbf{W} + T^T), \tag{2.10}$$

où $\mathbf{W}$ est la matrice de connexions, X^T le vecteur transposé du vecteur d'état, et T le vecteur de seuil.

2.3.1.3 Mode parallèle

Le mode parallèle, également appelé mode *bloc séquentiel*, est un compromis entre les deux modes évoqués précédemment. Ainsi, des neurones peuvent être évalués en parallèle afin d'accélérer la convergence du réseau. Cependant, des contraintes sur les neurones pouvant être exécutés en parallèle doivent être respectées. Ce mode sera étudié en détail dans la section 5.1.1.

Le but de ce chapitre étant de comprendre le fonctionnement d'un réseau de neurones de Hopfield, nous restreignons l'étude à l'utilisation du mode séquentiel afin de ne pas nuire à la clarté des explications.

2.3.2 Fonction d'énergie

Dans cette partie, nous allons rappeler la démonstration de la convergence d'un réseau de Hopfield. Les réseaux de neurones de Hopfield étant des systèmes dynamiques non linéaires à temps discret, il y a de nombreux points commun avec les systèmes dynamiques non linéaires à temps continu largement étudiés, et en particulier, l'étude de la convergence de tels systèmes qui utilise la *seconde méthode de Lyapunov* [48]. La méthode que nous allons utiliser dans le cadre des réseaux de Hopfield est directement inspirée de la méthode de Lyapunov.

L'idée de cette méthode est de parvenir à associer au vecteur d'état une fonction décroissante tout au long de l'évolution du réseau. Ainsi, le réseau convergera vers un point fixe. Cette fonction, notée E, est appelée *fonction d'énergie* par analogie à certains systèmes physiques.

Soit X le vecteur d'état associé à un réseau de Hopfield de n neurones avec des états appartenant à l'ensemble $[0, 1]$. Supposons qu'il existe une fonction $E(X)$ décroissante tant que le réseau évolue, c'est à dire, tant que $X(t) \neq X(t+1)$. Parce que le nombre de vecteurs d'état est borné par 2^n, l'ensemble des vecteurs d'état sera épuisé et un cycle ne peut pas exister car la fonction $E(X)$ est décroissante. Finalement, le réseau convergera vers un état stable où $X(t) = X(t+1)$.

Dans [34], John Hopfield proposa initialement la fonction d'énergie

$$E(X) = -\frac{1}{2}\sum_i \sum_j w_{ij} \times X_i \times X_j - \sum_i X_i \times t_i, \tag{2.11}$$

pouvant également être exprimée sous forme vectorielle

$$E(X) = -\frac{1}{2} X^T \times \mathbf{W} \times X - X^T \times T. \tag{2.12}$$

Il s'agit donc de montrer la décroissance de cette fonction appliquée aux réseaux de Hopfield. Pour prouver la décroissance, nous allons montrer que pour toute itération t, $E(X(t+1)) < E(X(t))$ si $X(t+1) \neq X(t)$. Il faut donc que la différence $\Delta(E) = E(X(t+1)) - E(X(t))$ soit négative. Pour mener à terme cette étude, des contraintes sur les paramètres du réseau doivent être définies. Ainsi,

- la matrice de connexions doit être symétrique ($w_{ij} = w_{ji}$) et
- les éléments de sa diagonale doivent être positifs ou nuls.

Ces contraintes s'avèreront nécessaires et seront rappelées lors de l'étude de la fonction d'énergie.

Dans la suite, nous allons étudier le signe de $\Delta(E)$ en considérant l'utilisation du mode séquentiel défini à la section 2.3.1. L'équation d'évaluation des neurones utilisée est donc l'équation (2.7).

Les neurones étant évalués séquentiellement, le signe de $\Delta(E)$ est étudié suite à l'évaluation d'un neurone. Afin de simplifier les notations, nous considérons l'évaluation du neurone associé au premier élément du vecteur d'état. Nous pouvons réécrire le vecteur d'état X, la matrice de connexions $\mathbf{W}$ ainsi que le vecteur de seuil T comme suit

$$X = \begin{bmatrix} x_1 \\ X' \end{bmatrix} \mathbf{W} = \begin{bmatrix} w_{11} & W_1^T \\ W_1 & \mathbf{W'} \end{bmatrix} T = \begin{bmatrix} t_1 \\ T' \end{bmatrix} \tag{2.13}$$

où $X' = [x_2, x_3, ..., x_n]$, $\mathbf{W}$ et T sont partitionnés de façon équivalente. En utilisant ce partitionnement, nous pouvons développer $E(X)$ à l'instant t de la façon suivante

$$\begin{aligned} E(X(t)) &= -\frac{1}{2}\left[x_1^2(t) \times w_{11} + X'^T(t) \times \mathbf{W'} \times X'(t) + 2x_1(t) \times W_1 \times X'(t)\right] \\ &\quad - \left[x_1(t) \times t_1 + X'(t) \times T'\right]. \end{aligned}$$

Comme précisé ci-dessus, le premier neurone est évalué à l'instant $t+1$, ce qui mène à l'énergie

$$\begin{aligned} E(X(t+1)) &= -\frac{1}{2}\left[x_1^2(t+1) \times w_{11} + X'^T(t) \times \mathbf{W'} \times X'(t) + 2x_1(t+1) \times W_1 \times X'(t)\right] \\ &\quad - \left[x_1(t+1) \times t_1 + X'(t) \times T'\right]. \end{aligned}$$

À partir de cette expression, nous pouvons alors exprimer la différence $\Delta(E)$ en considérant l'évaluation du neurone associé au premier élément du vecteur d'état X. L'objectif étant de montrer que $\Delta(E)$ est négatif, l'expression est développée pour y faire apparaître des termes analysables. La seule astuce de ce développement concerne l'expression (2.14) où une transformation est effectuée afin de faire apparaître l'équation (2.7).

$$\begin{aligned}
\Delta(E) &= E(X(t+1)) - E(X(t)) \\
&= -\frac{1}{2}\left[x_1^2(t+1)w_{11} + X'^T(t)\mathbf{W}'X'(t) + 2x_1(t+1)W_1X'(t)\right] \\
&\quad - \left[x_1(t+1)t_1 + X'(t)T'\right] \\
&\quad + \frac{1}{2}\left[x_1^2(t)w_{11} + X'^T(t)\mathbf{W}'X'(t) + 2x_1(t)W_1X'(t)\right] \\
&\quad + \left[x_1(t)t_1 + X'(t)T'\right] \\
&= -\frac{1}{2}x_1^2(t+1)w_{11} + \frac{1}{2}x_1^2(t)w_{11} \\
&\quad -x_1(t+1)W_1X'(t) + x_1(t)W_1X'(t) - x_1(t+1)t_1 + x_1(t)t_1 \\
&= -\frac{1}{2}w_{11}[(x_1(t+1) - x_1(t))^2 + 2x_1(t+1)x_1(t) - 2x_1^2(t)] \qquad (2.14) \\
&\quad -x_1(t+1)W_1X'(t) + x_1(t)W_1X'(t) - x_1(t+1)t_1 + x_1(t)t_1 \\
&= -\frac{1}{2}\overbrace{w_{11}(x_1(t+1) - x_1(t))^2}^{A} - \overbrace{(x_1(t+1) - x_1(t))\left(W_1X'(t) + x_1(t)w_{11} + t_1\right)}^{B}
\end{aligned}$$

Si les termes A et B sont tous deux positifs, alors la différence $\Delta(E)$ est négative. Concernant le terme A, son signe est déterminé par le signe de w_{11}. Or les éléments diagonaux de la matrice de connexions ayant été définis positifs, le terme A est positif.

Le second produit du terme B correspond à l'équation (2.7) d'évaluation séquentielle d'un neurone

$$\begin{aligned}
B &= (x_1(t+1) - x_1(t))\left(W_1X'(t) + x_1(t)w_{11} + t_1\right) \\
&= \overbrace{(x_1(t+1) - x_1(t))}^{B_1}\overbrace{\left(\sum_{j=1}^{n} x_j(t)w_{ji} + t_1\right)}^{B_2}. \qquad (2.15)
\end{aligned}$$

L'expression d'évaluation d'un neurone i à l'instant t détermine l'état de ce neurone à l'instant $t+1$, nous pouvons ainsi mettre en relation les termes B_1 et B_2 de l'expression (2.15). Plus précisément, nous pouvons déterminer le signe de B_2 en fonction du signe de $x_1(t+1)$.

Si $x_1(t+1) = x_1(t)$, le réseau n'a pas évolué, or nous étudions la différence d'énergie du réseau suite à une évolution. Nous considérons donc que $x_1(t+1) \neq x_1(t)$.

Parce que l'état des neurones est une variable binaire définie sur l'ensemble $\{0, 1\}$, deux cas peuvent se produire, à savoir $x_1(t+1) = 0$ ou $x_1(t+1) = 1$. Nous allons ainsi examiner l'expression B dans ces deux cas.

En vertu de l'équation (2.7),

$$x_1(t+1) = H(\sum_{j=1}^{n} x_j(t)w_{ji} + t_1) \quad (2.16)$$

$$= H(B_2). \quad (2.17)$$

Donc, d'après l'équation (2.1), $x_1(t+1) = 0$ implique que le terme B_2 est négatif ou nul. De plus, si $x_1(t+1) = 0$ alors $B_1 \leq 0$. Nous pouvons en déduire que l'expression B est positive lorsque $x_1(t+1) = 0$.

Concernant le second cas, à savoir $x_1(t+1) = 1$, le même type de raisonnement est appliqué. Dans ce cas, B_1 et B_2 sont tous deux positifs ou nul. Nous en déduisons que l'expression B est positive lorsque $x_1(t+1) = 1$.

Il s'avère donc que l'expression B est positive. Parce que nous avions précédemment montré que l'expression A est positive, l'expression $\Delta(E)$ est négative et donc la fonction d'énergie E est décroissante tant que $x_1(t+1) \neq x_1(t)$.

Finalement, en s'inspirant de la seconde méthode de Lyapunov, nous pouvons conclure que le réseau évolue vers un point fixe si
- la matrice de connexions est symétrique et si
- ses éléments diagonaux sont positifs ou nuls.

2.4 Réseau de Hopfield pour l'optimisation

Dans les sections précédentes de ce chapitre, les réseaux de neurones de Hopfield ont été présentés en tant que mémoires associatives. Cependant, ils peuvent également être utilisés pour résoudre des problèmes d'optimisations comme l'ont imaginés Hopfield et Tank dans [35]. Leur idée a été d'exprimer un problème d'optimisation sous la forme d'une fonction d'énergie puis de faire évoluer le réseau associé afin de minimiser cette fonction d'énergie. La solution obtenue est donc codée par l'état des neurones.

2.4.1 Définition d'un codage du problème

Afin de résoudre un problème d'optimisation au moyen d'un réseau de neurones de Hopfield, il convient de définir un codage du problème adapté à la technique de résolution. Un réseau de neurones étant caractérisé par un vecteur de variables binaires, il faut définir un codage du problème où une solution au problème est représentée par un vecteur de variables binaires. L'ensemble des états du réseau correspond donc à l'ensemble des solutions du problème. Le réseau parcourera cet ensemble afin de trouver une solution satisfaisante.

2.4.2 Construction d'un réseau de Hopfield associé à un problème d'optimisation

Dans la section 2.3.2, la décroissance de la fonction d'énergie a été utilisée pour montrer la convergence du réseau. Or, nous pouvons également affirmer que lorsque le

réseau converge, la fonction d'énergie décroît. Ainsi, un point fixe du réseau correspond à un minimum local de la fonction d'énergie.

Il s'agit donc d'exprimer un problème d'optimisation P sous la forme d'une fonction d'énergie. Cette démarche est illustrée par la figure 2.7. Un problème d'optimisation P est décrit par une fonction de la forme

$$P(X) = F_{contraintes}(X) + F_{coût}(X)$$

où X est un vecteur de variables binaires, $F_{contraintes}$ représente des contraintes associées au problème P et $F_{coût}$ une fonction caractérisant le coût d'une solution devant être minimisée.

La première étape consiste à transformer P en une fonction d'énergie. Cette étape est illustrée par des exemples dans les sections suivantes. La seconde étape consiste à utiliser la matrice de connexions et le vecteur de seuil issus de la fonction d'énergie pour paramétrer le réseau.

La figure 2.7 montre une étape de transformation du problème P en une fonction d'énergie E faisant apparaître la matrice $\mathbf{W}$ ainsi que le vecteur T. Un réseau de neurones de Hopfield peut donc être construit en utilisant $\mathbf{W}$ et T.

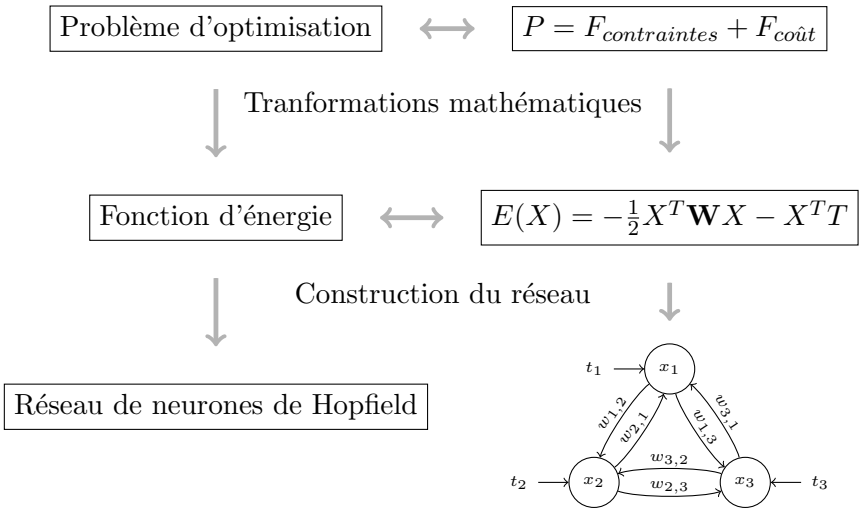


FIGURE 2.7 – Transformation d'un problème d'optimisation en un réseau de neurones de Hopfield. Nous supposons que le problème est codé sous une forme exploitable par un réseau de neurones.

2.4.3 Utilisation d'un réseau associé à un problème d'optimisation

Dans les sections précédentes, nous avons défini un codage et exprimé le problème sous la forme d'un réseau de neurones de Hopfield. Il s'agit maintenant de lancer et laisser converger le réseau afin d'obtenir une solution.

D'après la section 2.3.2, d'un état initial quelconque, le réseau converge vers un minimum local de la fonction d'énergie et donc vers un minimum local du problème

d'optimisation considéré. Ainsi, pour obtenir une solution, il suffit d'initialiser le réseau avec un état quelconque, laisser converger le réseau et lire l'état des neurones. Le vecteur d'état X code la solution trouvée par le réseau.

À propos du non-déterminisme. Jusqu'à présent, nous avons principalement évoqué le fonctionnement du réseau d'une manière déterministe, notamment lors de l'étude de la convergence car il était supposé que les neurones étaient évalués dans l'ordre de leur numérotation. Or, une caractéristique intéressante des réseaux de Hopfield est leur non-déterminisme.

Dans le cas général, les réseaux de Hopfield ne garantissent pas l'obtention d'une solution optimale. Il est alors intéressant que le réseau ait la capacité de générer différentes solutions. Du non-déterminisme peut être introduit en paramétrant aléatoirement l'état initial du réseau et en évaluant les neurones dans un ordre aléatoire. Généralement, ces deux techniques sont utilisées simultanément.

2.5 Règles génériques de construction d'un réseau

Les problèmes d'optimisations sont modélisés sous la forme d'un ensemble de contraintes et d'une fonction de coût. Afin de simplifier la transformation du problème d'optimisation en fonction d'énergie, des règles permettant de satisfaire des contraintes génériques ont été définies. Ces règles peuvent alors être appliquées simplement sur une partie du réseau. Les sections suivantes présentent quelques règles majeures.

2.5.1 Règle k-de-n

Tagliarini et al. [63] ont proposé la règle k-de-n. Cette règle permettant d'activer k neurones parmi n, l'évolution du réseau doit mener à un état stable tel que

$$\sum_{1}^{n} x_i = k$$

où x_i est l'état du neurone i et n le nombre de neurones dans le réseau.

Dans un premier temps, il s'agit de trouver une fonction qui est minimale lorsque k neurones sont actifs parmi n. Tagliarini et al. ont proposé la fonction

$$E_{k\text{-de-}n} = (k - \sum_{1}^{n} x_i)^2$$

qui est égale à zéro et minimale lorsque k neurones sont activés.

La fonction E n'est pas assimilable à une fonction d'énergie définie dans la section 2.3.2. Elle peut cependant être simplement transformée afin d'y faire apparaître la

matrice de connexions et le vecteur de seuil.

$$\begin{aligned} E_{k\text{-de-}n} &= (k - \sum_{1}^{n} x_i)^2 \\ &= k^2 - 2k \sum_{1}^{n} x_i + (\sum_{1}^{n} x_i)^2 \\ &= k^2 - 2k \sum_{1}^{n} x_i + 2 \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} x_i x_j + \sum_{1}^{n} x_i \\ &= k^2 + \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} x_i x_j - (2k-1) \sum_{1}^{n} x_i \\ &= k^2 - \frac{1}{2} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} (-2) x_i x_j - \sum_{1}^{n} (2k-1) x_i \end{aligned} \tag{2.18}$$

Le terme k^2 est une constante n'influençant pas les positions des minimums locaux, il peut donc être supprimé de l'expression (2.18) qui devient

$$E_{k\text{-de-}n} = -\frac{1}{2} \sum_{i}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} (-2) x_i x_j - \sum_{i} (2k-1) x_i. \tag{2.19}$$

L'expression (2.19) correspond à la fonction d'énergie telle qu'elle a été définie dans la section 2.3.2. Elle spécifie donc la matrice de connexion et le vecteur de seuil nécessaire à la réalisation d'un réseau ayant un état stable où k neurones parmi n sont activés. En identifiant les termes de (2.19) par rapport à ceux de la fonction d'énergie (2.11), les expressions

$$\begin{aligned} w_{ij} &= \begin{cases} -2 \text{ si } i \neq j, \forall i, j \\ 0 \text{ sinon} \end{cases} \\ t_i &= 2k - 1 \ \forall i \end{aligned} \tag{2.20}$$

permettent donc de paramétrer une règle k-de-n.

La figure 2.8 présente l'application de la règle k-de-n avec $k = 2$ sur un réseau de Hopfield de 3 neurones. Comme le stipulent les expressions (2.21), le poids des connexions est paramétré à -2 et les valeurs de seuil des neurones à $2k - 1$.

Il est relativement aisé de comprendre le fonctionnement de la règle sur cet exemple. En considérant les valeurs utilisées dans la figure 2.8, la valeur de seuil d'un neurone est égale à 3. En l'absence d'énergie négative apportée par d'autres neurones, l'évaluation de ce neurone l'activera d'après l'équation (2.7). Pour désactiver ce neurone, il faut qu'il reçoive une énergie négative strictement supérieure à -3, ce qui correspond à l'énergie apportée par deux neurones actifs. En effet, les poids des connexions étant fixés à -2, deux neurones actifs apportent une énergie égale à -4. En résumé, si moins de deux

neurones sont activés, alors l'évaluation d'un autre neurone activera ce dernier et, si deux neurones ou plus sont activés, l'évaluation d'un autre neurone le désactivera. Ce comportement correspond bien au comportement souhaité par l'application de la règle k-de-n.

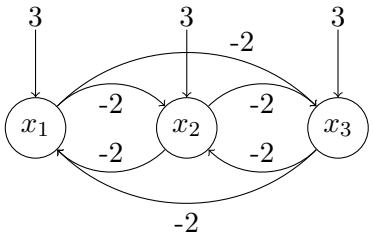


FIGURE 2.8 – Exemple de l'application d'une règle 2-de-3. Les connexions sont paramétrées à -2 et les valeurs de seuil à $2k-1=3$ avec $k=2$.

2.5.2 Règle au-plus-k-de-n

Dans [63], Tagliarini et al. propose également la règle au-plus-k-de-n permettant de résoudre des contraintes du type

$$\sum_{1}^{n} x_i \leq k.$$

Cette règle est construite à partir de la règle k-de-n en ajoutant des neurones cachés. Les neurones x_4 et x_5 de la figure 2.9 sont des neurones cachés. Une règle k-de-n est appliquée sur les cinq neurones du réseau. Ainsi, lors de l'évolution du réseau, il est possible que des neurones cachés soient activés. Si tel est le cas, moins de deux neurones effectifs seront activés ce qui correspond au comportement attendu de la règle au-plus-k-de-n.

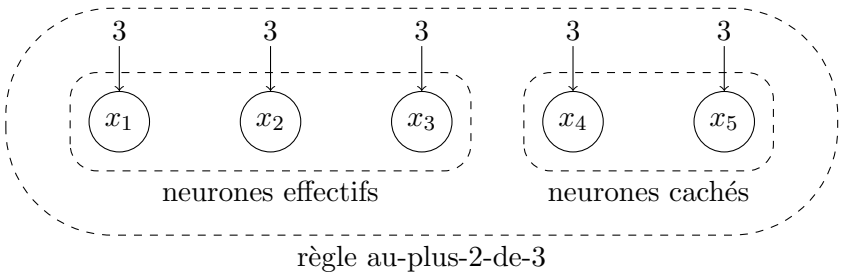


FIGURE 2.9 – Exemple de l'application d'une règle au-plus-2-de-3. Les connexions sont paramétrées à -2 et les valeurs de seuils à $2k-1=3$ avec $k=2$. Les neurones x_1 x_2 et x_3 sont des neurones utilisés par le codage du problème contrairement aux neurones cachés x_4 et x_5 qui ne sont utilisés que par la règle au-plus-2-de-3

N'étant pas nécessaire au codage du problème, les neurones cachés augmentent articiellement le nombre de neurones. Or, l'évaluation du réseau consistant à évaluer

successivement tous les neurones, l'augmentation du nombre de neurones induit un surcoût lors de l'évaluation du réseau. Il est donc préférable d'éviter les règles ajoutant des neurones cachés au réseau.

2.5.3 Règle 0-ou-1-de-n

La règle 0-ou-1-de-n est un cas particulier de la règle au-plus-k-de-n en paramétrant k à 1. Il est cependant possible d'éviter l'ajout de neurones cachés. La fonction

$$E = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} x_i x_j$$

est minimale (égale à zéro) s'il y a zéro ou un neurone activé. Une simple réécriture de la fonction E

$$E = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} x_i x_j = -\frac{1}{2} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} (-2) x_i x_j - \sum_{i=1}^{n} 0 \times x_i$$

suffit à exhiber les poids de connexions ainsi que les valeurs de seuil tels que

$$\begin{aligned} w_{ij} &= \begin{cases} -2 \text{ si } i \neq j \\ 0 \text{ sinon} \end{cases} \\ t_i &= 0 \ \forall i \end{aligned} \tag{2.21}$$

2.5.4 Règle de k-consécutivité

La règle de k-consécutivité est une règle visant à assurer que k neurones parmi n consécutifs soient activés. Carderia et al. [12] ont proposé la règle *succ-k-de-n* dont le rôle est identique. Cependant, à cause de la fonction d'énergie qu'ils ont choisie, le fonctionnement du réseau a été modifié. Ils ont en effet proposé l'ajout de neurones inhibiteurs. Un neurone inhibiteur est un neurone qui peut inhiber une connexion, c'est à dire forcer son poids à zéro. Cette extension nécessite une modification du modèle de Hopfield qui se traduit par une implémention plus complexe et moins performante.

Nous proposons une autre modélisation de cette règle, qui présente l'avantage de ne nécessiter aucune extension au modèle de réseau défini par Hopfield. La figure 2.10 représente un réseau de sept neurones ainsi qu'un sous ensemble des connexions permettant de réaliser un règle de 2-consécutivité. Afin d'alléger le schéma, seules les connexions issues des neurones x_3 et x_4 sont représentées. L'idée principale est de désactiver des neurones séparés de plus de deux neurones. Nous pouvons ainsi distinguer deux types de connexions, les connexions ayant comme poids a et 0, où a est un poids négatif suffisament important pour désactiver les neurones ciblés par ce type de connection. Lorsqu'il est activé, les connexions de poids a du neurone x_3 lui permettent d'inhiber l'activation des neurones distant de plus de deux neurones, par exemple, le neurone x_1. Les connexions de poids 0, quant à elles, n'auront aucune influence sur l'activation des neurones les recevant.

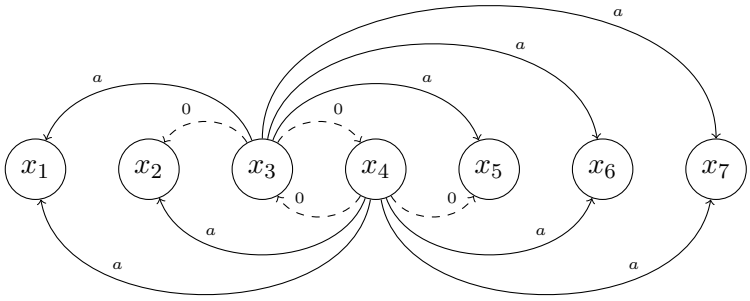


FIGURE 2.10 – Exemple d'une règle de k-consécutivité de 2 neurones parmi 7. Seules les connexions des neurones x_3 et x_4 sont représentées.

Soit k le nombre de neurones consécutifs souhaités, la fonction

$$\sum_{i=1}^{N} \sum_{\substack{j=1 \\ j\neq i \\ j\neq\{i+1,\ldots,i+k\} \\ j\neq\{i-1,\ldots,i-k\}}}^{N} x_i x_j \tag{2.22}$$

est minimale s'il n'y a pas de neurones non consécutifs activés. En effet, les connexions de neurones distants de plus de k neurones ne sont pas prises en compte dans cette somme. Cependant, cette fonction n'est pas satisfaisante car elle est également minimale lorsqu'aucun neurone n'est activé. Or, le terme $-\sum_{i=1}^{N} x_i$ est minimal lorsque N neurones sont activés. Nous l'ajoutons à la fonction (2.22) qui devient

$$E = \overbrace{\sum_{i=1}^{N} \sum_{\substack{j=1 \\ j\neq i \\ j\neq\{i+1,\ldots,i+k\} \\ j\neq\{i-1,\ldots,i-k\}}}^{N} x_i x_j}^{A} \overbrace{- \sum_{i=1}^{N} x_i}^{B} . \tag{2.23}$$

Il n'est cependant plus évident que cette fonction soit minimale lorsque k neurones consécutifs sont activés. Lors des activations de k neurones consécutifs, le terme A de l'équation (2.22) est nul et le terme B décroît. Lorsque k neurones consécutifs sont activés, l'activation d'un neurone supplémentaire engendrerait une croissance de E car

- le terme A est supérieur ou égal à 2 et
- le terme B est égal à -1.

La fonction E est donc minimale lorsque k neurones consécutifs sont activés. Il convient

maintenant de transformer cette fonction sous la forme d'une fonction d'énergie (2.11).

$$\begin{aligned} E &= \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j\neq i \\ j\neq\{i+1,\ldots,i+k\} \\ j\neq\{i-1,\ldots,i-k\}}}^{N} x_i x_j - \sum_{i=1}^{N} x_i \\ &= -\frac{1}{2} \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j\neq i \\ j\neq\{i+1,\ldots,i+k\} \\ j\neq\{i-1,\ldots,i-k\}}}^{N} -2x_i x_j - \sum_{i=1}^{N} 1x_i \\ &= -\frac{1}{2} \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j\neq i}}^{N} w_{i,j} x_i x_j - \sum_{i=1}^{N} t_i x_i \\ \text{avec} \quad w_{ij} &= \begin{cases} 0 & \text{si } j = \{i, i+1, i-1, ..., i+k, i-k\} \\ -2 & \text{sinon} \end{cases} \\ t_i &= 1 \end{aligned} \tag{2.24}$$

D'après l'équation (2.24), la matrice **W** est symétrique et les valeurs de sa diagonale sont nulles, elle satisfait donc les critères de convergence. Les équations (2.24) définissent donc une règle de k-consécutivité qui ne nécessite aucune modification du modèle de réseau défini par Hopfield.

2.6 Propriété d'additivité des réseaux de Hopfield

Les sections précédentes ont présenté diverses règles permettant de construire simplement un réseau de neurone de Hopfield adapté à un problème particulier. Cependant, un problème est généralement plus complexe que simplement « activer k neurones parmi n ». Pour exprimer cette complexité, les règles doivent être combinées. Les réseaux de Hopfield permettent d'additionner des règles tout en garantissant la convergence du réseau.

Toutes les règles présentées sont décrites par une fonction d'énergie satisfaisant les critères de convergence définis dans la section 2.3. Considérons deux fonctions d'énergie E_1 et E_2 respectivement associées à deux règles r_1 et r_2 appliquées sur un même réseau. Ces fonctions sont par définition décroissantes tant que le réseau évolue. La combinaison des règle r_1 et r_2 s'exprime par une nouvelle fonction d'énergie

$$E = E_1 + E_2$$

qui est la somme de deux fonctions décroissantes. Ainsi, la fonction E est également décroissante et donc la convergence du réseau est garantie lors de la combinaison de règles.

La propriété d'additivité permet donc de construire un réseau en associant des règles tout en préservant la convergence du réseau.

2.7 Conclusion

Dans ce chapitre, nous avons présenté les réseaux de neurones de Hopfield. Nous avons pu constater qu'ils sortent du modèle classique des réseaux de neurones (de type Perceptron par exemple). En effet, leur caractère récursif modifie complètement leur utilisation ainsi que leur configuration. Il peut s'avérer particulièrement complexe de configurer un réseau - définir une fonction d'énergie - pour qu'il résolve un problème donné, c'est pourquoi nous avons présenté des règles de construction permettant de simplifier leur application. Ainsi, dans les chapitres suivants, nous utiliserons fréquemment des règles de construction pour définir des réseaux de Hopfield résolvant des problèmes d'ordonnancement.

Chapitre 3

Ordonnancement temporel par réseaux de neurones de Hopfield

Ce chapitre présente un ordonnanceur temporel, basé sur des réseaux de neurones de Hopfield, supportant une architecture reconfigurable hétérogène [13]. La première section présente la modélisation neuronale d'un problème d'ordonnancement. Dans un premier temps, les travaux fondateurs [12] de l'ordonnancement par réseau de neurones de Hopfield, bien que destinés à une architecture homogène, sont présentés afin d'introduire l'idée principale de cette modélisation. Ces travaux ont été étendus à la gestion d'une architecture hétérogène dans [9], cependant, le nombre de neurones nécessaires à la modélisation d'une architecture hétérogène était problématique. Nous proposons donc une réduction importante du nombre de neurones nécessaires à la réalisation de l'ordonnanceur. Ainsi, la deuxième section introduit un nouveau type de neurones permettant de réduire le nombre de neurones, à savoir, les neurones inhibiteurs. Finalement, la troisième section présente des résultats d'ordonnancement ainsi qu'une comparaison avec l'algorithme Pfair. Cette section montre que les neurones inhibiteurs permettent de réduire la quantité de ressources nécessaire à l'implémentation du réseau de neurones et d'accélérer le temps de convergence du réseau.

3.1 Modélisation neuronale d'un problème d'ordonnancement

Afin de construire un réseau de neurones résolvant le problème d'ordonnancement temporel, il est nécessaire de modéliser ce problème sous une forme convenable. Dans un premier temps, nous présentons une modélisation de l'ordonnancement temporel pour une architecture homogène. Une extension de cette modélisation permettant de supporter une architecture hétérogène est présentée dans la deuxième partie de cette section.

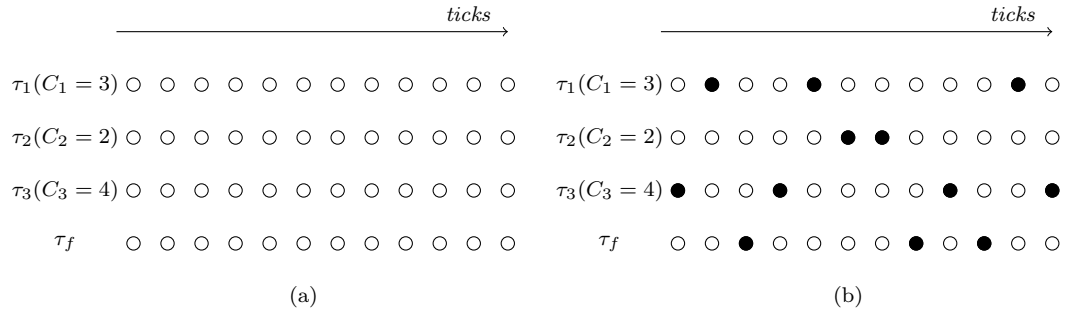


FIGURE 3.1 – (a) La représentation initiale d'un problème d'ordonnancement de trois tâches sur une architecture disposant d'une ressource d'exécution. Chaque ligne correspond à une tâche τ_i ayant un WCET C_i. Les colonnes correspondent à des *slots* d'exécution. (b) Un exemple de scénario d'exécution valide.

3.1.1 Architecture homogène

Cardeira et al. ont proposé un réseau de neurones de Hopfield permettant de résoudre un problème d'ordonnancement pour un mono-processeur [12]. Ces travaux sont principalement basés sur la règle k-de-n définie dans la section 2.5.1.

La figure 3.1.a présente un exemple de modélisation neuronale d'un problème d'ordonnancement mono-processeur. Chaque cercle représente un neurone. Chaque ligne de neurones est associée à une tâche τ_i, et chaque colonne à un *slot*. Le nombre de colonnes S est égale au nombre de *slots* d'exécution. Dans la suite, nous notons $x_{i,j}$ l'état (actif ou inactif) du neurone $x_{i,j}$ associé à la tâche τ_i et au *slot* j. Un neurone $x_{i,j}$ actif (disque noir dans la figure 3.1) signifie que la tâche τ_i est exécutée au *slot* j. Inversement, un neurone $x_{i,j}$ inactif (disque blanc) signifie que la tâche τ_i n'est pas exécutée au *slot* j. La tâche τ_f est une tâche fictive modélisant l'inactivité du processeur, elle est donc active lorsqu'aucune tâche n'utilise la ressource d'exécution. Ainsi, les neurones associés à la tâche τ_f de la figure 3.1a sont tous actifs car aucune tâche n'a été ordonnancée.

Le nombre de *slots* d'exécution requis par une tâche τ_i est décrit par le paramètre C_i correspond au WCET . L'application utilisée en exemple dans la figure 3.1 possède trois tâches τ_1, τ_2 et τ_3 dont les WCET sont respectivement $C_1 = 3$, $C_2 = 2$ et $C_3 = 4$ et les périodes $P_1 = P_2 = P_3 = 12$. Le nombre total de *slots* est égal à 12. Dans ces travaux, nous considérons que la DFPT d'une tâche est égale à sa période. La figure 3.1.b présente un exemple de scénario d'exécution valide.

La construction d'un réseau de neurones de Hopfield associé au problème d'ordonnancement temporel est basé sur l'utilisation de la règle k-de-n. La figure 3.2 décrit l'application des règles k-de-n sur l'exemple présenté par la figure 3.1.

Une règle 1-de-4 est appliquée sur chaque colonne. Cette règle permet de contrôler le nombre de tâches exécutées à chaque *slot*. Parce que nous considérons une architecture disposant d'une seule ressource d'exécution, seule une tâche peut être exécutée par *slot*. Le paramètre k est donc égal à un et le paramètre n au nombre de tâches soumises (plus la tâche fictive τ_f).

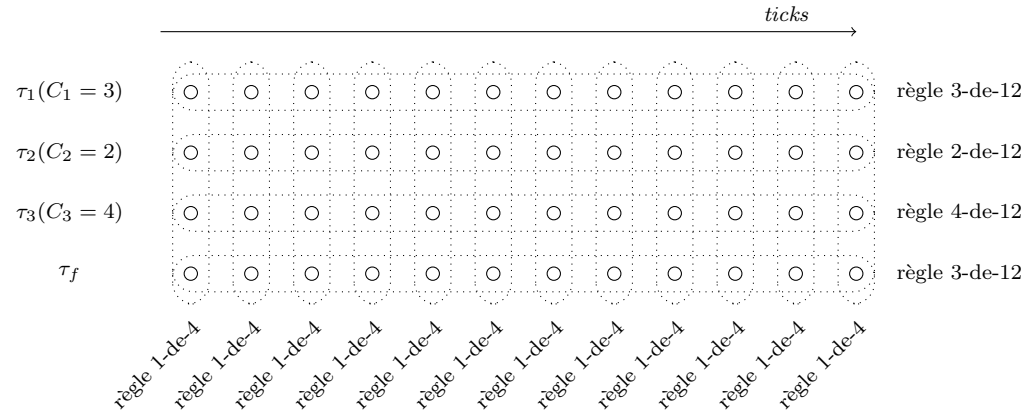


FIGURE 3.2 – Application des règles nécessaires à la construction du réseau.

Les règles k-de-n appliquées sur les lignes permettent de contrôler le nombre de *slots* nécessaires à chaque tâche. Ainsi, une règle 3-de-12 est appliquée sur la première ligne, celle correspondant à la tâche τ_1. Parce que le WCET de la tâche τ_1 est égal à trois ($C_1 = 3$), le paramètre k de cette règle est paramétré à trois. Le paramètre n est quant à lui égal au nombre de *slots* considérés. Des règles k-de-n paramétrées de façon équivalentes sont appliquées sur les autres tâches à l'exception de la tâche fictive τ_f. Par définition, la tâche fictive se voit attribuer les *slots* non utilisés par d'autres tâches. Ainsi, le WCET fictif de cette tâche est égal à

$$C_f = S - \sum_{i=0}^{t} C_i, \tag{3.1}$$

où S est égal au nombre de *slots* total sur lequel l'ordonnancement est recherché, et t au nombre de tâches. Dans l'exemple de la figure 3.2, le WCET C_f de la tâche τ_f est égal à $12 - (3 + 2 + 4) = 3$.

En résumé, pour construire un réseau de neurones de Hopfield associé au problème d'ordonnancement mono-processeur, il convient d'appliquer

- une règle C_i-de-S pour chaque tâche τ_i (dont la tâche fictive) et,
- une règle 1-de-t où t est égal au nombre de tâches, pour chaque colonne.

3.1.2 Architecture hétérogène

Le modèle précédemment développé a été étendu à des architectures multiprocesseurs hétérogènes [9]. En ce qui concerne l'ordonnancement temporel, la conséquence principale de l'exécution d'une tâche sur une architecture hétérogène est que le WCET de cette tâche n'est pas identique sur toutes les ressources. De plus, la migration d'une tâche devient très complexe car il est nécessaire de transposer le contexte d'exécution courant vers une cible d'exécution potentiellement différente. L'idée principale de notre démarche consiste à garantir qu'une tâche commençant à être exécutée sur une res-

source ne peut pas être exécutée sur une autre ressource, ce qui permet d'éviter toute migration.

Chaque cible d'exécution est représentée par un plan de neurones semblable au réseau utilisé pour l'ordonnancement mono-processeur présenté dans la section précédente. Chaque ressource est donc modélisée par $(T+1)\times S$ neurones, où T est le nombre de tâches (sans la tâche fictive) et S le nombre de *slots* considérés par l'ordonnanceur. La figure 3.3 est un exemple de modélisation considérant un nombre de *slots* égale à six ($S = 6$) et trois ressources. L'ensemble des neurones associés à une tâche τ_i pour une ressource r est noté $\epsilon_{i,r}$ et le nombre de ressources est noté R.

Une tâche n'est exécutée que sur une ressource, il est possible qu'une tâche ne soit pas exécutée sur une ressource.

Dans le cas d'une architecture homogène, une règle k-de-n était appliquée aux neurones associés à chaque tâche τ_i. Cette règle permettait une activation de C_i neurones. Dans le cas d'une architecture hétérogène, il n'est pas possible d'appliquer cette règle sur chaque ensemble $\epsilon_{i,r}$ car une tâche n'est éxecutée que sur une ressource, il existe donc des ensembles $\epsilon_{i,r}$ pour lesquels aucun neurone ne doit être activé. Ainsi, des règles 0-ou-k-de-n [9] sont appliquées sur les ensembles $\epsilon_{i,r}$. Cette règle permet de garantir que le nombre de neurones activés pour une tâche τ_i et pour une ressource p est soit égal à zéro soit égal à $C_{i,p}$, avec $C_{i,p}$ le WCET de la tâche τ_i sur la ressource p. L'application d'une règle 0-ou-k-de-n nécessite l'ajout de k neurones cachés, notés xh. Sur la figure 3.3, les neurones cachés sont placés dans les zones grises. Pour chaque ensemble $\epsilon_{i,p}$, la règle 0-ou-k-de-n nécessite l'ajout de $C_{i,p}$ neurones cachés, notés $xh_{i,p}$.

Les règles 0-ou-k-de-n appliquées aux tâches permettent de contrôler l'exécution d'une tâche ou non sur une ressource. Cependant, rien ne garantit qu'une tâche n'est exécutée que sur une seule ressource. Afin de satisfaire cette contrainte, pour chaque tâche τ_i, $max(|C_{i,j} - C_{i,k}|)\forall(j,k)$ neurones cachés sont ajoutés au modèle. Ces neurones sont notés xhg_i (pour une tâche τ_i) dans la figure 3.3. Pour chaque tâche τ_i, une règle k-de-n est appliquée sur chaque ensemble $\bigcup_{j=1}^{R} \epsilon_{i,j} \cup xhg_i$ où k est égal à $max(C_{i,j})\forall j$.

Finalement, une règle 1-de-n est appliquée sur chaque colonne afin de garantir qu'une seule tâche utilise la ressource à un *slot* donné.

Le nombre de règles nécessaire à la modélisation du problème est relativement important. Or, l'addition excessive de règles dans un réseau de neurones, qui est permise par la propriété d'additivité, peut augmenter dramatiquement le nombre de minimums locaux. Ces minimums locaux ne correspondant pas nécessairement à une solution valide, il est nécessaire de relancer la convergence du réseau afin d'obtenir une nouvelle solution. Bien entendu, des ré-initialisations successives augmentent le temps d'obtention d'une solution.

Le nombre de neurones utiles N_u, c'est à dire participant au codage de la solution est égal à

$$N_u = R \times S \times (T+1), \tag{3.2}$$

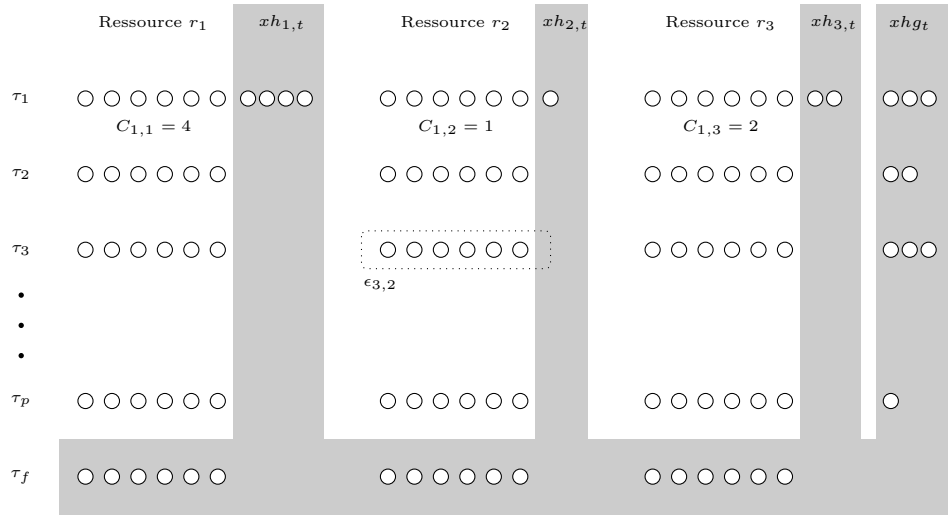


FIGURE 3.3 – Exemple d'un problème d'ordonnancement (T tâches et R ressources) modélisé par un réseau de neurones utilisant des neurones inhibiteurs (placés dans les zones grisées).

et le nombre de neurones cachés N_c à

$$N_c = \sum_{i=1}^{T} \sum_{j=1}^{R} C_{i,j} + \sum_{i=1}^{T} max(|C_{i,j} - C_{i,k}|) \forall j, k = 1, 2, ..., R. \tag{3.3}$$

Le nombre de neurones nécessaires N_n à la modélisation est donc

$$N_n = N_u + N_c. \tag{3.4}$$

L'équation (3.3) montre que le nombre de neurones croît linéairement en fonction du nombre de tâches et de ressources mais dépend également du WCET de chaque tâche sur les différentes ressources. Pour des applications complexes, l'augmentation importante du nombre de neurones peut devenir problématique, tant au niveau du temps de simulation du réseau de neurones de Hopfield que de son implémentation.

Les principales limitations de ce modèle sont donc le nombre de neurones ainsi que le nombre de ré-initialisations nécessaires à l'obtention d'une solution satisfaisante. Dans la suite, nous allons réduire le nombre de neurones en supprimant des neurones cachés et diminuer le nombre de ré-initialisations nécessaires en réduisant le nombre de règles appliquées au réseau. La section 3.3.3 présente des résultats illustrant précisément ces améliorations.

3.2 Simplification du réseau : neurones inhibiteurs

Les limites de la proposition précédemment présentée proviennent principalement du fait qu'une tâche ne doit être exécutée que sur une ressource. Dans cette section,

une autre modélisation du problème est utilisée afin de réduire le nombre de neurones cachés ainsi que le nombre de règles appliquées au réseau. Cette modélisation utilise des neurones particuliers, nommés *neurones inhibiteurs*. Le rôle des neurones inhibiteurs consiste à détecter l'activation d'une tâche sur une ressource puis à empêcher l'activation de cette même tâche sur une autre ressource [9].

3.2.1 Définition d'un neurone inhibiteur

Le principe d'un neurone inhibiteur est de provoquer la désactivation de neurones lorsqu'il est activé. Il reçoit de l'énergie d'un ensemble de neurones, et lorsque l'énergie reçue par un neurone inhibiteur est suffisante pour l'activer, il émet alors de l'énergie négative capable de désactiver des neurones auxquels il est connecté.

Il est important de noter que l'évaluation d'un neurone inhibiteur est identique à celle d'un neurone standard. La différence entre un neurone classique et un neurone inhibiteur se manifeste par la manière dont il est connecté au réseau. En fait, les neurones inhibiteurs introduisent des connexions asymétriques au sein du réseau de neurones.

Afin de présenter plus en détail les neurones inhibiteurs, quelques notations sont introduites. Les S neurones d'une tâche τ_i sur une ressource R_j sont regroupés dans un ensemble noté $\epsilon_{i,j} = \{x_{i,j,1}, x_{i,j,2}, \ldots, x_{i,j,S}\}$. La figure 3.4 présente un exemple d'un réseau modélisant une tâche τ_i et une architecture disposant de deux ressources d'exécution j et l.

Phase d'activation d'un neurone inhibiteur. Un neurone inhibiteur $xh_{i,j}$ est associé à chaque ensemble $\epsilon_{i,j}$. Afin d'activer le nombre de neurones nécessaires à une tâche, une règle k-de-n est appliquée à chaque ensemble $\epsilon_{i,j}$ avec $k = C_{i,j}$. La valeur de seuil $th_{i,j}$ d'un neurone inhibiteur $xh_{i,j}$ est paramétrée à $th_{i,j} = -C_{i,j}$. Les neurones d'un ensemble $\epsilon_{i,j}$ sont connectés au neurone $xh_{i,j}$ avec un poids de connexion égal à 1. Lorsque $C_{i,j}$ neurones de l'ensemble $\epsilon_{i,j}$ sont actifs, le neurone $xh_{i,j}$ peut être activé. L'équation d'évaluation du neurone $xh_{i,j}$ est alors

$$xh_{i,j} = H(\sum_{k=1}^{N} x_{i,j,k} w_{(x_{i,j,k}, xh_{i,j})} - th_{i,j}) \tag{3.5}$$

où $x_{i,j,k}$ est l'état du neurone $x_{i,j,k}$, $w_{(x_{i,j,k}, xh_{i,j})}$ la connexion du neurone $x_{i,j,k}$ vers le neurone inhibiteur $xh_{i,j}$. Comme le poids des connexions $w_{(x_{i,j,k}, xh_{i,j})}$ est toujours égal à 1, l'équation (3.5) peut être réécrite ainsi

$$xh_{i,j} = H(\sum_{k=1}^{N} x_{i,j,k} - C_{i,j}) \tag{3.6}$$

La règle k-de-n appliquée à l'ensemble $\epsilon_{i,j}$ mène à l'activation de $C_{i,j}$ neurones, et l'équation (3.6) montre bien que le neurone $xh_{i,j}$ est activé si $C_{i,j}$ neurones appartenant à l'ensemble $\epsilon_{i,j}$ sont activés. Le neurone $xh_{i,j}$ est donc bien activé lorsque la tâche τ_i s'est vue attribuer un nombre de *slots* suffisants.

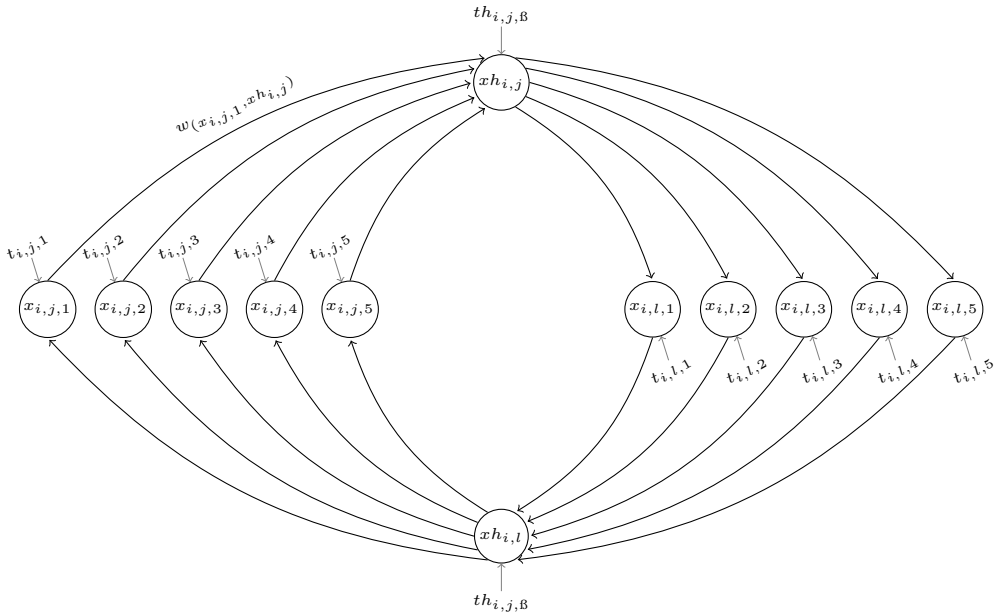


FIGURE 3.4 – Réseau de neurones comportant deux tâches décrites chacune par cinq neurones ainsi que deux neurones inhibiteurs $xh_{i,j}$ et $xh_{i,l}$. Le neurone $xh_{i,j}$ est activé par les neurones $x_{i,j,k}$ pour tout $k \in 1, \ldots, 5$ et inhibe les neurones $x_{i,l,k}$ pour tout $k \in 1, \ldots, 5$.

Phase d'inhibition. Les neurones inhibiteurs ont pour rôle d'inhiber des neurones suite à leur activation. Dans notre contexte, un neurone $xh_{i,j}$ permet d'inhiber les neurones associés aux instances de la tâche τ_i sur les ressources $k, \forall k \neq j$. Sur la figure 3.4, le neurone $xh_{i,j}$ doit inhiber les neurones $x_{i,l,k}, \forall k = 1, \ldots, 5$. Les poids des connexions $w_{xh_{i,j},x_{i,l,k}}, \forall k = 1, \ldots, 5$ doivent être paramétrés à une valeur négative suffisamment importante pour désactiver les neurones $x_{i,l,k}, \forall k = 1, \ldots, 5$. D'après l'équation d'évaluation d'un neurone (2.7), pour désactiver le neurone $x_{i,l,k}$, il faut que

$$w_{(xh_{i,j},x_{i,l,k})} < -t_{i,l,k}. \tag{3.7}$$

Afin d'inhiber un neurone x_i, le poids des connexions provenant d'un neurone inhibiteur et à destination du neurone x_i doit être strictement inférieur à la valeur de seuil t_i du neurone x_i.

3.2.2 Application à l'ordonnancement

Dans la section précédente, le fonctionnement d'un neurone inhibiteur a été présenté sur un exemple simple. Il s'agit désormais de montrer comment ils peuvent être utilisés dans le cadre de l'ordonnancement.

La figure 3.5 présente un exemple comprenant T tâches et R ressources. Les neurones placés dans la zone grisée sont des neurones inhibiteurs. À chaque ressource r sont associés T neurones inhibiteurs noté $xh_{r,t}$. Ainsi, lorsqu'une tâche se voit attribuer un

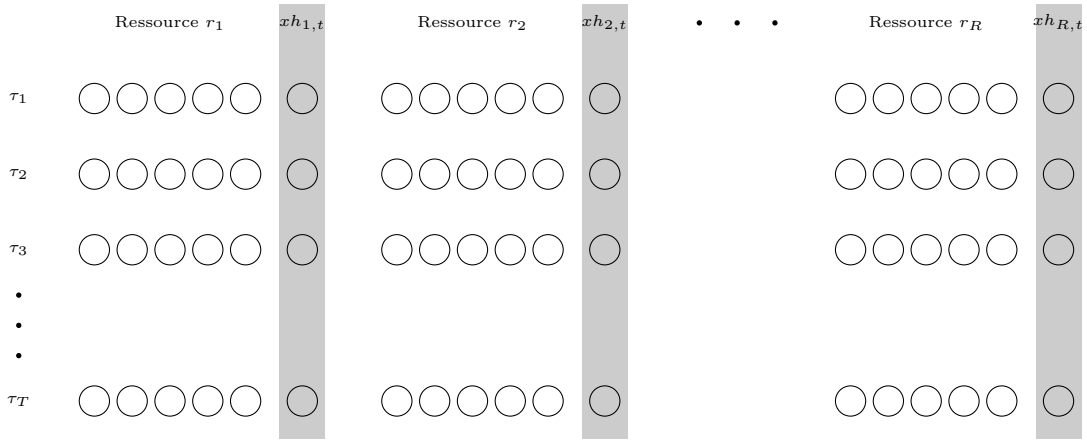


FIGURE 3.5 – Exemple d'un problème d'ordonnancement (T tâches et R ressources) modélisé par un réseau de neurones utilisant des neurones inhibiteurs (placés dans les zones grisées).

nombre suffisant de *slots*, le neurone inhibiteur lui étant associé est activé. Ce neurone inhibiteur inhibe alors tous les neurones associés aux autres implémentations de cette tâche.

L'utilisation de neurones inhibiteurs simplifie considérablement la construction du réseau par rapport au modèle présenté dans la section 3.1.2. La construction du réseau requiert alors :

- une règle k-de-n sur chaque ensemble $\epsilon_{i,j}$ paramétrée avec $k = C_{i,j}$ et $n = S$;
- une règle 1-de-T appliquée sur les neurones inhibiteurs $(xh_{i,j}, \forall j)$ associées à chaque tâche τ_i ;
- la valeur de seuil d'un neurone inhibiteur est paramétrée à $th_{i,j} = -C_{i,j}$;
- les connexions des neurones inhibiteurs sont paramétrées à
  - $w_{(xh_{i,j},x_{i,j,1})} < th_{i,j} \forall i, j, k, l | j \neq l$,
  - $w_{(x_{i,j,1},xh_{i,j})} = 1$

3.2.3 Convergence d'un réseau pourvu de neurones inhibiteurs

Les connexions associées aux neurones inhibiteurs illustrées par la figure 3.4, sont asymétriques. Un neurone $xh_{i,j}$ reçoit des connexions des neurones associés à la tâche τ_i mais n'émet aucune énergie vers ces neurones. De même, les connexions en provenance des neurones inhibiteurs ne sont pas symétriques. Or, lors de la preuve de la convergence d'un réseau de Hopfield dans la section 2.3.2, la symétrie des connexions a été utilisée afin de montrer la décroissance de la fonction d'énergie.

La figure 3.6 présente l'évolution de l'énergie d'un réseau disposant de neurones inhibiteurs. Nous pouvons constater que l'énergie n'est pas strictement décroissante. Cette figure montre que la méthode de Lyapunov utilisant la fonction d'énergie définie par l'équation (2.11) est inapplicable sur un réseau disposant de neurones inhibiteurs.

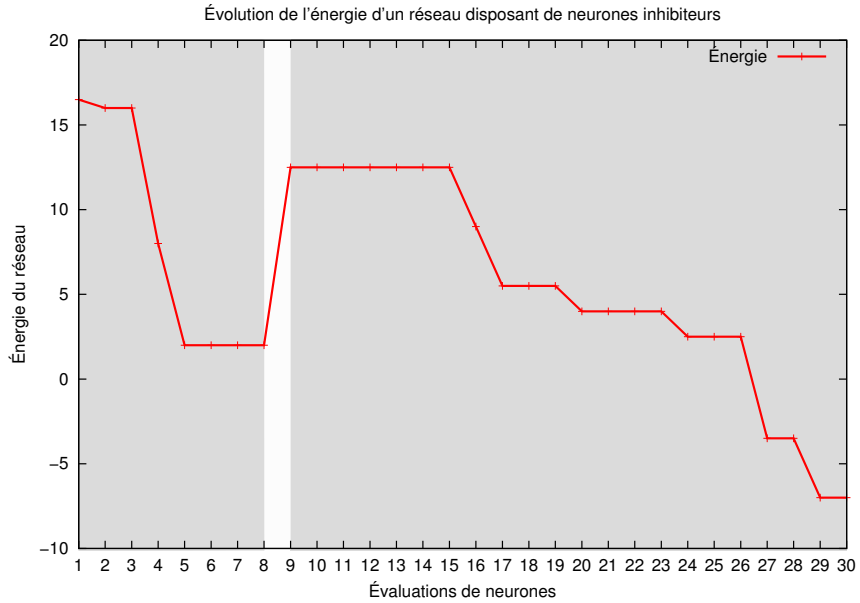


FIGURE 3.6 – Évolution de l'énergie d'un réseau disposant de neurones inhibiteurs. L'énergie du réseau n'est pas strictement décroissante au fil des évaluations de neurones. La première partie grisée - $[1, 8]$ - correspond à la phase d'activation des neurones. La zone blanche correspond à l'activation d'un neurone inhibiteur. La seconde partie - $[9, 30]$ - grisée correspond à la convergence du réseau suite à l'activation des neurones inhibiteurs.

Cependant, la décroissance de la fonction d'énergie est une condition suffisante mais pas nécessaire pour garantir la convergence du réseau.

Lors des multiples simulations effectuées, le réseau a toujours convergé et ce, malgré la présence de connexions non symétriques. Bien que nous ne soyons pas en mesure d'établir mathématiquement la convergence du réseau, nous pouvons l'expliquer intuitivement à l'aide des étapes ci-dessous.

1. Conditions initiales : Les neurones inhibiteurs sont initialisés à l'état inactif et l'état des autres neurones est déterminé aléatoirement.

$$xh_{i,j} = 0 \ \forall i, j, \tag{3.8}$$
$$x_{i,j,k} = random(0, 1) \ \forall i, j, k, \tag{3.9}$$

   où $random(0, 1)$ est une fonction retournant 0 ou 1 aléatoirement.

2. Première étape de convergence : les neurones des ensembles $\epsilon_{i,j}$ (correspondant à la modélisation de la tâche τ_i sur la ressource R_j) vont évoluer de façon à activer $C_{i,j}$ neurones (à cause de l'application des règles k-de-n sur ces ensembles). Cette étape correspond à la première zone grisée de la figure 3.6.
3. Étape d'inhibition : lorsque le nombre de neurones actif d'un ensemble $\epsilon_{i,j}$ est

égal à $C_{i,j}$, l'évaluation du neurone inhibiteur $xh_{i,j}$ mènera à son activation. Cette étape correspond à la zone blanche de la figure 3.6.

4. Seconde étape de convergence : lorsqu'un neurone $xh_{i,j}$ est actif, il inhibe les neurones des ensembles $\epsilon_{i,r} \forall r \neq j$. Ainsi, lorsqu'un neurone appartenant à l'un de ces ensembles est évalué, il est désactivé par le poids de la connexion provenant du neurone inhibiteur. Cette étape correspond à la seconde zone grisée de la figure 3.6.

   Cette étape de convergence permet donc de garantir qu'une tâche n'est exécutée que sur une ressource.

3.2.4 Réduction du nombre de neurones cachés

3.2.4.1 Réduction par les neurones inhibiteurs

L'utilisation de neurones inhibiteurs a réduit de façon importante le nombre de neurones par rapport à la proposition initiale d'ordonnancement sur architectures hétérogènes.

Dans le cas de l'utilisation de neurones inhibiteurs, le nombre de neurones nécessaires N'_n est égale à

$$N'_c = T \times R \tag{3.10}$$
$$N'_n = N_u + N'_c \tag{3.11}$$

où N'_c est le nombre de neurones cachés. Le nombre de neurones cachés N'_c n'est donc fonction que du nombre de tâches et de ressources. Plus précisément, le nombre de neurones cachés pour chaque tâche et pour chaque ressource est égale à un. Le modèle utilisé dans [9] requiert un nombre de neurones cachés, exprimés par l'équation (3.3), fonction du WCET des tâches. Alors que la méthode utilisant les neurones inhibiteurs ajoute un neurone caché à chaque ensemble $\epsilon_{i,j}$, la précédente méthode en ajoutait $C_{i,j}$.

3.2.4.2 Suppression des tâches fictives

Le modèle proposé par Cardeira [12] illustré par la figure 3.7 utilise des tâches fictives pour modéliser l'inactivité du processeur. Dans le cas d'une architecture disposant de β unités d'exécution, β tâches fictives sont ajoutées au réseau. Ainsi, l'architecture considérée par la figure 3.7 possède trois unités d'exécutions, dont l'inactivité est modélisée par les tâches fictives τ_{f_1}, τ_{f_2} et τ_{f_3}. Les neurones associés aux tâches fictives sont en fait les neurones cachés de règles au-plus-k-de-n, telles que définies à la section 2.5.2, appliquées à chaque *slot*. Les neurones associés aux tâches fictives nécessitent des ressources et des évaluations supplémentaires. Parce qu'ils ne participent pas directement au codage de la solution, nous proposons de les supprimer en modifiant les règles au-plus-k-de-n appliquées à ce réseau.

Dans notre contexte, l'objectif des règles au-plus-k-de-n est d'assurer que le nombre de tâches exécutées simultanément n'est pas supérieur au nombre d'unités d'exécution. Lorsqu'un neurone $x_{i,j}$ est activé, cela signifie que la tâche τ_i est exécutée au *slot* j.

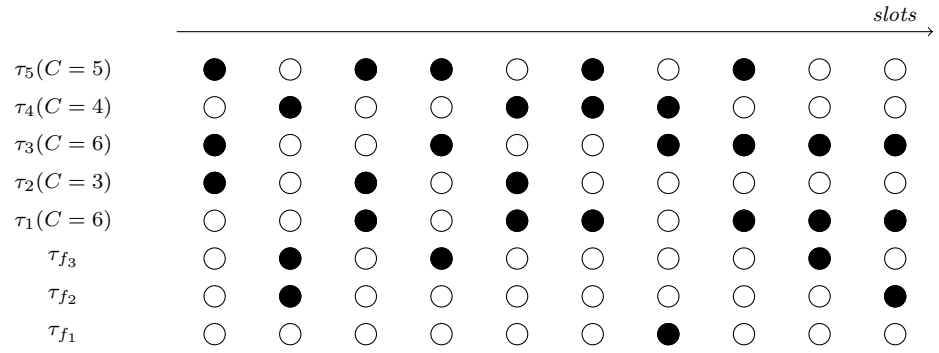


FIGURE 3.7 – Exemple présentant un scénario d'évaluation d'un réseau de neurones modélisant l'ordonnancement de cinq tâches sur une architecture disposant de trois unités d'exécution. À chaque *slot*, le nombre de neurones activés est égal au nombre d'unités d'exécution. Les tâches fictives τ_{f_1}, τ_{f_2} et τ_{f_3} permettent de modéliser l'inactivité des unités d'exécution.

L'activation du neurone $x_{i,j}$ est due à la règle k-de-n appliquée sur la tâche τ_i, et d'après l'équation (2.21), lorsque k neurones d'une tâche sont activés, l'énergie des neurones actifs est égale 1. Un apport d'énergie négative strictement inférieure à -1 est donc suffisant pour désactiver un tel neurone. Les connexions visant à assurer que le nombre de tâches exécutées à chaque *slot* n'est pas supérieur au nombre de ressources d'éxecution doivent donc satisfaire les deux contraintes suivantes :

- si β neurones associés à un *slot* sont activés, l'énergie apportée par les connexions de ces neurones doit être strictement inférieure à -1 ;
- si moins de β neurones associés à un *slot* sont activés, l'énergie apportée par les connexions de ces neurones doit être supérieure ou égale à -1 afin de ne pas perturber les règles k-de-n appliquées sur les tâches.

Les valeurs des connexions w_s entre les neurones appartenant à un même *slot* sont alors définies par

$$w_s = -\frac{1}{\beta - \alpha}, \tag{3.12}$$

où $\alpha \in]0,1[$ et β le nombre d'unités d'exécution présentes sur l'architecture considérée.

La figure 3.8 présente un scénario d'évaluation d'un réseau de neurones modélisant l'ordonnancement de cinq tâches sur une architecture disposant de trois unités d'exécution. Les neurones noirs sont des neurones actifs, les neurones blancs sont inactifs et les gris sont des neurones inactifs dont nous allons analyser l'évaluation. Des règles k-de-n ont été appliquées sur chaque tâche et des connexions avec un poids w_s égal à $w_s = -\frac{1}{3-0.5}$ ont été appliquées entre les neurones appartenant à un même *slot* (le paramètre α de l'équation (3.12) a été arbitrairement fixé à 0.5).

Nous supposons que le prochain neurone évalué est le neurone $x_{3,3}$ (associé à la tâche τ_3 et au *slot* 3). À ce stade, la tâche s'est vue attribuer cinq *slots* alors que son WCET est égal à six. En ne considérant que les connexions apportées par la règle k-de-n appliquée aux neurones de la tâche τ_3, d'après l'équation (3.13), l'énergie du

neurone $x_{3,3}$ est égale à un.

$$\begin{aligned} E(x_{3,3})_{k\text{-de-}n} &= \sum_{i=1}^{10} x_{3,i} * w_{x_{3,i},x_{3,3}} + (2 * k - 1) \\ &= \sum_{i=1}^{10} x_{3,i} * -2 + (2 * 6 - 1) = -10 + 11 = 1 \end{aligned} \tag{3.13}$$

Les connexions appliquées sur les neurones du *slot* 3 apportent une énergie au neurone $x_{3,3}$ égale à

$$\begin{aligned} E(x_{3,3})_{slot} &= \sum_{i=1}^{5} w_s \times x_{i,3} \\ &= w_s \times (x_{1,3} + x_{2,3} + x_{5,3}) \\ &= -\frac{1}{3 - 0.5} \times 3 = -1.2. \end{aligned} \tag{3.14}$$

L'énergie totale du neurone $x_{3,3}$ est égale à

$$E(x_{3,3}) = E(x_{3,3})_{slot} + E(x_{3,3})_{k\text{-de-}n} = -0.2.$$

Ce neurone n'est donc pas activé, ce qui correspond bien au comportement attendu étant donné que l'architecture ne dispose que de trois unités d'exécution.

Lorsque le neurone $x_{3,10}$ est évalué, l'énergie provenant de la règle k-de-n est égale à un et l'énergie provenant des connexions appliquées aux neurones du *slot* 10 est égale à

$$-\frac{1}{3 - 0.5} \times 1 = -0.4.$$

L'énergie totale de ce neurone est donc égale à $1 - 0.4 = 0.6$, ce qui est suffisant pour l'activer, et qui satisfait bien la contrainte indiquant que trois tâches peuvent être ordonnancées sur ce *slot*.

L'exemple illustré par la figure 3.8 a permis d'expliquer le fonctionnement de cette nouvelle règle au-plus-k-de-n. Cette règle permet de garantir qu'au plus k neurones parmi n sont activés sans ajouter de neurones cachés. En utilisant la règle au-plus-k-de-n classique, cet exemple aurait requis 30 neurones cachés (10 *slots* et 3 unités d'exécution) et 50 neurones codant la solution, soit 80 neurones au total. La réduction du nombre de neurones apportée par cette nouvelle règle au-plus-k-de-n est donc égale à 27% pour l'exemple de la figure 3.8.

3.3 Résultats

3.3.1 Comparaison avec des approches neuronales classiques

Le premier exemple présente l'ordonnancement de trois tâches τ_1,τ_2 et τ_3 sur une architecture disposant d'une seule unité d'exécution. Le modèle proposé par Cardeira et al.

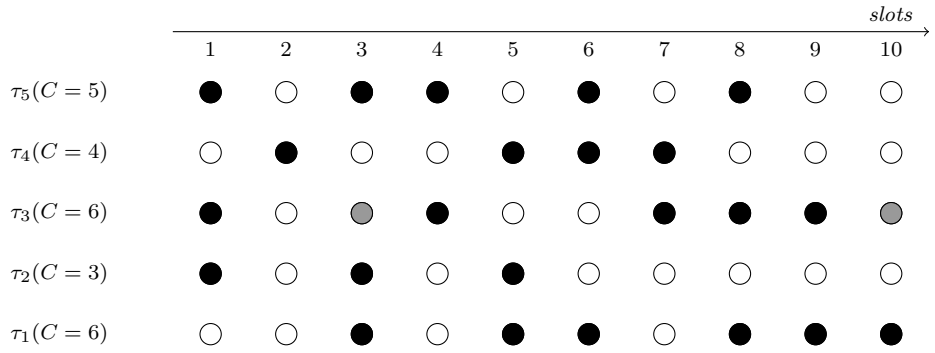


FIGURE 3.8 – Exemple présentant un scénario d'évaluation d'un réseau de neurones modélisant l'ordonnancement de cinq tâches sur une architecture disposant de trois unités d'exécution. Les neurones noirs sont des neurones actifs, les neurones blancs sont inactifs et les gris sont des neurones inactifs dont nous analysons l'évaluation.

dans [11] utilise des règles au-plus-k-de-n classiques. Il est donc nécessaire d'ajouter des neurones cachés modélisant la tâche fictive τ_f. La figure 3.9.a représente un état initial aléatoire du réseau proposé par Cardeira et al. et la figure 3.9.b une solution obtenue suite à l'évaluation de ce réseau. Les auteurs de [11] indique qu'il est nécessaire d'effectuer plus de 600 évaluations de neurones pour atteindre un état stable et donc obtenir une solution. La figure 3.9.c présente notre modélisation du même problème d'ordonnancement. Parce que nous n'utilisons pas de neurones cachés pour réaliser la règle au-plus-k-de-n, notre modélisation nécessite moins de neurones que le modèle classique et donc converge après un nombre beaucoup plus faible d'itérations. À partir d'un état initial, le nombre moyen d'évaluations requis pour obtenir une solution est égale à 79.

La figure 3.10 présente un exemple de problème d'ordonnancement de tâche sur une architecture disposant de deux unités d'exécutions. Cette figure compare le modèle neuronal classique ainsi que notre modèle dépourvu de neurones cachés. Dans les deux cas, les réseaux sont présentés dans un état initial quelconque. La figure 3.10.a présente le modèle classique où il a été nécessaire d'ajouter deux tâches fictives τ_{f_1} et τ_{f_2}. Lors de simulations de ce réseau, plus de 300 évaluations de neurones ont été nécessaires pour obtenir une solution. La figure 3.10.b présente quant à elle notre modélisation du même problème. Dans ce cas, la convergence du réseau est atteinte en effectuant environ 70 évaluations neuronales.

3.3.2 Comparaison avec l'algorithme Pfair

Cette section présente des comparaisons de notre algorithme avec l'algorithme Pfair [6, 5] prouvé optimal dans un contexte d'ordonnancement de tâches périodiques sur une architecture disposant de plusieurs unités d'exécution équivalentes. L'algorithme Pfair ne permettant que d'ordonnancer des tâches sur une architecture homogène, nous utilisons notre algorithme dans un contexte homogène bien qu'il soit principalement conçu pour être utilisé sur des architectures hétérogènes.

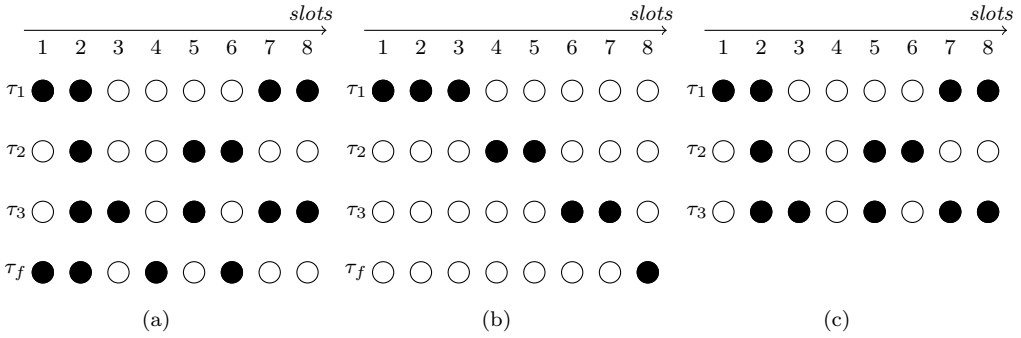


FIGURE 3.9 – Exemple de réseaux modélisant l'ordonnancement de trois tâches (τ_1, τ_2 et τ_3) sur une architecture disposant d'une seule unité d'exécution. Les WCET des tâches τ_1, τ_2 et τ_3 sont respectivement égaux à 3, 2 et 2. (a) présente le réseau issu de la modélisation classique dans un état initial aléatoire. (b) présente ce même réseau après son évaluation. (c) présente le réseau neuronal issu de notre modélisation dans un état initial aléatoire. Sa convergence est semblable à (b).

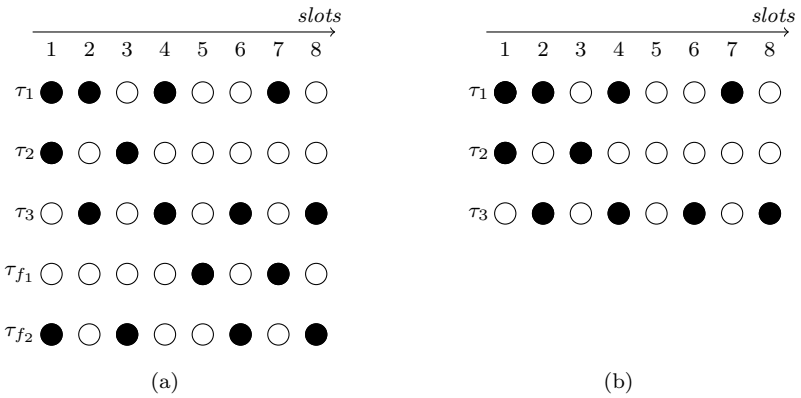


FIGURE 3.10 – Exemple d'ordonnancement de trois tâches sur une architecture disposant de deux unités d'exécution. (a) est le réseau issu de la modélisation classique dans un état quelconque. (b) est le réseau dépourvu de neurones cachés.

Un inconvénient majeur de l'algorithme Pfair est le nombre de préemptions et de migrations de tâches qu'il engendre. Notre algorithme garantit, quant à lui, l'absence de migrations de tâches. En contrepartie, certains jeux de tâches ne peuvent pas être ordonnancés par notre algorithme bien qu'ils le soient avec Pfair.

3.3.2.1 Résultats préliminaires

Afin de présenter quelques résultats préliminaires, nous utilisons un jeu de huit tâches indépendantes dont les caractéristiques sont

$$\{\tau_i, C_i\} = \{(\tau_1, 4), (\tau_2, 5), (\tau_3, 3), (\tau_4, 7), (\tau_5, 9), (\tau_6, 6), (\tau_7, 8), (\tau_8, 4)\}\,. \tag{3.15}$$

De plus, nous considérons une architecture disposant de quatre unités d'exécution, une période d'ordonnancement égale à 20 *slots* durant laquelle chaque tâche doit être exécutée une fois.

La figure 3.11.a présente le scénario d'ordonnancement des tâches obtenu par l'algorithme Pfair, la figure 3.11.b un scénario obtenu par notre algorithme. Dans les deux cas, les scénarios obtenus sont valides car toutes les tâches ont été ordonnancées sur une période de 20 *slots*.

La figure 3.11.a montre que le nombre de migrations de tâches engendré est relativement important. Par exemple, la tâche τ_4 est exécutée sur la ressource R_1 au *slot* 0, puis sur la ressource R_2 au *slot* 5, puis sur la ressource R_1 au *slot* 11. Dans le scénario présenté par la figure 3.11.a, le nombre total de migrations engendrées par l'algorithme Pfair est égal à 16 (0 pour τ_1, 1 pour τ_2, 1 pour τ_3, 2 pour τ_4, 2 pour τ_5, 5 pour τ_6, 5 pour τ_7, 0 pour τ_8). Les multiples migrations sont l'inconvénient majeur de l'algorithme Pfair. Ce problème est connu et a donné lieu à modifications de l'algorithme Pfair, notamment dans [3].

Par construction, notre réseau de neurones produit un ordonnancement dépourvu de migrations. Cette propriété est assurée par les neurones inhibiteurs garantissant qu'une tâche n'est exécutée que sur une seule ressource. Le scénario présenté par la figure 3.11.b conforte cette assertion.

Concernant le nombre de préemptions, les deux algorithmes produisent des préemptions. Le nombre de préemptions dans le scénario produit par l'algorithme Pfair est égal à 35, alors que le scénario généré par notre algorithme n'en possède que 21.

Remarque. Dans notre contexte d'expérimentation, il est possible d'effectuer une passe de compactage de la solution générée afin d'exécuter les tâches au plus tôt. La figure 3.12 présente le scénario issu de la figure 3.11.b en version compactée. Parce que notre algorithme ne produit pas de migrations de tâches, il est très simple d'effectuer ce compactage, contrairement aux scénarios produits par l'algorithme Pfair. En effet, les tâches pouvant être exécutées sur des unités d'exécution différentes, il serait nécessaire de s'assurer qu'une tâche ne soit pas simultanément exécutée sur deux unités d'exécution différentes.

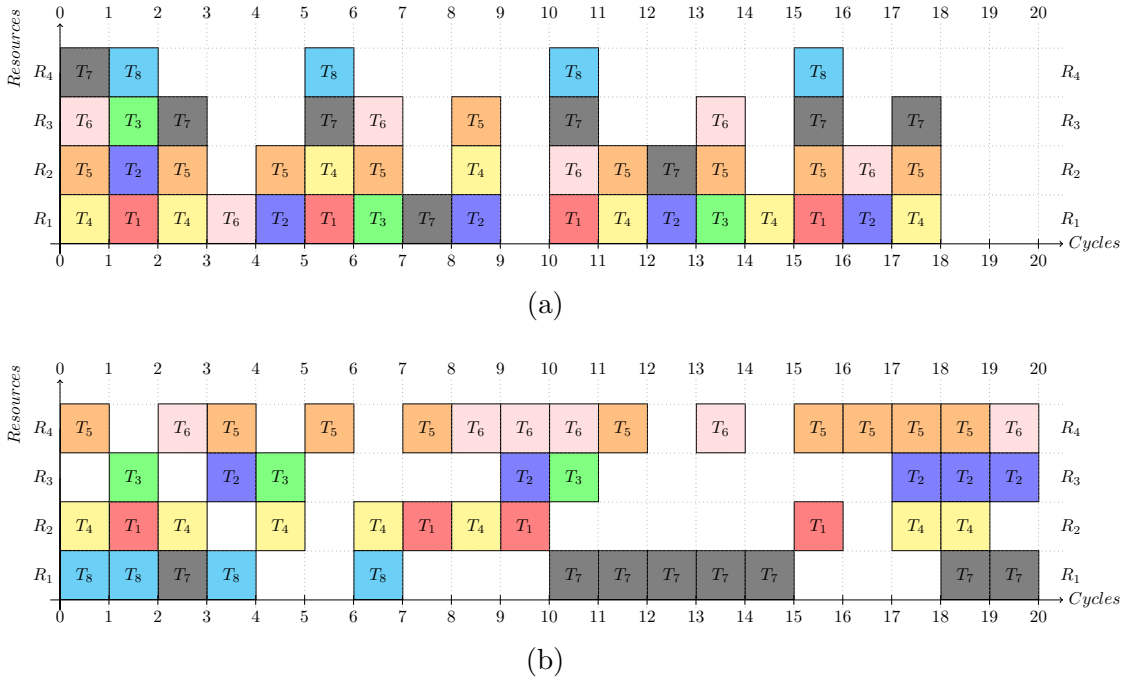


FIGURE 3.11 – Diagramme de Gantt de l'exécution des huit tâches décrites par l'équation (3.15) sur une architecture disposant de quatre unités d'exécution. (a) Solution obtenue par l'algorithme Pfair. (b) Solution obtenue par notre algorithme.

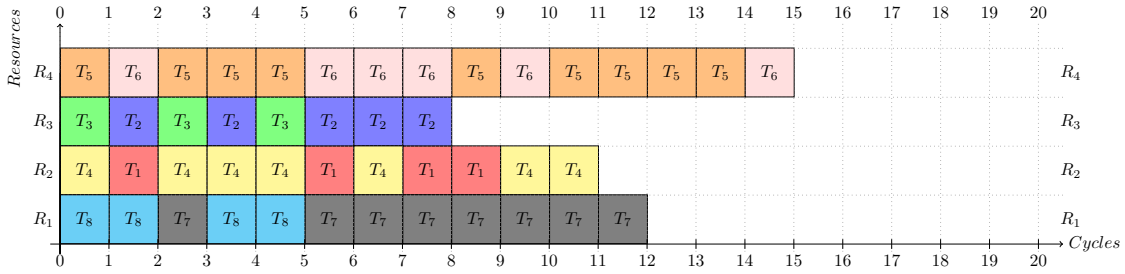


FIGURE 3.12 – Scénario illustré par la figure 3.11.b après une passe de compactage.

3.3.2.2 Comparaison avec des jeux de tâches aléatoirement générés

Dans cette section, nous comparons notre proposition avec l'algorithme Pfair sur des jeux de tâches générés aléatoirement par l'outil TGFF [22]. Chaque jeu contient 30 tâches indépendantes, et la période d'ordonnancement S est fixée à $S = 100$ *slots*. Le WCET de chaque tâche est déterminé aléatoirement par TGFF dans l'intervalle $[5\ ;\ 15]$. L'architecture considérée dans ces expérimentations possède cinq unités d'exécutions.

L'algorithme Pfair originel [6] engendre de nombreuses migrations de tâches. Les

auteurs de [3] proposent des heuristiques visant à diminuer le nombre de préemptions et de migrations. Nous avons implémenté l'heuristique nommée *H2* dans [3].

La table 3.1 présente les résultats de l'ordonnancement de 40 jeux de tâches générés par TGFF (le jeu i est noté G_i). Les deux première colonnes du tableau indiquent le nom du jeu et le pourcentage d'utilisation P qu'il induit sur l'architecture. Le pourcentage d'utilisation d'un jeu G_i est donné par l'expression

$$P_{G_i} = \frac{\sum C_{\tau_j}}{S \times R} \times 100, \forall \tau_j \in G_i, \tag{3.16}$$

où R est le nombre d'unités d'exécution et S la période d'ordonnancement choisie.

Les colonnes suivantes présentent les résultats des deux algorithmes pour chaque jeu de tâches G_i. Les résultats de chacun de ces algorithmes sont détaillés dans trois colonnes. La première donne le pourcentage de tâches placées, la seconde, le nombre de préemptions et la troisième le nombre de migrations. La dernière ligne présente les moyennes des différents résultats.

Pour tous les jeux de tâches, l'algorithme Pfair produit un scénario où toutes les tâches sont ordonnancées. En moyenne, notre réseau de neurones produit un scénario d'ordonnancement où 91% des tâches sont ordonnancées, cependant le nombre de ré-initialisations nécessaires à l'obtention d'un scénario complet est faible (voir la section 3.3.3). Le nombre moyen de préemptions générées par l'algorithme Pfair avec l'heuristique *H2*, est égal à 306 alors que celui de notre réseau est d'environ 340, ce qui représente une augmentation d'environ 10%. En revanche, notre réseau de neurones ne produit aucune migration de tâches, alors qu'en moyenne l'algorithme Pfair en produit 100 par jeu de tâches.

Ces résultats montrent que notre algorithme d'ordonnancement produit des résultats d'une qualité relativement proche d'un algorithme optimal d'ordonnancement de tâches sur architecture homogène. Les migrations produites par l'algorithme Pfair le rendent très difficilement applicable dans un contexte d'architecture hétérogène. Dans les sections suivantes, nous présentons des résultats de notre algorithme sur une architecture hétérogène.

3.3.3 Résultats sur une architecture hétérogène

Dans cette section, nous comparons les gains obtenus par notre modèle par rapport au modèle initialement présenté dans [9]. Nous considérons une architecture disposant de deux unités d'exécution différentes. La table 3.2 présente les caractéristiques du jeu de tâches utilisés dans cette section. Les tâches ont un WCET différents sur chaque unité d'exécution.

Tâche	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7
$C_{\tau_i,1}$	1	2	4	3	4	3	2
$C_{\tau_i,2}$	2	1	2	5	6	2	3

TABLE 3.2 – WCET de chaque tâche sur les deux ressources.

Jeu de tâches	Pourcentage d'utilisation des ressources	PFair avec l'heuristique *H2*			Réseau de neurones		
		Pourcentage de tâches placées	Nombre de préemptions	Nombre de migrations	Pourcentage de tâches placées	Nombre de préemptions	Nombre de migrations
G_1	75.0 %		308	146	95.9 %	338.7	
G_2	74.5 %		306	142	97,7 %	336.5	
G_3	71.3 %		290	115	99.5 %	330.8	
G_4	76.8 %		315	86	87.9 %	345.5	
G_5	77.3 %		317	110	92.1 %	343.4	
G_6	68.0 %		278	82	100.0 %	322.1	
G_7	73.8 %		299	85	96.8 %	335.9	
G_8	76.8 %		315	114	90.7 %	343.7	
G_9	81.0 %		331	120	64.5 %	355.8	
G_{10}	71.3 %		293	115	97.1 %	329.9	
G_{11}	82.0 %		335	109	64.5 %	358.3	
G_{12}	65.3 %		266	42	100.0 %	318.1	
G_{13}	72.3 %		297	101	98.6 %	331.3	
G_{14}	75.8 %		311	86	91.1 %	341.5	
G_{15}	72.8 %		297	90	98.9 %	334.5	
G_{16}	67.3 %		277	98	100.0 %	322.7	
G_{17}	75.3 %		309	97	98.1 %	336.8	
G_{18}	78.0 %		320	113	83.7 %	348.4	
G_{19}	77.3 %		317	113	85.8 %	345.9	
G_{20}	75.5 %	100 %	310	94	59.5 %	341.2	0
G_{21}	74.0 %		304	88	96.9 %	334.5	
G_{22}	70.3 %		288	94	100.0 %	327.4	
G_{23}	73.8 %		303	99	96.5 %	337.3	
G_{24}	78.8 %		321	85	80.5 %	350.1	
G_{25}	69.0 %		284	96	99.7 %	325.5	
G_{26}	73.8 %		303	92	98.2 %	334.3	
G_{27}	74.0 %		304	99	95.4 %	338.1	
G_{28}	67.0 %		272	112	100.0 %	320.6	
G_{29}	73.8 %		300	94	97.9 %	336.3	
G_{30}	79.3 %		323	95	73.1 %	352.4	
G_{31}	80.0 %		328	86	73.4 %	353.5	
G_{32}	78.3 %		319	85	79.6 %	350.1	
G_{33}	75.3 %		306	113	90.1 %	341.5	
G_{34}	79.3 %		325	101	81.6 %	349.7	
G_{35}	76.8 %		315	100	89.5 %	343.8	
G_{36}	75.3 %		307	123	95.9 %	336.8	
G_{37}	78.5 %		322	71	80.6 %	348.7	
G_{38}	76.0 %		308	103	90.7 %	341.1	
G_{39}	78.0 %		320	143	86.2 %	347.4	
G_{40}	74.0 %		303	90	94.9 %	336,82	
Moyenne	74.8 %	100 %	306.2	100.7	90.95 %	339.6	0

TABLE 3.1 – Comparaisons entre l'algorithme Pfair et notre réseau de neurones.

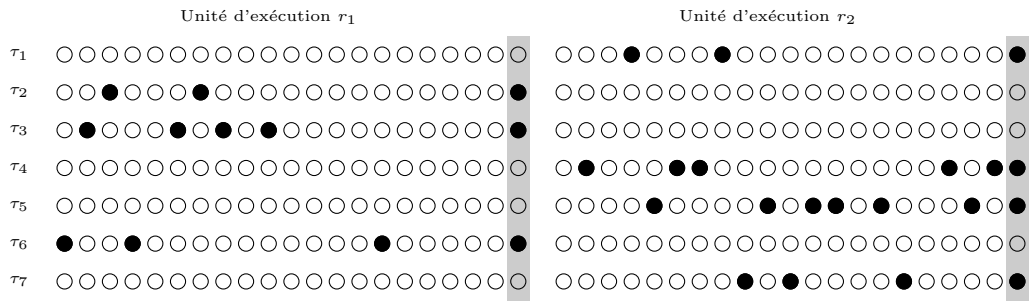


FIGURE 3.13 – Exemple d'une solution générée par notre réseau de neurones. Les sept tâches décrites par la table 3.2 sont ordonnancées sur une architecture disposant de deux unités d'exécution. La période d'ordonnancement est égale à 20 *slots*. Les zones grisées contiennent les neurones inhibiteurs.

La figure 3.13 présente un scénario d'ordonnancement du jeu de tâches décrit par la table 3.2 généré par notre réseau de neurones. Nous pouvons constater qu'une tâche est entièrement exécutée sur la même unité d'exécution, cette propriété étant assurée par les neurones inhibiteurs placés dans la zone grisée.

Nous étudions maintenant les comportements de notre réseau, ainsi que d'un réseau classique (dépourvu de neurones inhibiteurs) sur des jeux de tâches variant de deux à sept tâches. Les tâches utilisées sont celles décrites par la table 3.2. Les tables 3.3 et 3.4 présentent les caractéristiques de réseaux associés à différents jeux de tâches ainsi que le nombre d'évaluations nécessaires à l'obtention d'une solution.

La table 3.3 montre que le nombre de neurones nécessaire à la modélisation du problème est bien plus élevé que le nombre de neurones utiles, c'est à dire le nombre de neurones représentant la solution. La troisième ligne de ces tables indiquent le surcoût induit par les neurones cachés. Par exemple, pour un jeu de deux tâches, 180 neurones sont nécessaires à la modélisation de ce problème, mais seuls 80 neurones codent la solution. Les autres neurones sont donc des neurones cachés ce qui représente 2.25 fois le nombre de neurones dits utiles.

Le nombre de ré-initialisations du réseau permettant d'obtenir une solution valide s'avère être très important pour une utilisation en ligne de ce type de réseau. En effet, les nombreuses ré-initialisations du réseau engendrent des évaluations de neurones supplémentaires qui sont directement liées au temps de convergence du réseau.

Les résultats d'un réseau pourvu de neurones inhibiteurs sont présentés par la table 3.4. Nous pouvons constater que l'utilisation des neurones inhibiteurs permet de réduire ce coût à un coût fixe égal à 5% du nombre de neurones total. Les neurones inhibiteurs permettent une réduction importante du nombre de neurones. Cette réduction par rapport au modèle classique est explicitée par la quatrième ligne de la table 3.4. Ainsi, sur les jeux de tâches considérés, les neurones inhibiteurs ont permis de réduire d'un facteur allant de 1.6 à 2.1 le nombre de neurones.

Les trois dernières lignes de la table 3.4 illustrent des résultats quant au nombre

Nombre de tâches	2	3	4	5	6	7
Nombre de neurones (N_n)	180	240	306	364	416	468
Nombre de neurones utiles (N_u)	80	120	160	200	240	280
Coût des neurones cachés	2.25	2	1.91	1.82	1.73	1.67
Nombre moyen de ré-initialisations du réseau	54	338	1492	9191	31783	28546
Nombre moyen d'évaluations de neurones	545	3379	14920	91910	317830	285460

TABLE 3.3 – Résultats d'un réseau modélisant l'ordonnancement de jeux de tâches possédant de 2 à 7 tâches sur une architecture disposant de deux unités d'exécution différentes, en considérant 20 cycles d'ordonnancements.

Nombre de tâches	2	3	4	5	6	7
Nombre de neurones	84	126	168	210	252	294
Nombre de neurones utiles	80	120	160	200	240	280
Coût des neurones cachés	1.05 (+5 %)					
Ratio de réduction	2.1	1.9	1.8	1.7	1.65	1.6
Nombre maximal de ré-initialisation du réseau	0	0	0	1	2	5
Nombre moyen d'évaluation de neurones	339	539	1025	1170	1583	1951
Gain du nombre d'évaluations	1.6	6.3	14.5	78	200	146

TABLE 3.4 – Résultats d'un réseau pourvu de neurones inhibiteurs sur un problème identique à celui décrit par la table 3.3.

d'évaluations nécessaire à la convergence du réseau. Sur ce point, les améliorations obtenues grâce aux neurones inhibiteurs sont très importantes par rapport à la solution classique. Par exemple, pour le jeu de six tâches, le nombre d'évaluations de neurones est réduit d'un facteur égal à 200. Cette amélioration est également due à la réduction du nombre de ré-initialisations nécessaires. En effet, les neurones inhibiteurs simplifiant les règles appliquées au réseau, il y a moins de minimums locaux et donc moins de risque de converger vers une solution non valide.

3.3.4 Résultats dans le contexte des SoC reconfigurables

Les architectures de type SoC reconfigurables peuvent posséder une dizaine de ressources différentes. Dans un souci de simplification de la présentation, nous considérons ici un SoC composé de cinq ressources différentes et une application composée de dix tâches. Nous supposons que ce SoC possède les ressources suivantes :

– R_1 est un GPP (*General Purpose Processor*),

– R_2, R_3, R_4 sont des blocs matériels IP (*Intellectual Property*) et
– R_5 est un accélerateur reconfigurable DRA (*Dynamic Reconfigurable Accelerator*).

A cause de la nature très hétérogène des ressources, certaines tâches ne sont exécutables que sur un sous-ensemble de ressources. Ainsi, nous supposons que sept tâches sont exécutables par le GPP (τ_2, τ_3, τ_4, τ_6, τ_8, τ_9, τ_{10}), trois tâches ne sont définies que pour les IP (τ_1, τ_5, τ_7) et six tâches sont également décrites pour le DRA (τ_2, τ_3, τ_6, τ_8, τ_9, τ_{10}). Les unités d'exécution étant de natures différentes, le temps d'exécution d'une tâche n'est pas le même sur toutes ces unités. Les WCET de chaque tâche pour chaque ressource sont récapitulés par la matrice $\mathbf{C}$ définie par l'équation (3.17). Lorsque la tâche n'est pas définie pour une architecture, son WCET est fixé à ∞.

$$\mathbf{C} = \{C_{i,j}\} = \left|\begin{array}{cccccccccc|c} \infty & 2 & 2 & 4 & \infty & 4 & \infty & 4 & 4 & 2 & R_1 \\ \infty & \infty & \infty & \infty & \infty & \infty & 10 & \infty & \infty & \infty & R_2 \\ \infty & \infty & \infty & \infty & 5 & \infty & \infty & \infty & \infty & \infty & R_3 \\ 4 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & R_4 \\ \infty & 2 & 1 & \infty & \infty & 2 & \infty & 2 & 1 & 2 & R_5 \\ \tau_1 & \tau_2 & \tau_3 & \tau_4 & \tau_5 & \tau_6 & \tau_7 & \tau_8 & \tau_9 & \tau_{10} & \end{array}\right| \tag{3.17}$$

La figure 3.14 présente des résultats de l'ordonnancement de l'application décrite précédemment sur le SoC considéré dans cette section. Toutes les tâches de l'application n'étant pas forcément exécutées simultanément, le comportement du réseau est évalué pour un nombre de tâches variant de 1 à 10. De plus, nous considérons une période d'ordonnancement égale à 10 *slots*. La modélisation du problème requiert 10 neurones par tâche et par ressource. Le réseau est donc composé de 500 neurones. Cependant, certaines tâches ne sont exécutables que sur certaines ressources. Les neurones représentant des *slots* d'exécution d'une tâche sur une ressource sur laquelle elle ne peut s'exécuter sont inutiles. Par exemple, la tâche τ_1 ne peut être exécutée que sur la ressource R_1. Il n'est donc pas nécessaire de placer des neurones représentant les *slots* de la tâche τ_1 sur les autres ressources. Finalement, d'après la matrice $\mathbf{C}$ (équation (3.17)), il existe 16 possibilités d'exécution différentes. Le réseau peut donc être réduit à 160 neurones.

La figure 3.14 montre que le nombre d'évaluations de chaque neurone est relativement stable et compris entre 8 et 12 sur l'exemple choisi dans cette section. Cette constance permet d'obtenir une évolution linéaire du nombre total d'évaluations de neurones par rapport au nombre de tâches. En effet, le nombre de neurones étant fonction du nombre de tâches, le nombre total d'évaluations des neurones est lié au nombre de tâches. Cette évolution linéaire est illustrée par la figure 3.14. Le nombre d'évaluations nécessaires à la convergence du réseau demeure limité grâce aux diminutions du nombre de neurones et du nombre de ré-initialisations.

3.3.5 Complexité

Dans les sections précédentes, les résultats présentaient le nombre d'évaluations de neurones. Si l'on considère l'évaluation d'un neurone comme opération élémentaire, le

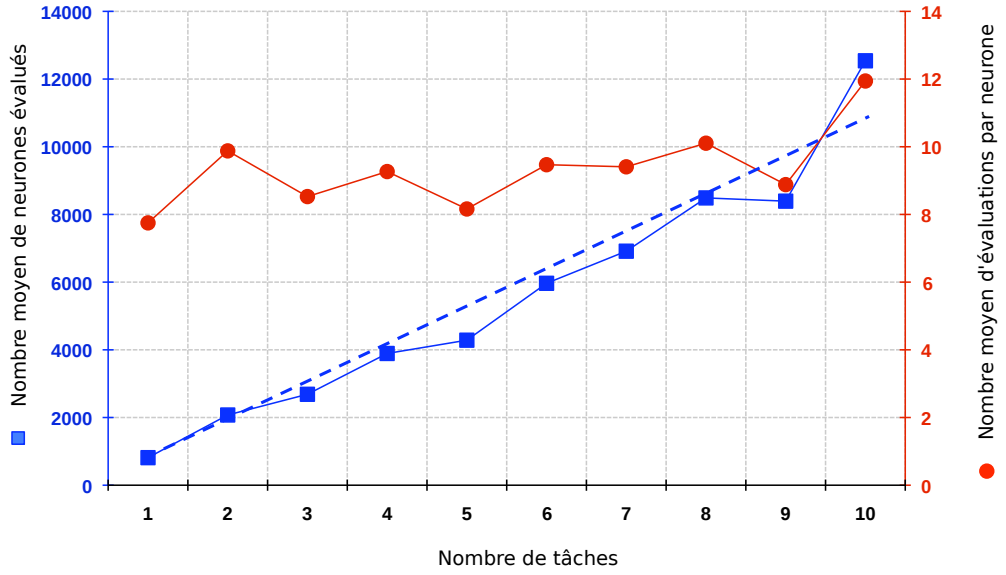


FIGURE 3.14 – Résultats de l'ordonnancement d'une application composée de 10 tâches sur un SoC composé de cinq unités d'exécution.

nombre d'évaluations de neurones fournit une indication quant à la complexité de l'évolution du réseau. Par exemple, la figure 3.14 a montré que la complexité de l'évaluation du réseau était linéairement fonction du nombre de tâches soumis à l'ordonnanceur.

Nous pouvons cependant être plus précis en exprimant la complexité d'évaluation d'un neurone. L'évaluation d'un neurone est donnée par l'équation (2.7). Cette équation est composée d'une multiplication-accumulation (les connexions) ainsi que d'une multiplication et d'une somme (la valeur de seuil). Le nombre d'éléments à multplier-accumuler pour calculer l'état d'un neurone x_i correspond au nombre de connexions vers ce neurone x_i.

Considérons un problème d'ordonnancement sur une période de S *slots*, consistant à ordonnancer T tâches, sur une architecture comportant R ressources. Nous notons $x_{t,r,s}$ le neurone associé au *slot* s de la tâche t sur la ressource r. Nous avons vu dans la section 3.2 que trois types de connexions sont appliqués au réseau. Nous les récapitulons et nous précisons le nombre de chacune d'entre elles.

- Un neurone $x_{t,r,s}$ est connecté aux neurones $x_{t,r,i}, \forall i \neq s$. Ces connexions permettent de contrôler le nombre de *slots* requis. Il y a donc $S-1$ connexions de ce type.
- Un neurone $x_{t,r,s}$ est connecté aux neurones $x_{i,r,k}, \forall i \neq t$. Ces connexions permettent d'éviter que plusieurs tâches soient exécutées simultanément sur la res-

source r. Il y a donc $T-1$ connexions de ce type.
- De plus, $R-1$ connexions proviennent des neurones inhibiteurs associés aux autres ressources.

Finalement, le nombre de connexions Nb_{conn} par neurone est égal à

$$Nb_{conn} = (S-1) + (T-1) + (R-1) = S + T + R - 3. \tag{3.18}$$

L'opération de multiplication-accumulation de n éléments est composée de n multiplications et $n-1$ additions. Ainsi, d'après l'équation (2.7), l'évaluation d'un neurone requiert

$$Nb_{mult} = Nb_{conn} + 1 = S + T + R - 2 \tag{3.19}$$
$$Nb_{add} = Nb_{conn} - 1 + 1 = S + T + R - 3, \tag{3.20}$$

où Nb_{mult} est le nombre de multiplications et Nb_{add} le nombre d'additions.

Le nombre d'opérations nécessaires à l'évaluation d'un neurone croît donc linéairement avec la période d'ordonnancement (S), le nombre de tâches (T) et de ressources (R).

3.4 Conclusion

Dans ce chapitre, nous avons présenté un ordonnanceur temporel basé sur des réseaux de neurones de Hopfield, supportant une architecture hétérogène. Afin de réduire le temps de convergence du réseau, il a été nécessaire d'introduire des neurones inhibiteurs. L'ajout de ces neurones inhibiteurs permet également d'améliorer la qualité des solutions générées par le réseau. Notre ordonnanceur a été comparé à l'algorithme d'ordonnancement pour architecture multiprocesseur Pfair. Nous avons montré que les résultats obtenus par notre réseau de neurones s'avèrent relativement proche de ceux obtenus par l'algorithme Pfair, qui est optimal. Ensuite, nous avons comparé notre réseau pourvu de neurones inhibiteurs à une solution neuronale plus classique. Nous avons ainsi pu constater que la réduction du nombre d'évaluations - engendrée notamment par l'utilisation de neurones inhibiteurs - est très importante. Finalement, nous avons proposé une utilisation de notre réseau dans un contexte réaliste, à savoir, l'ordonnancement temporel de tâches sur une architecture hétérogène de type SoC.

Chapitre 4

Placement de tâches pour architectures reconfigurables hétérogènes

Ce chapitre présente des algorithmes d'ordonnancement spatial pour architectures hétérogènes partiellement reconfigurables. Lors de la présentation des travaux relatifs au placement de tâches pour ce type d'architectures, nous avons constaté qu'une grande partie d'entre eux considèrent une surface homogène en termes de ressources. Dans ce chapitre, nous présentons des ordonnanceurs destinés à une surface hétérogène.

La première partie de ce chapitre présente un ordonnanceur spatial basé sur des réseaux de neurones de Hopfield [1]. Cet ordonnanceur permet de gérer l'hétérogénéité de l'architecture sans nécessiter de mécanismes complexes tels que de la relocation de *bitstream*. La deuxième partie, quant à elle, est fondée sur des hypothèses avant-gardistes, mais envisageables dans un futur proche. Ces hypothèses permettent de développer un modèle de tâches augmentant la flexibilité du système.

4.1 Placement d'un ensemble de tâches par un réseau de Hopfield

Cette section présente un réseau de neurones de Hopfield permettant de placer un ensemble de tâches sur une architecture reconfigurable hétérogène. Dans un premier temps, nous exposons comment l'architecture est modélisée afin de prendre en considération son caractère hétérogène. Puis, nous présentons comment les tâches sont modélisées afin d'exploiter l'hétérogénéité de la zone. Les sections suivantes présentent en détail la construction du réseau de neurones. De plus, afin d'améliorer la qualité des solutions générées, nous proposons une modification de la dynamique du réseau. Finalement, afin d'évaluer la qualité des résultats obtenus par notre réseau de neurones, il est comparé à l'algorithme SUP Fit [44].

Le réseau de neurones présenté dans cette section est construit par rapport à un

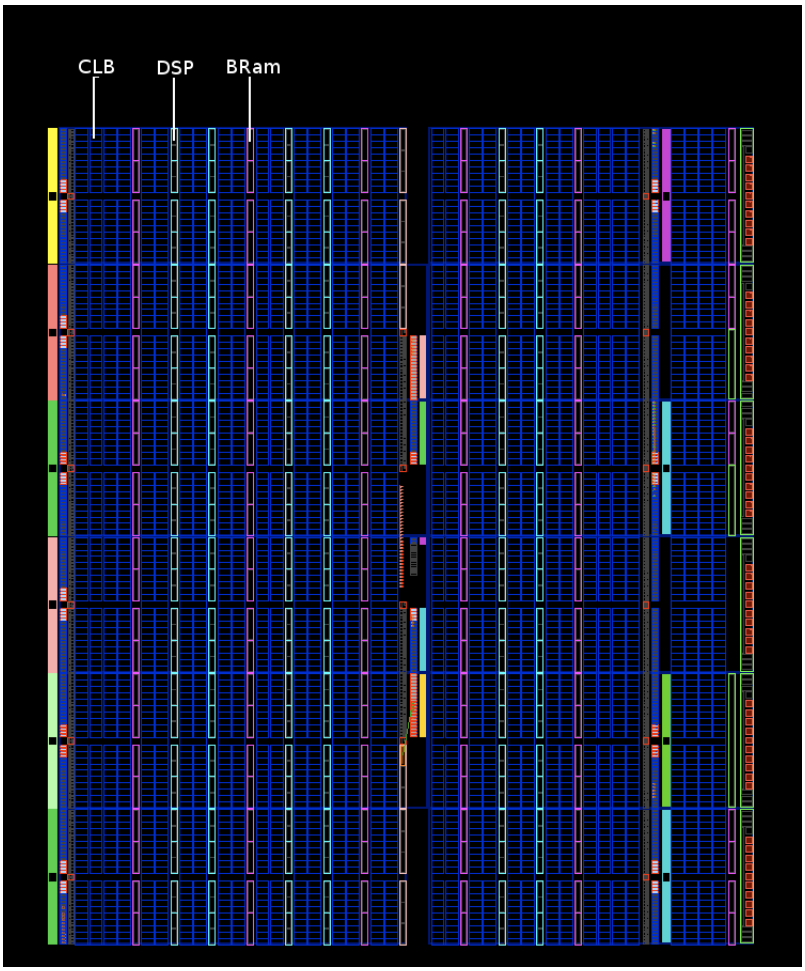


FIGURE 4.1 – Floorplan d'un Virtex 5

ensemble de tâches spécifiques. Cela signifie que de nouvelles tâches ne peuvent pas être ajoutées au réseau sans en modifier sa topologie. Nous considérons donc des applications composées de plusieurs tâches, s'exécutant dans un ordre non connu avant l'exécution.

4.1.1 Modélisation d'une architecture reconfigurable hétérogène

La figure 4.1 présente le *floorplan* d'un FPGA Xilinx Virtex 5. Ce FPGA est constitué principalement de trois types de ressources, à savoir, des CLB (régions bleues), des DSP (régions vertes) ainsi que des BRam (régions rouges). Il est important de constater l'hétérogénéité de la zone reconfigurable. Lors de la synthèse d'une application, un *bitstream* est généré pour une zone particulière du FPGA. Le choix de cette zone est décidé par le concepteur. Cependant, la position d'un *bitstream* sur la zone reconfigurable est fixée lors de la synthèse.

Il convient donc de définir un modèle permettant d'exprimer l'hétérogénéité d'une surface reconfigurable. La figure 4.2 présente la modélisation d'une partie du *floorplan* illustré par la figure 4.1. Les différents types de ressources sont modélisés en suivant le même code couleur que celui utilisé pour le *floorplan*. De plus, les ressources sont regroupées afin de créer des *régions*. Sur cet exemple, il y a 25 régions, chacune étant composée de quatre ressources. Trois types différents de régions apparaissent. Le type 1 contient uniquement des CLB, le type 2, deux CLB et deux DSP, et le type 3, trois CLB et un BRam. Le choix de la taille et du type des régions est déterminé par le concepteur.

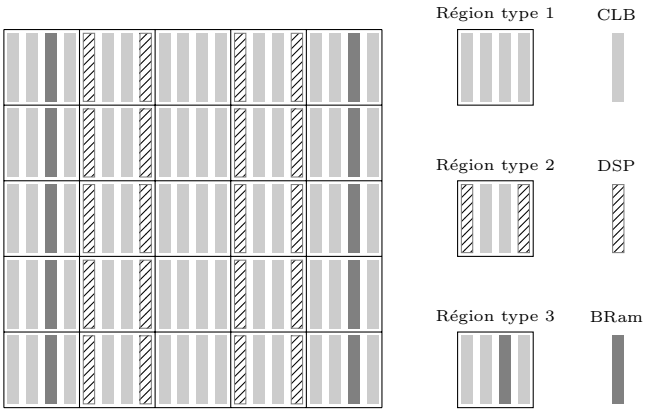


FIGURE 4.2 – Modélisation d'une architecture reconfigurable. La surface modélisée possède trois types de ressources. Ces ressources sont regroupées dans trois types de régions. La surface est composée de 25 régions.

Hypothèses matérielles. Au démarrage du système, la zone reconfigurable est découpée en régions et une modification de cette configuration requiert un redémarrage de l'architecture. Chaque région est reconfigurable indépendamment des autres. La taille des régions permet donc d'adapter le grain de reconfiguration.

Chaque tâche est matériellement exprimée par un *bitstream*. Une tâche peut être placée sur plusieurs régions. Il est donc nécessaire de disposer d'un réseau de communication entre les régions. Nous considérons disposer d'un tel mécanisme.

Les régions permettant d'adapter le grain de reconfiguration, dans la suite, nous modéliserons systématiquement la zone reconfigurable par une grille constituée de régions.

4.1.2 Modélisation d'une application

Dans nos travaux, nous considérons une application composée d'un ensemble de tâches. L'exécution d'une tâche nécessite un certain nombre et type de ressources déterminés lors de la synthèse. Les outils de synthèse disponibles actuellement génèrent

un *bitstream* ne pouvant être placé qu'à un endroit particulier de la zone reconfigurable. Afin d'exploiter la flexibilité offerte par les architectures partiellement reconfigurables, nous proposons de générer plusieurs implémentations d'une même tâche. Ces implémentations sont appelées *instances de tâches*. Dans la suite, $\tau_{i,j}$ désigne l'instance j de la tâche τ_i. La figure 4.3 présente une architecture reconfigurable disposant de deux types de régions au nombre de 25, ainsi que trois instances de la tâche τ_1. Les instances $\tau_{1,2}$ et $\tau_{1,3}$ utilisent le même type de régions, à savoir, six régions de type 1. L'instance $\tau_{1,1}$, quant à elle, utilise quatre régions de type 1 et deux régions de type 2.

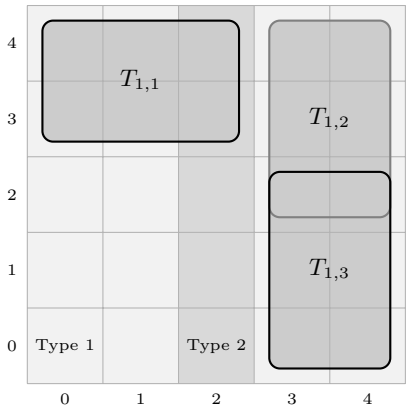


FIGURE 4.3 – Instances de la tâche 1

La figure 4.4 présente les instances de deux tâches τ_1 et τ_2. Les ressources utilisées par ces instances sont récapitulées dans le tableau 4.1. Cet exemple permet d'illustrer l'utilité des instances en termes de flexibilité. Supposons que l'instance $\tau_{2,1}$ soit placée et que la tâche τ_1 doive être exécutée. Alors, l'instance $\tau_{1,1}$ ne peut pas être placée car elle utilise des régions occupées par l'instance $\tau_{2,1}$, qui est en cours d'exécution. En revanche, l'instance $\tau_{1,2}$ ou $\tau_{1,3}$ peut être placée. Les différentes instances d'une même tâche permet donc au système de s'adapter dynamiquement au flot d'exécution.

Tâches	Instances	♯ Type 1	♯ Type 2
τ_1	$\tau_{1,1}$	4	2
τ_1	$\tau_{1,2}$	6	0
τ_1	$\tau_{1,3}$	6	0
τ_2	$\tau_{2,1}$	5	0
τ_2	$\tau_{2,2}$	3	3

TABLE 4.1 – Ressources utilisées par le jeu de tâches de la figure 4.4.

4.1.3 Modélisation du problème

Nous considérons un ensemble de tâches, chacune étant décrite par au moins une instance. Le problème du placement de tâches, tel que nous le considérons dans cette

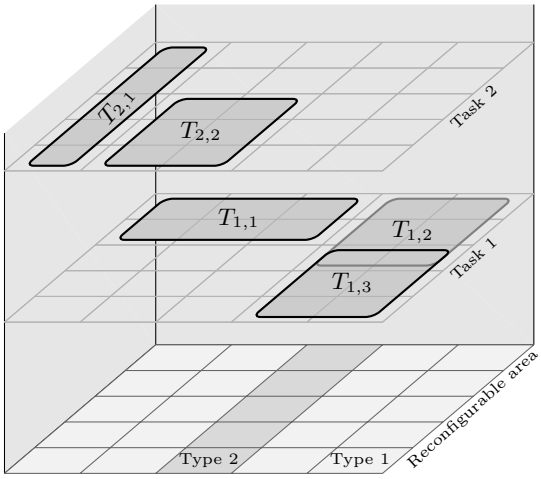


FIGURE 4.4 – Exemple de ressources utilisées et de position des instances de deux tâches.

section, consiste à déterminer un emplacement pour une instance de chaque tâche. L'algorithme que nous proposons permet donc de placer un ensemble de tâches simultanément. Cette section décrit la construction du réseau de neurones associé au problème de placement de tâches.

La première étape consiste à définir une représentation neuronale du problème. Nous associons un neurone à une instance de tâche. Si nous considérons un ensemble de tâches τ et $l(t)$ le nombre d'instances de la tâche $t \in \tau$, le nombre de neurones est $\sum_{i \in \tau} l(i)$. Quand le réseau a convergé, un neurone activé signifie que la tâche lui étant associée peut être placée et exécutée.

La figure 4.5 présente le réseau (représenté ici sans les connexions) permettant de modéliser l'exemple présenté par la figure 4.4. La figure 4.5 servira de schéma de base permettant d'illustrer les différentes étapes de la construction du réseau. Il possède donc cinq neurones représentant les cinq instances de tâches. Trois neurones représentent les trois instances de la tâche τ_1 et deux neurones représentent les deux instances de la tâche τ_2. L'instance j de la tâche i étant notée $\tau_{i,j}$, le neurone associé à cette instance sera noté $x_{i,j}$.

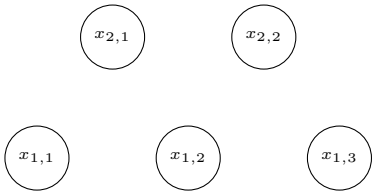


FIGURE 4.5 – Réseau de neurone associé aux tâches décrites par la figure 4.4

Une fois le modèle neuronal défini, il convient de définir les poids des connexions et les valeurs des seuils des neurones. Selon notre modèle, la résolution du problème

de placement consiste à satisfaire deux contraintes. La première permet de contrôler le nombre d'instances de tâches devant être activées, la seconde permet d'éviter le chevauchement de tâches. De plus, une fonction de coût peut être ajoutée au réseau afin d'accroître le nombre d'instances placées. Ces trois fonctions sont formulées indépendamment comme trois fonctions d'énergie, qui sont finalement sommées en vertu du caractère additif des réseaux de Hopfield. Les sections suivantes présentent en détail la construction de ces trois fonctions puis leur sommation. En guise d'exemple, l'ensemble de tâches décrit par la figure 4.4 est utilisé.

4.1.3.1 Contrôle du nombre d'instances activées

Chaque tâche est décrite par une ou plusieurs instances. Il convient donc de contrôler le nombre d'instances activées pour chaque tâche. Tagliarini et al. [63] ont défini la règle k-de-n que nous avons largement présentée dans la section 2.5.1. Nous rappelons qu'elle permet de s'assurer qu'un réseau contenant n neurones converge vers un état où exactement k neurones sont activés. Les paramètres associés à cette règle sont

$$w_{i,j} = \begin{cases} 0 & \text{si } i == j \\ -2 & \text{sinon} \end{cases} \qquad t_i = 2k - 1. \tag{4.1}$$

Cette règle est appliquée sur toutes les instances de chaque tâche, elle est donc appliquée sur un sous ensemble des neurones du réseau. Ainsi, pour toute tâche t, n est égal à $l(t)$. Généralement, k est instancié à 1 afin de n'activer qu'une instance de chaque tâche. Une règle 1-de-$l(t)$ est donc appliquée sur tous les neurones représentant les instances de la tâche t.

Le nombre d'instances à placer peut être choisi aisément en paramétrant k. Il est ainsi possible d'activer zéro, une ou plusieurs instances d'une tâche. Dans la suite, le problème de placement est restreint au placement d'exactement une instance pour chaque tâche afin d'améliorer la lisibilité. Cette restriction n'engendre aucune perte de généralité.

La figure 4.6 illustre l'application des règles 1-de-n sur le réseau associé à l'ensemble de tâches décrit par la figure 4.4. Une règle 1-de-3 est appliquée sur les neurones associés à la tâche τ_1, et une règle 1-de-2 sur les neurones associés à la tâche τ_2.

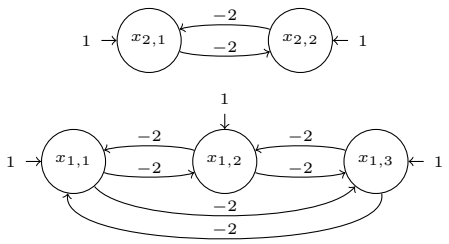


FIGURE 4.6 – Application des règles 1-de-n au réseau associé aux tâches décrites par la figure 4.4

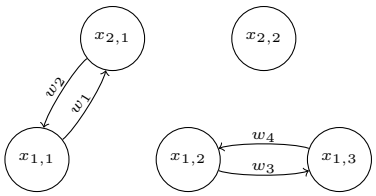


FIGURE 4.7 – Illustration des connexions permettant de contrôler le chevauchement des instances des tâches décrites par la figure 4.4.

4.1.3.2 Éviter le chevauchement d'instances de tâches

Deux instances sont dites *chevauchantes* si ces deux instances utilisent des régions identiques. Une solution présentant un chevauchement d'instances est une solution incorrecte car inutilisable. Le réseau de neurones doit donc converger vers une solution dépourvue de tout chevauchement. Une relation entre deux instances se chevauchant est créée afin d'éviter l'activation de ces deux instances simultanément.

En termes de neurones, la figure 4.7 présente les connexions entre des neurones associés à des instances se chevauchant. Des connexions sont créées entre les neurones $x_{1,1}$, $x_{2,1}$ et $x_{1,2}$, $x_{1,3}$ car, d'après la figure 4.4, les instances leur étant associées utilisent des régions communes. Dans la suite des explications, nous nous focaliserons sur les poids des connexions entre les neurones $x_{1,1}$ et $x_{2,1}$. w_1 est le poids de la connexion du neurone $x_{1,1}$ vers le neurone $x_{2,1}$ et w_2 est le poids de la connexion du neurone $x_{2,1}$ vers le neurone $x_{1,1}$.

Afin d'éviter le chevauchement, l'activation du neurone $x_{1,1}$ doit inhiber l'activation du neurone $x_{2,1}$ et réciproquement. Par l'équation (2.2), l'équation d'évaluation du neurone $x_{1,1}$ est :

$$x_{1,1} = H(x_{2,1} \times w_2 + t_{1,1}). \tag{4.2}$$

Si $x_{2,1}$ est activé ($x_{2,1} = 1$), il est nécessaire que $w_2 < -|t_{1,1}|$ afin d'inhiber $x_{1,1}$. Symétriquement, w_1 doit être inférieur à $-|t_{2,1}|$. Si l'on considère un réseau plus grand, cette condition reste suffisante car les connexions provenant d'autres neurones sont négatives, elles ne peuvent donc pas amener d'énergie.

Une condition nécessaire pour assurer la convergence du réseau de Hopfield est la symétrie de la matrice de connexion $\mathbf{W}$. Les poids w_1 et w_2 doivent donc être égaux, cependant cela n'est pas vrai selon la précédente définition. Pour satisfaire cette condition, la fonction *min* est utilisée. Les poids w_1 et w_2 sont donc définis par

$$w_1 = w_2 = min(-|t_{1,1}|, -|t_{2,1}|) - 1. \tag{4.3}$$

Ainsi, les connexions empêchant le chevauchement des instances sont symétriques. De plus, les poids de ces connexions sont toujours négatifs. Donc, toutes les contraintes nécessaires à la convergence du réseau sont satisfaites. Cette règle est appliquée entre toutes les paires d'instances présentant un chevauchement.

La construction des poids w_1 et w_2 est fonction des valeurs de seuil des neurones. Cette règle doit donc être appliquée quand le seuil des neurones est fixé, c'est à dire,

après que l'application de toutes les autres rêgles participant à la construction du réseau aient été appliquées.

Notons qu'une règle 1-de-n pourrait également être utilisée afin de contrôler l'exclusion mutuelle de deux tâches se chevauchant. Cependant, lorsque d'autres règles sont appliquées sur ce type de neurones, une règle 1-de-n ne permet que de tendre vers une exclusion mutuelle de ces neurones. Or nous souhaitons expressément éviter l'activation simultanée de neurones associés à des tâches se chevauchant car cela conduit à une solution non valide.

4.1.3.3 Augmenter le nombre d'instances placées

Afin d'augmenter le nombre d'instances placées, une fonction de coût est ajoutée au réseau de neurones. Dans [44], les auteurs ont proposé un algorithme de placement pour architecture hétérogène en considérant également différentes instances d'une tâche. Ils ont proposé une heuristique permettant de définir un ordre de prédilection quant au choix d'une instance.

L'idée principale de cette heuristique est qu'une instance utilisant des régions peu utilisées est préférable à une instance utilisant des régions fortement utilisées par d'autres instances. En effet, si une instance est placée à un emplacement fortement demandé par d'autres instances, elle va empêcher le placement d'autres instances nécessitant cet emplacement.

La zone reconfigurable est divisée en régions, et une instance de tâche utilise une ou plusieurs régions. Pour chaque région, le *degré d'occupation*[1] est déterminé en fonction du nombre d'instances occupant cette région. Le degré d'occupation d'une région est d'autant plus important que cette région est utilisée par un grand nombre d'instances de tâche. À partir de ce degré d'occupation, un *poids de position*[2] est attribué à chaque instance. Ce poids dépend du degré d'occupation de toutes les régions utilisées par l'instance. Plus ces régions sont utilisées, plus le poids de position sera important. Les instances sont alors classées par poids de position, du plus faible au plus important. Ce classement permet de choisir une instance nécessitant des ressources peu utilisées.

En guise d'exemple, les poids de position des instances $\tau_{2,1}$ et $\tau_{2,2}$ sont respectivement 0.65 et 0.5. Intuitivement, la sélection de $\tau_{2,2}$ est un meilleur choix que $\tau_{2,1}$, car $\tau_{2,1}$ et $\tau_{1,1}$ se chevauchent. Si $\tau_{2,1}$ est placée, il n'est alors plus possible de placer $\tau_{1,1}$. Soit $L(\tau_i)$ la liste ordonnée des instances de la tâche τ_i, d'après les poids de position de la tâche τ_2, nous avons

$$L(\tau_2) = [\tau_{2,2}, \tau_{2,1}]. \tag{4.4}$$

Soit $Pos(\tau_{i,j})$ la position de l'instance $\tau_{i,j}$ dans la liste ordonnée $L(\tau_i)$ dont les indices appartiennent à $\mathbb{N}^*$. En considérant $L(\tau_2)$, nous aurions donc $Pos(\tau_{2,1}) = 2$ et $Pos(\tau_{2,2}) = 1$. Pour un certain nombre de neurones activés, la valeur minimale de la fonction de coût,

$$F_c = -\sum_i \sum_j x_{i,j} \times \frac{1}{Pos(\tau_{i,j})}, \tag{4.5}$$

1. nommé *static utilization probability* dans l'article originel [44]
2. nommé *position weight* dans l'article originel [44]

a tendance à minimiser le nombre de tâches rejetées, car les neurones utilisant des régions peu utilisées sont activés.

Des valeurs de seuil peuvent être dérivées de la fonction F_c. Ainsi, la valeur de seuil du neurone $x_{i,j}$ est

$$t_{i,j} = \frac{1}{Pos(\tau_{i,j})}. \tag{4.6}$$

Par exemple, les valeurs des seuils des neurones $x_{2,1}$ et $x_{2,2}$ sont respectivement 0.5 et 1.

La fonction de coût F_c permet donc de créer des valeurs de seuil permettant de réduire le nombre de tâches rejetées.

4.1.3.4 Fusion des différentes fonctions

Les sections précédentes ont présenté deux contraintes et une fonction de coût. Pour construire le réseau, les poids des connexions et les valeurs des seuils résultants de ces trois fonctions doivent être additionnés. L'énergie du réseau est la somme des énergies des fonctions le constituant. Soit,

- E_o la fonction d'énergie associée à la contrainte de chevauchement,
- E_i la fonction d'énergie associée à la contrainte de respect du nombre d'instances,
- E_c la fonction d'énergie associée à la fonction de coût,

la fonction d'énergie globale du réseau est alors

$$E = \alpha E_o + \beta E_i + E_c$$

où α et β sont deux coefficients réels.

Lorsque l'énergie E du réseau n'est pas minimale, l'énergie apportée par au moins une fonction d'énergie n'est pas minimale et la solution produite n'est pas optimale. Si l'énergie apportée par les fonctions exprimant les contraintes E_o et E_i n'est pas minimale, la solution est non seulement sous optimale mais surtout non valide. Si seule la fonction de coût E_c n'est pas minimale, la solution est sous optimale mais valide.

On peut cependant considérer une solution ne respectant pas le nombre requis d'instances comme étant une solution valide. En effet, le nombre d'instances n'est pas celui souhaité mais les instances peuvent être placées. Ce cas se produit notamment lorsqu'il n'est pas possible de placer toutes les instances demandées, par manque de surface ou de ressources. La solution contiendra un sous ensemble des instances souhaitées.

L'addition des différentes règles est généralement une opération assez complexe car la valeur des coefficients (α et β) appliqués aux fonctions d'énergie doit être déterminée. Ces coefficients permettent d'accorder plus ou moins d'importance à chaque fonction. Il est ainsi possible de privilégier les contraintes au détriment de la fonction de coût afin d'obtenir des solutions valides. Comme nous l'avons vu dans la section 2.6, la détermination de ces coefficients peut s'avérer être une tâche très complexe.

Lors de la construction des fonctions d'énergie, des précautions ont été prises afin de privilégier implicitement certaines fonctions. Comme nous l'avons vu dans la section 4.1.3.2, la fonction E_o est appliquée après les autres fonctions car elle dépend des

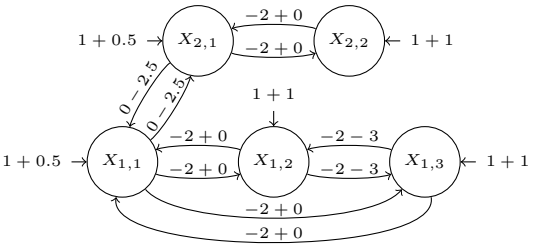


FIGURE 4.8 – Réseau de neurones de Hopfield associé aux tâches décrites par la figure 4.4.

valeurs de seuils apportées par les fonctions E_i et E_c. La contrainte incarnée par la fonction E_o est donc, par construction, toujours respectée.

Les valeurs des seuils apportées par la fonction de coût E_c sont quant à elles strictement inférieures à un. Cette propriété lui permet de ne pas compromettre les règles k-de-n. Si un neurone est désactivé par une règle k-de-n, son énergie est au maximum égale à

$$-2k + 2k - 1 = -1$$

car au moins k neurones sont dans un état actif. Or une énergie strictement inférieure à zéro aurait été suffisante pour désactiver le neurone. Cette marge peut donc être utilisée par la fonction de coût F_c en garantissant qu'elle ne compromettra pas les règles k-de-n.

Finalement, grâce à la construction des fonctions d'energie, les coefficients du réseau sont trivialement déterminés et valent

$$\alpha = \beta = 1.$$

La figure 4.8 présente la construction du réseau de neurones de Hopfield associé à la figure 4.4. Le premier terme de la somme des valeurs de seuil est la valeur provenant des règles k-de-n, le second, provient de la fonction de coût. Le premier terme de la somme des poids de connexions provient des règles k-de-n, le second, de la contrainte de chevauchement.

4.1.4 Modification de la dynamique du réseau

Afin d'exploiter la fonction de coût présentée dans la section 4.1.3.3, il est nécessaire de modifier la dynamique du réseau. En effet, l'application seule de la fonction de coût n'est pas suffisante pour améliorer la qualité des résultats car elle ne s'exprime qu'à travers des valeurs de seuil.

L'objectif de notre démarche est de privilégier, lors des premières itérations, l'activation de certains neurones, en l'occurrence les neurones associées aux instances occupant des zones peu utilisées par d'autres instances. Nous avons vu que la fonction de coût permet de définir des valeurs de seuil en fonction du taux d'utilisation des régions : plus le taux d'utilisation des zones occupées par une instance est important, plus la valeur

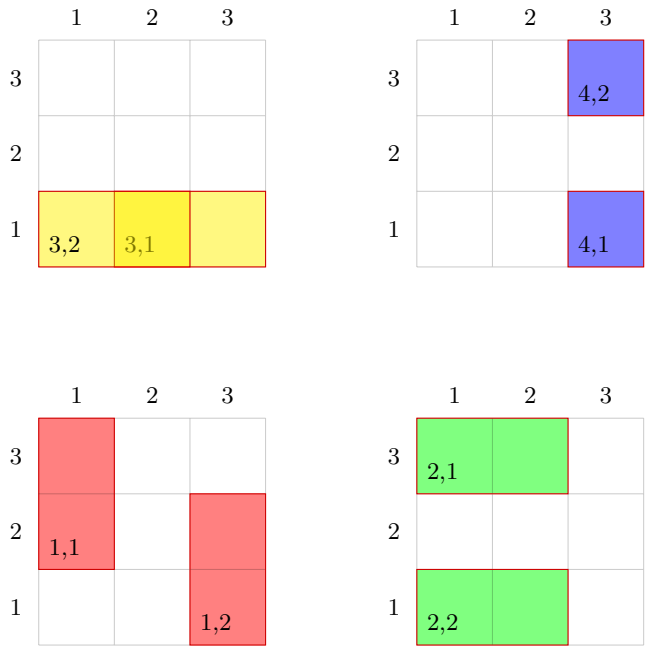


FIGURE 4.9 – Exemple d'un jeu de tâches

de seuil est petite. Les neurones à privilégier possèdent donc une valeur de seuil plus importante. Pour tirer parti de ces valeurs de seuil, nous allons modifier le comportement de la fonction de seuillage H définie par l'équation (2.1).

Afin d'expliquer notre démarche, nous allons considérer l'ensemble de tâches représentées par la figure 4.9. Les caractéristiques de ces tâches sont résumées dans le tableau 4.2. Ce tableau présente également la valeur de seuil des neurones associés à chaque instance. Cette valeur est la somme de la valeur de seuil apportée par les règles k-de-n ainsi que par la fonction de coût. Notons que, comme précédemment, des règles 1-de-n sont appliquées entre les différentes instances d'une même tâche. Les valeurs de seuil appartiennent donc à l'intervalle $]1, 2]$.

Les instances ayant une valeur de seuil égale à deux sont les instances associées aux neurones occupant les régions les moins utilisées de la surface reconfigurable. Ce sont donc ces neurones qui doivent être privilégiés.

À l'initialisation du réseau, tous les neurones sont à l'état inactif. Ainsi, aucun neurone ne reçoit d'énergie via ses connexions. En utilisant la fonction d'activation H classique (qui active un neurone si son énergie est supérieure ou égale à zéro), le premier neurone évalué de cet exemple sera activé, qu'il soit ou non intéressant. Or, si la fonction d'activation compare l'énergie du réseau, non plus à zéro, mais à la valeur deux, alors seul les neurones les plus intéressants seront activés lors de cette itération. Il convient ensuite d'utiliser la fonction d'activation H afin d'effectuer une exécution normale du réseau. Cette technique permet donc de guider l'initialisation afin d'améliorer la qualité des solutions générées par le réseau.

Instance	x	y	Largeur	Hauteur	Valeur de seuil
$\tau_{1,1}$	1	2	1	2	2.0
$\tau_{1,2}$	3	1	1	2	1.5
$\tau_{2,1}$	1	3	2	1	2.0
$\tau_{2,2}$	1	1	2	1	1.5
$\tau_{3,1}$	2	1	2	1	2.0
$\tau_{3,2}$	1	1	2	1	1.5
$\tau_{4,1}$	3	1	1	1	2.0
$\tau_{4,2}$	3	3	1	1	1.5

TABLE 4.2 – Récapitulatif des caractéristiques des tâches représentées par la figure 4.9

Cette modification de la dynamique du réseau peut être pratiquée sur plusieurs itérations du réseau. Afin de généraliser cette technique, la fonction d'activation H définie par l'équation (2.1) est modifiée. Nous définissons donc la fonction d'activation H' telle que

$$\forall x(t) \in \mathbb{R}, H'(x(t)) = \begin{cases} 0 & \text{si} \quad x(t) < s(t) \\ 1 & \text{si} \quad x(t) \geq s(t) \end{cases}, \tag{4.7}$$

où $s(t)$ est une fonction décroissante telle que

$$\exists \alpha \in \mathbb{N} : \forall t > \alpha \Rightarrow s(t) = 0.$$

La fonction s, illustrée par la figure 4.10, est donc une fonction décroissante qui est égale à zéro après un certain nombre d'itérations. Concrètement, lors des premières itérations, l'activation d'un neurone requiert une énergie strictement supérieure à zéro. Après un certain nombre d'itérations, l'énergie des neurones est comparée à zéro et la fonction H' se comporte comme la fonction H. Il est important de remarquer qu'après un certain nombre d'itérations, les comportements de la fonction H et H' sont identiques. Cela permet d'affirmer que la convergence du réseau est également garantie lorsque la fonction H' est utilisée.

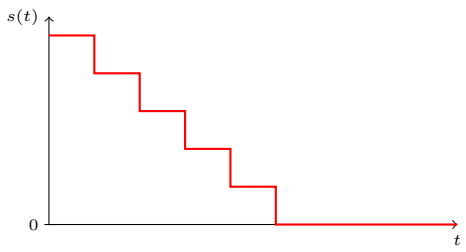


FIGURE 4.10 – Exemple d'une fonction s. La fonction s est une fonction décroissante par échelon puis constante en zéro.

Si l'on considère l'ensemble de tâches décrit par la figure 4.9, les valeurs de seuil des

$s(t)$	$H = [0]$	$H' = [2\,;\,0]$
Pourcentage de jeux placés entièrement	37.58	50.75

TABLE 4.3 – Comparaison des fonctions d'activation H et H'.

neurones sont égales à 1.5 ou 2. Ainsi, la fonction

$$s(t) = \begin{cases} 2 & \text{si } t = 0 \\ 0 & \text{sinon} \end{cases}, \tag{4.8}$$

peut être utilisée afin de guider l'initialisation des neurones. Lors de la première itération, la fonction H' compare l'énergie des neurones à 2, puis, lors des itérations suivantes à 0. Ainsi, dans un premier temps, seuls les neurones possédant une valeur de seuil égale à 2 pourront être activés. À cause des contraintes entre neurones, il est possible que tout les neurones ayant une valeur de seuil égale à 2 ne puissent être activés. Alors, les itérations suivantes activeront des neurones ayant une valeur inférieure à 2.

La table 4.3 présente des résultats de placement illustrant l'amélioration apportée par la modification de la fonction d'activation. Un réseau de neurones a été construit en considérent le jeu de tâches décrit par la table 4.2. Nous comparons la fonction d'activation standard H définie par l'équation (2.1) à la fonction d'activation H' utilisant la fonction s définie par l'équation (4.8). Pour chacune de ces fonctions 1000 simulations ont été effectuées. Le pourcentage moyen de jeux entièrement placés a été retenu. Nous pouvons ainsi constater que la fonction H' apporte une amélioration d'environ 13% sur un jeu où il n'y a que peu de possibilités de placement.

Finalement, nous pouvons résumer le fonctionnement d'un réseau disposant d'un seuillage dynamique par les trois étapes suivantes

1. définir les échelons d'une fonction de seuillage s,
2. initialiser les neurones à l'état inactif,
3. évaluer tous les neurones tant que $s(t) \neq 0$,
4. quand $s(t) = 0$, laisser converger le réseau.

4.1.4.1 Choix d'une fonction s

L'exemple de la figure 4.9 utilisé précédement pour illustrer notre méthode est un exemple très simple. Il n'y a en effet que deux instances par tâche. La simplicité de cet exemple a permis de facilement déterminer les échelons de la fonction s. Lorsque les jeux de tâches sont plus importants, il peut s'avérer plus complexe de déterminer des échelons adéquats. La détermination des échelons va donc être effectuée de manière expérimentale.

La fonction de seuillage est décroissante dans l'intervalle $[1, 2]$ puis constante en zéro. Les valeurs proches de deux permettent de sélectionner les neurones privilégiés par la fonction de coût. Il semblerait donc avantageux de systématiquement définir

une fonction s dont les premières valeurs de seuil seraient proches de deux. Or, ce type de fonction peut mener à des minimums locaux réduisant ainsi la qualité du placement.

La figure 4.11 illustre ce phénomène. Différentes fonctions s sont testées sur un jeu de sept tâches possédant chacune six instances. Afin d'évaluer la qualité des résultats produits par l'utilisation de chacune de ces fonctions, la figure 4.11 présente le pourcentage moyen de simulations ayant menées au placement de toutes les tâches soumises.

Comme un réseau de Hopfield est non déterministe, 1000 simulations pour chaque fonction s ont été effectuées et la moyenne a été retenue. La figure 4.11 présente donc le pourcentage de placement complet par rapport l'utilisation de différentes fonctions de seuils. En abscisse, les valeurs de seuil utilisées sont précisées. Lorsque la fonction H est utilisée ([0.0]), le pourcentage de jeux entièrement placés est de 50.5%.

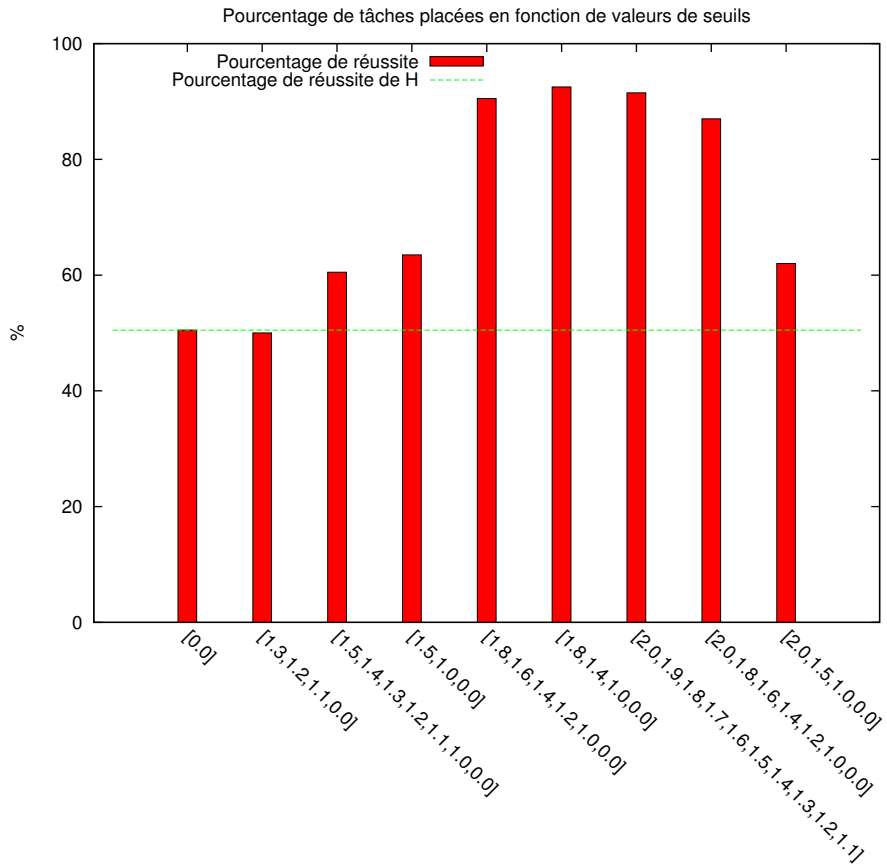


FIGURE 4.11 – Simulation de différentes fonction s. L'abscisse décrit les échelons utilisés tandis que l'ordonnée présente le pourcentage de tâches placées en fonction des échelons utilisés.

Nous pouvons observer que les meilleurs résultats ne sont pas nécessairement obtenu en utilisant les valeurs de seuil les plus élevées (les plus proches de deux). En effet, ce type de valeurs de seuil restreint la diversité des premières instances pouvant être placées, ce qui conduit à une augmentation du nombre de minimums locaux. Par exemple, lorsque la valeur de seuil est fixée à deux, seules les instances ayant une valeur de seuil supérieure ou égale à deux peuvent être placées. Si la valeur de seuil est fixée à 1.8, il existe potentiellement plus d'instances pouvant être placées.

De plus, il est important de noter que les valeurs de seuil dynamiques ne dégradent jamais la qualité des résultats par rapport à la dynamique classique (illustrée par la valeur de seuil [0.0]).

4.1.4.2 Seuillage dynamique et nombre d'instances activées.

Dans les sections précédentes, afin de simplifier les explications, des règles 1-de-n ont été appliquées aux neurones associés aux instances d'une même tâche. Ainsi, il n'est pas possible d'exécuter (et donc de placer) deux instances d'une même tâche simultanément.

Or, dans un cadre plus général, notre réseau permet également d'activer plusieurs instances d'une même tâche. Les règles 1-de-n sont alors remplacées par des règles k-de-n, k étant égal aux nombres d'instances souhaitées. Dans ce cas, les valeurs de seuil des neurones n'appartiennent plus à l'intervalle $]1, 2]$ comme énoncé dans la section 4.1.4. Pour appliquer notre technique de seuillage dynamique, il est alors nécessaire de normaliser les valeurs de seuil et les connexions de tels neurones de manière à ramener à 1 les valeurs de seuil des neurones. Cette normalisation peut être réalisée en divisant par $2k - 1$, les valeurs de seuil ainsi que les poids de connexions.

4.1.4.3 Implémentation du seuillage dynamique.

Pour réaliser le seuillage dynamique, la fonction H a été modifiée. Une alternative aurait été de modifier dynamiquement les valeurs de seuil des neurones du réseau. Ces deux solutions sont équivalentes en terme de comportement cependant, leur implémentation diffère. Dans nos travaux, nous avons choisi d'implémenter le seuillage dynamique en modifiant la fonction H.

4.1.5 Résultats

Cette section présente des expérimentations de notre algorithme de placement. Dans un premier temps, les résultats du placement de jeux de tâches par notre algorithme sont présentés. Ces résultats sont finalement comparés à ceux obtenus en utilisant l'algorithme *SUP Fit* [44].

Pour évaluer ces algorithmes, nous générons des jeux de tâches aléatoirement ayant certaines caractéristiques. Nous verrons que la taille des instances est définie aléatoirement dans un certain intervalle. Mais les paramètres importants des jeux sont le nombre de tâches et le nombre d'instances par tâche. De ces deux paramètres dépend directement la facilité ou non du placement d'un jeux de tâches. Nous pourrons donc analyser

le comportement de notre algorithme en fonction de la complexité de placement des jeux de tâches.

Conditions d'expérimentation. La zone reconfigurable considérée pour ces simulations est modélisée par une matrice de dimension 10×10. Parce que les instances ont des positions prédéfinies, il n'est pas nécessaire d'exprimer l'hétérogénéité de la zone. L'algorithme est testé sur des jeux de tâches générés aléatoirement. La hauteur et la largeur de chaque instance de tâche générée appartient à l'intervalle $[1, 5]$, ce qui correspond au maximum à la moitié de la taille de la surface modélisée. De plus, nous avons appliqué un seuillage dynamique au réseau. La fonction de seuillage utilisée possède les valeurs de seuils

$$[1,8\ ;\ 1,4\ ;\ 1\ ;\ 0].$$

Ces valeurs ont été choisies empiriquement en effectuant des simulations avec des valeurs de seuils différentes sur des jeux de tâches représentatifs.

Parce que nous sommes dans un contexte hétérogène, il paraît difficile d'estimer la qualité du placement en évaluant le degré de fragmentation de la zone. Nous basons donc l'estimation de la qualité des résultats sur le pourcentage de tâches placées.

4.1.5.1 Simulations du placement d'un ensemble de tâches

La table 4.4 présente les résultats du placement d'un jeu de neuf tâches possédant chacune cinq instances. Les réseaux de neurones de Hopfield ayant un comportement non déterministe, il est nécessaire d'effectuer plusieurs simulations pour estimer la qualité des résultats. En guise d'exemple, la table 4.4 contient les résultats de 10 simulations.

Le nombre de tâches placées n'est pas toujours optimal. Cependant, aucune solution obtenue ne présente d'instances se chevauchant. De plus, dans aucune solution, deux instances d'une même tâche ont été sélectionnées. Toutes les solutions sont donc utilisables. La moyenne du nombre de tâches placées est 8.5, le nombre optimal de tâches placées étant de 9, les résultats de cette simulation s'avèrent très proche de l'optimum.

Simulations	1	2	3	4	5	6	7	8	9	10
♯ Tâches placées	9	8	9	8	7	8	9	9	9	9

TABLE 4.4 – Nombre de tâches placées pour 10 simulations de placement d'un jeu de 9 tâches possédant chacune 5 instances.

4.1.5.2 Simulations sur plusieurs jeux de tâches ayant des caractéristiques identiques

La figure 4.12 est un histogramme présentant les résultats du placement de 10 jeux de tâches. Dans cet exemple, chaque jeu de tâches contient neuf tâches possédant chacune cinq instances. Chacune de ces instances est générée aléatoirement, elles possèdent donc des positions et formes différentes à chaque simulation.

Pour chaque jeu de tâches, 20 simulations sont effectuées afin d'évaluer le comportement moyen du réseau de neurones. L'histogramme présente donc les moyennes de 20 simulations par jeu de tâches. Le nombre moyen de tâches placées varie entre six et neuf. La moyenne est de 7.9 pour l'ensemble des simulations. De plus, lors de toutes les simulations, aucun chevauchement n'a été produit et le nombre d'instances par tâche est inférieur ou égal à un. Les solutions produites sont donc toutes valides.

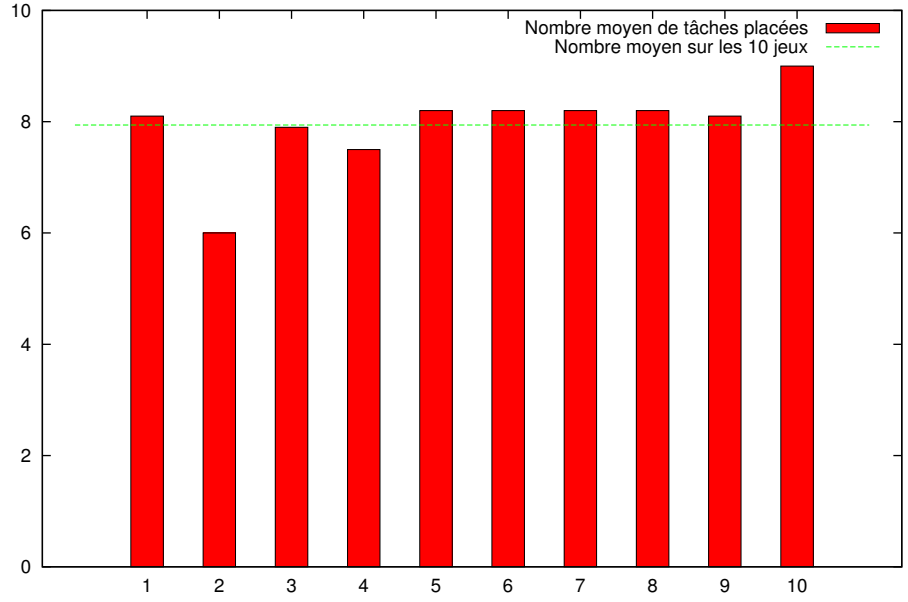


FIGURE 4.12 – Moyennes du nombre de tâches placées. Chaque jeu contient neuf tâches, et chaque tâche possède cinq instances.

4.1.5.3 Comparaisons avec l'algorithme *SUP Fit*

La figure 4.13 présente une comparaison entre l'algorithme *SUP Fit* [44] et notre algorithme basé sur des réseaux de neurones. Cet histogramme compare le *taux de succès* de ces deux algorithmes. Nous considérons ici qu'un *succès* est une solution où toutes les tâches sont placées. Le taux de succès est donc le pourcentage de placement complet.

Afin de comparer ces deux algorithmes, ils ont été testés sur plusieurs jeux de tâches aux propriétés différentes. Ainsi, un jeu de tâches particulier contient un certain nombre de tâches et chaque tâche possède un certain nombre d'instances. Par exemple, le jeu de tâches 9×6 de la figure 4.13 contient neuf tâches possédant chacune six instances. Dans cette évaluation, toutes les tâches d'un jeu de tâches possèdent le même nombre d'instances afin de simplifier le cadre d'étude. Bien entendu, chaque tâche pourrait posséder un nombre distinct d'instances. 40 jeux de tâches sont générés aléatoirement

pour chaque type. La procédure d'évaluation pour chaque jeu de tâches est la même que celle décrite pour la figure 4.12. Finalement, le taux de succès moyen de chaque ensemble de 40 jeux de tâche est retenu.

L'algorithme *SUP Fit* a été conçu pour placer une tâche sur une architecture hétérogène. Pour placer un ensemble de n tâches, il est donc nécessaire d'exécuter n fois l'algorithme. L'ordre de placement des différentes tâches impacte les résultats du placement de l'ensemble de tâches. Afin d'en capturer le comportement moyen, tous les ordres de placement de tâches sont soumis à l'algorithme *SUP Fit*, le taux de succès moyen est alors retenu.

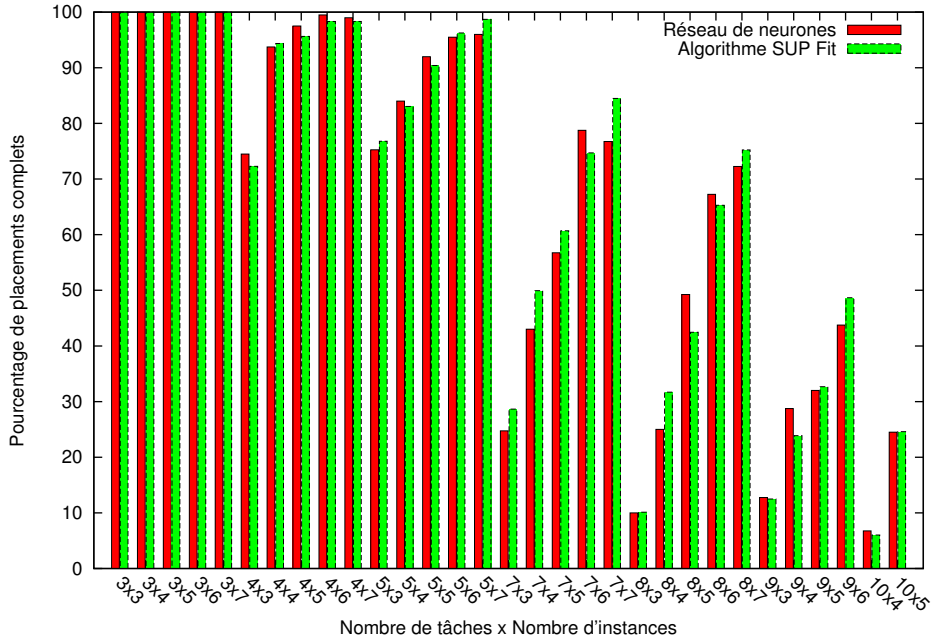


FIGURE 4.13 – Comparaison des taux de succès de l'algorithme *SUP Fit* et de notre algorithme.

La figure 4.13 montre que notre réseau de neurones fournit des résultats similaires à ceux obtenus par l'algorithme *SUP Fit*. En effet, le taux de succès moyen obtenu par l'algorithme *SUP Fit* sur l'ensemble des jeux de tâches est égal à 66.96% et pour notre réseau de neurones, il est égal à 66.43%. Nous disposons donc d'un ordonnanceur spatial fournissant des solutions de qualité équivalente à l'algorithme *SUP Fit* tout en bénéficiant des qualités intrinsèques des réseaux de neurones de Hopfield. Ainsi, notre ordonnanceur profite d'une implémentation matérielle efficace commune aux réseaux de Hopfield. De plus, des techniques développées dans le chapitre 5, telles que la tolérance aux fautes et la parallélisation de l'évaluation du réseau, sont applicables à notre ordonnanceur. Finalement, un dernier point très intéressant est que le caractère non déterministe des réseaux de neurones permet de générer des solutions différentes à chaque appel à l'ordonnanceur. Il nous est ainsi possible de générer plusieurs solutions

afin de choisir la plus satisfaisante.

4.1.5.4 Comparaison en temps d'exécution

Dans cette section, nous avons présenté des résultats relatifs à la qualité des solutions obtenues. Afin de mener une étude complète de notre algorithme, il serait nécessaire d'effectuer une comparaison en temps d'exécution par rapport à d'autres solutions. Cependant, il s'avère complexe de comparer une solution neuronale à une autre solution, de part les différences existantes entre les modèles d'exécution. Pour effectuer une comparaison en complexité, il serait nécessaire de borner le nombre d'itérations assurant la convergence. Or cette étude mathématique requiert des outils très sophistiqués issus du domaine des systèmes dynamiques. En ce qui concerne une comparaison en temps d'exécution, nous ne disposons pas actuellement d'une implémentation matérielle de notre réseau de neurones. Parce qu'une implémentation logicielle du réseau de neurones n'est pas représentative de ses performances, nous ne pouvons donc pas évaluer correctement son temps d'exécution. Afin de fournir un ordre de grandeur des performances de notre réseau, la figure 4.14 présente le nombre d'évaluations de neurones en fonction du nombre de tâches et d'instances. Chaque résultat présenté est la moyenne de 20 jeux de tâches aléatoirement évalués 20 fois. Ainsi, pour un ensemble de neuf tâches, chacune disposant de cinq instances, le réseau a convergé suite à l'évaluation de $176,7$ neurones.

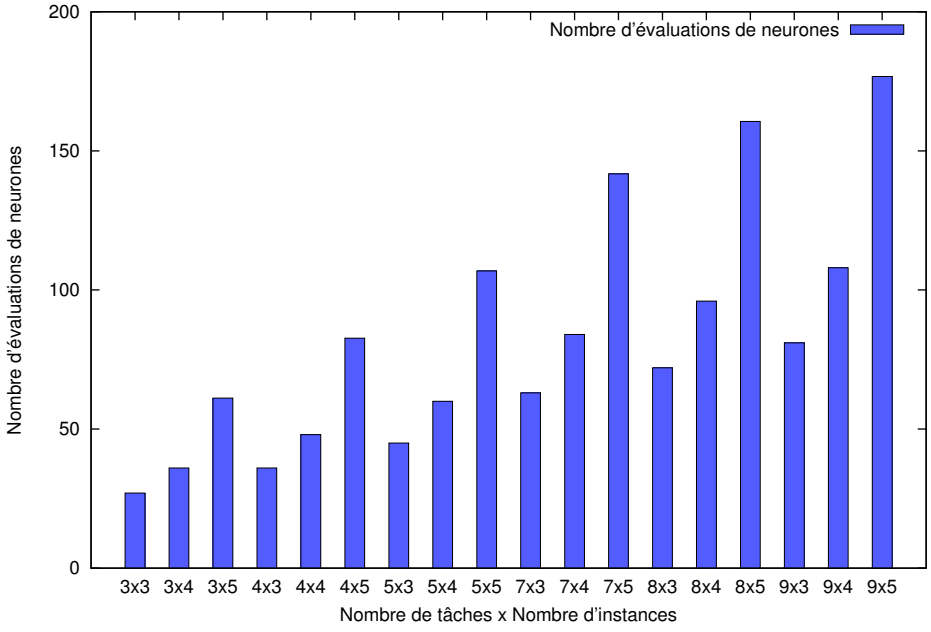


FIGURE 4.14 – Nombre d'évaluations de neurones en fonction du nombre de tâches et d'instances.

4.1.6 Utilisation du réseau de neurones dans un contexte réel

Jusqu'à présent, tant pour la présentation du réseau que pour son expérimentation, nous avons considéré le placement d'un ensemble de tâches sur une zone reconfigurable complètement libre. Dans cette section, nous présentons les mécanismes permettant de s'abstraire de ces hypothèses, à savoir, le placement d'un sous ensemble de tâches et la modélisation de l'état de la zone reconfigurable.

4.1.6.1 Placement d'un sous ensemble de tâches

Nous avons vu, qu'à la construction du réseau de neurones, toutes les instances des tâches sont prises en compte. Or, lors de l'exécution de l'application, il est fortement probable qu'à un instant donné, seul un sous ensemble des tâches doit être placé. Dans ce cas, l'état des neurones associés aux instances de tâches ne devant pas être placées peut être bloqué à l'état inactif, peu importe le résultat de leurs évaluations. Cette méthode présente l'avantage d'être très simple à implémenter. Cependant, des évaluations inutiles seront effectuées. Pour éviter ces évaluations, il est nécessaire d'ajouter un mécanisme visant à contrôler quels sont les neurones à évaluer.

Le mécanisme bloquant l'état des neurones dépasse le modèle d'évaluation originel d'un réseau de neurones de Hopfield. Un moyen théoriquement plus élégant de contrôler les tâches à placer consisterait à profiter des règles k-de-n appliquées sur les instances de chaque tâche. Supposons qu'à un instant t, toutes les tâches de l'application doivent être placée à l'exception de la tâche τ_i. Si à l'instant t, la règle 1-de-n appliquée sur les instances de la tâche τ_i est modifiée en une règle 0-de-n, alors aucune instance de la tâche τ_i ne sera placée à l'instant t. Cette méthode impliquant des modifications des paramètres du réseau (valeurs de seuil et de connexion), nous préférerons bloquer artificiellement l'état des neurones, dans un souci de simplifier l'implémentation du réseau.

4.1.6.2 Gestion de l'état de la zone reconfigurable

Lors de l'exécution d'une application, les tâches sont placées à différents instants, en fonction de son flot d'exécution. L'ordonnanceur doit donc placer des tâches sur une zone partiellement occupée. Il convient donc d'être en mesure de spécifier ces occupations à l'ordonnanceur. En fait, les zones occupées correspondent à des tâches en cours d'exécution, plus précisément, à des instances de tâches en cours d'exécution. Si l'état des neurones associés à ces instances est bloqué à l'état actif, la contrainte de chevauchement du réseau de neurone empêchera l'activation d'autres instances utilisant ces zones. Nous disposons donc d'un moyen simple de spécifier les zones occupées : lorsqu'une instance est placée, l'état du neurone lui étant associé est bloqué à l'état inactif jusqu'à que l'instance ait terminé son exécution.

Les deux mécanismes précédemment évoqués utilisent la même technique, à savoir, la capacité de bloquer artificiellement l'état d'un neurone. Ainsi, une implémentation

du réseau de neurone pourvue de cette fonctionnalité permet d'exploiter l'ordonnanceur dans un contexte applicatif réel.

4.1.7 Conclusion

Dans cette section, nous avons présenté un algorithme basé sur un réseau de Hopfield permettant de placer un ensemble de tâches sur une zone reconfigurable hétérogène. La méthode de construction du réseau a été présentée et s'avère relativement simple en vertu de l'absence de coefficients appliqués aux règles composant le réseau. Ces règles garantissent l'obtention d'une solution de placement valide. La détermination des échelons permettant d'améliorer la qualité des solutions requiert cependant quelques expérimentations. Enfin, nous avons comparé notre algorithme par rapport à un autre algorithme de placement de tâches sur une architecture hétérogène reconfigurable.

4.2 Placement en colonne

Dans la première partie de ce chapitre, le modèle d'instance que nous avons utilisé permettait d'apporter de la flexibilité tout en évitant les problèmes que peuvent poser la relocation de *bitstream*. Dans cette partie, nous supposons disposer d'un outil de relocation, plus précisément, d'un outil supportant un cas particulier de la relocation de *bitstream*. En effet, nous pouvons distinguer deux degrés différents dans la complexité de l'opération de relocation, à savoir

1. reloger un *bitstream* sur des zones offrant les mêmes ressources,
2. reloger un *bitstream* sur des zones offrant des ressources de types différents.

La synthèse d'une tâche sur une architecture reconfigurable peut utiliser des ressources particulières, non présentes sur l'ensemble de la zone. Ainsi, il semble très complexe de reloger un *bitstream* sur des zones offrant des ressources de types différents car il serait nécessaire d'effectuer dynamiquement une nouvelle synthèse afin d'obtenir un *bitstream* utilisant uniquement les ressources offertes par l'emplacement choisi. Cependant, la relocation d'un *bitstream* à un emplacement offrant les mêmes ressources que l'emplacement originel semble bien plus réaliste.

Dans cette partie du chapitre, nous présentons un modèle de tâche permettant de dissocier ces deux degrés de relocation. Ainsi, l'algorithme présenté suppose la relocation de *bitstream* uniquement entre des zones offrant des ressources identiques.

Motivations. Dans la première partie du chapitre, nous avons présenté un algorithme de placement pour architectures hétérogènes. Nous utilisions alors les *instances* de tâches pour supporter le caractère hétérogène de l'architecture. À chaque emplacement d'exécution d'une tâche, nous associons une instance. Lorsque la surface offerte par l'architecture reconfigurable augmente, pour maintenir une flexibilité équivalente, le nombre d'instances doit être d'autant augmenté. Ainsi, le concept d'instance tel que nous l'avions défini peut montrer ces limites.

La figure 4.15 présente l'évolution en terme de quantité de ressources disponibles pour différentes générations de FPGA Xilinx issus de la famille Virtex. La métrique choisie pour illustrer l'évolution de ces FPGA est le nombre de *slices*[3]. Comme les familles Virtex sont composées de plusieurs modèles aux nombres et types de ressources différents, le graphique indique le nombre de *slices* minimum et maximum de l'ensemble des modèles de chaque famille.

Nous constatons donc qu'en moins de quatre ans, le nombre de *slices* des plus gros modèles de FPGA est passé d'environ 40 000 *slices* à environ 300 000 *slices*. Cette augmentation très importante du nombre de *slices*, et donc des capacités de calcul des FPGA, permet d'exécuter des applications de plus en plus complexes sur ce type de cibles.

3. Les *slices* sont les unités programmables élémentaires des FPGAs Xilinx. À partir des Virtex 5, ils sont principalement composés de quatre LUT.

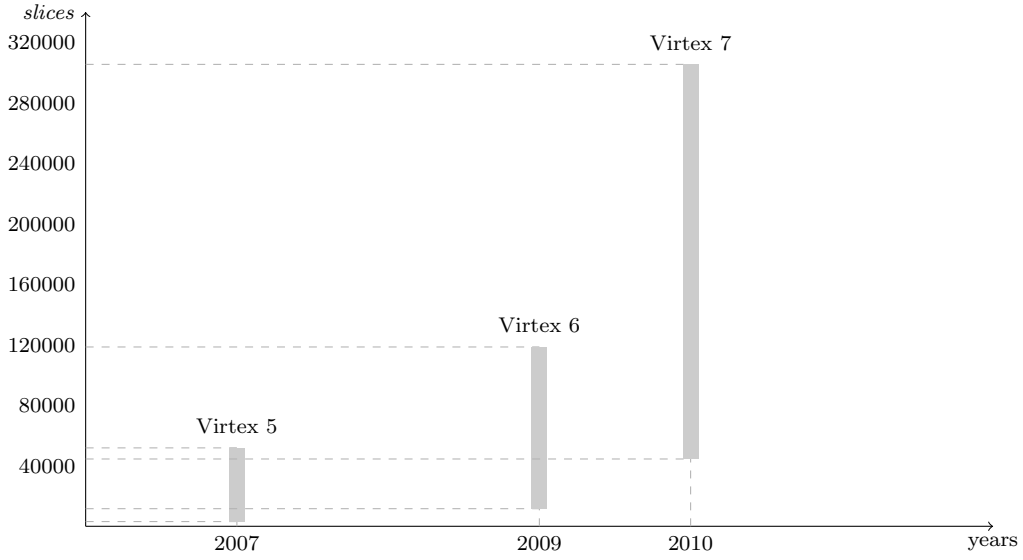


FIGURE 4.15 – Évolution du nombre de *slices* de la famille Virtex de Xilinx. Chaque barre indique le nombre minimal et maximal de *slices* de l'ensemble des modèles au sein d'une famille.

À cause de l'augmentation importante de la surface des FPGA, le nombre de positions pour chaque tâche doit être augmenté afin de s'adapter au mieux au flot d'exécution. C'est ainsi que le nombre d'instances augmente.

Nous tâchons donc de trouver une méthode permettant de réduire ce nombre d'instances tout en conservant une flexibilité identique de placement sur une architecture reconfigurable hétérogène.

La première section de cette partie présente une organisation typique des ressources d'une architecture reconfigurable. Les observations de cette organisation permettent de développer un nouveau modèle de tâches, présenté dans la deuxième section. La troisième section présente un algorithme simple ayant pour objectif d'illustrer les avantages de ce modèle de tâches. Finalement, la dernière section compare le modèle de tâches utilisé dans la première partie du chapitre avec ce nouveau modèle.

4.2.1 Organisation des ressources d'une architecture reconfigurable hétérogène

Les ressources d'une architecture reconfigurable sont relativement organisées. La figure 4.16 expose explicitement cette organisation. Nous pouvons observer que les zones en surbrillance sont composées des mêmes ressources, à savoir, des DSP (en vert), des CLB (en bleu) et des BRam (en rouge), en nombres et emplacements identiques. Bien que cette figure ne mette en évidence que les deux premières lignes, toute la zone reconfigurable est organisée de cette manière. La zone reconfigurable est donc composée

FIGURE 4.16 – Illustration de la présence de symétries dans la disposition des ressources d'une architecture reconfigurable hétérogène (Xilinx Virtex 5 SX50t).

de colonnes, et chaque colonne est constituée de régions identiques. Dans la suite, nous exploitons cette organisation pour définir un nouveau modèle de tâches.

4.2.2 Modélisation d'une tâche

La section précédente a montré que la surface reconfigurable peut être décomposée en colonnes et qu'une colonne offre des ressources identiques sur toute sa hauteur. Nous allons donc utiliser cette propriété pour définir un nouveau modèle d'instances de tâche.

Une instance peut utiliser les ressources de plusieurs colonnes consécutives. Elle a donc une colonne comme origine et sa largeur est définie par un nombre de colonnes. Comme une colonne offre des ressources identiques sur toute sa hauteur, nous considérons qu'une instance peut être placée verticalement n'importe où.

La figure 4.17 présente l'instance $\tau_{1,1}$ d'une tâche τ_1. La figure 4.17.a présente les caractéristiques de l'instance. Elle a pour origine la colonne 1 et requiert des ressources des colonnes 2 et 3, sa hauteur est égale à deux régions, une région étant composée d'un ensemble de ressources. Les figures 4.17.b, 4.17.c et 4.17.d présentent les trois autres positions possibles pour l'instance $\tau_{1,1}$. Ainsi, une instance peut être placée à quatre emplacements différents.

Les caractéristiques $c_{\tau_{1,1}}$ d'une instance $\tau_{1,1}$ sont définies par le triplet $(l_{\tau_{1,1}}, h_{\tau_{1,1}}, V_{\tau_{1,1}})$ où

- $l_{\tau_{1,1}}$ correspond au nombre de colonnes utilisées ;
- $h_{\tau_{1,1}}$ correspond au nombre de lignes ;
- $V_{\tau_{1,1}}$ est un vecteur de positions possibles défini par $V_{\tau_{1,1}} = [1, ..., h_{ard} - h_{\tau_{1,1}}]$, avec h_{ard} qui correspond à la hauteur de la surface.

De plus, dans la suite, nous noterons $\tau_{i,j,k}$ l'instance $\tau_{i,j}$ placée à la position $v^k_{\tau_{i,j}}$, où $v^k_{\tau_{i,j}}$ dénote l'élément k du vecteur de positions potentielles $V_{\tau_{i,j}}$. Et nous nommerons *instance positionnée* l'instance $\tau_{i,j,k}$.

Par exemple, les caractéristiques de l'instance $\tau_{1,1}$ décrite par la figure 4.17 sont définies par le triplet $(3, 2, [1, 2, 3, 4])$.

De plus, nous conservons la notion d'instance afin de pouvoir utiliser différents types de ressources pour exécuter une tâche. La figure 4.18 présente deux instances de la tâche τ_1. L'instance $\tau_{1,1}$ a été décrite précédemment. Quant à l'instance $\tau_{1,2}$, elle utilise

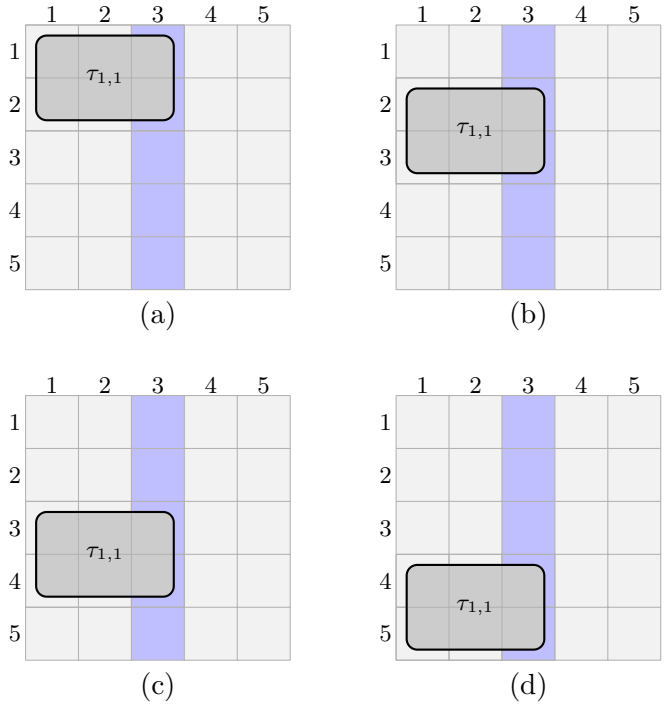


FIGURE 4.17 – Description de toutes les positions verticales de l'instance $\tau_{1,1}$.

des colonnes différentes et dispose de trois positions différentes. Ces caractéristiques sont définies par le triplet $c_{\tau_{1,2}} = (2, 3, [1, 2, 3])$. Une tâche est donc décrite par un vecteur de triplets, chacun de ces triplets décrivant une instance. Ainsi, la tâche τ_1 est décrite par le vecteur de caractéristiques de ses instance $C_{\tau_1} = [c_{\tau_{1,1}}, c_{\tau_{1,2}}]$.

Nous constatons donc qu'avec seulement deux instances, nous disposons désormais de sept emplacements différents pour l'exécution de la tâche τ_1. En effet, nous avons $|V_{\tau_{1,1}}| + |V_{\tau_{1,2}}| = 7$ où $|V|$ est le cardinal du vecteur V.

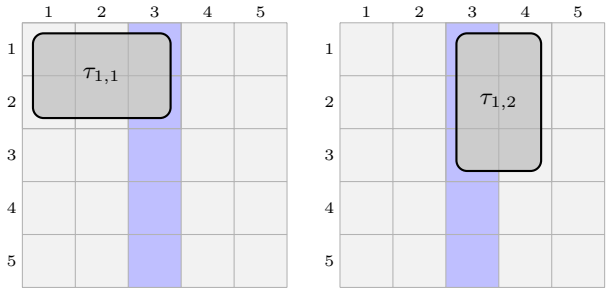


FIGURE 4.18 – Illustration de deux instances de la tâche τ_1.

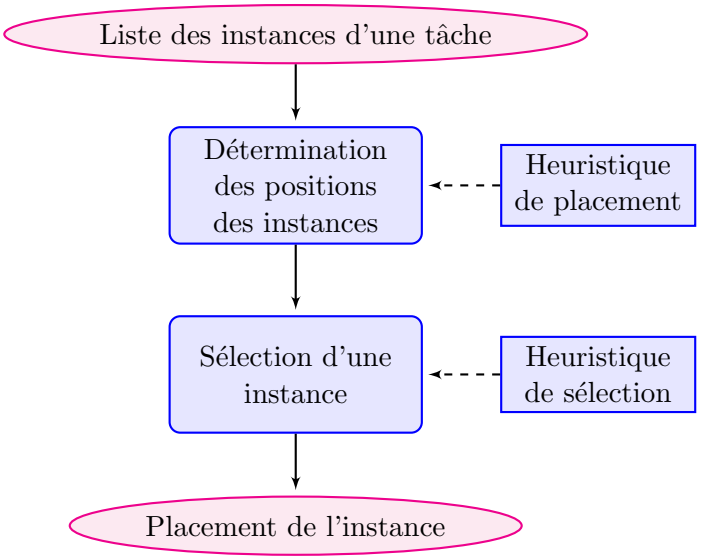


FIGURE 4.19 – Illustration des étapes de l'algorithme de placement de tâches.

4.2.3 Algorithme de placement

Dans cette section, nous présentons un algorithme simple exploitant notre modèle d'instance par colonne. Dans la suite, nous nommons cet algorithme Colonne. Nous détaillons l'opération de placement d'une tâche. Cette opération est basée sur une heuristique inspirée des travaux de Marconi et al.[49]. De plus, nous préciserons brièvement l'opération de suppression d'une tâche, qui est relativement triviale.

4.2.3.1 Placement d'une tâche

La figure 4.19 présente le fonctionnement global de l'algorithme dans le cas du placement d'une tâche ; il est principalement constitué de deux phases. La première étape consiste à déterminer des positions potentielles pour l'ensemble des instances associées à la tâche devant être placée. La seconde étape consiste à sélectionner une instance parmi les instances pouvant être placées. Ces deux étapes nécessitent des heuristiques pouvant être de nature différente en fonction des objectifs imposés. Par exemple, si l'on souhaite conserver un type de ressources en particulier, il convient de choisir une instance n'utilisant pas ce type de ressources. Dans la suite, nous avons choisi d'appliquer une heuristique plus généraliste ayant pour but de limiter la fragmentation de la zone reconfigurable.

Détermination de la position des instances. La première étape de l'algorithme consiste à déterminer des positions pour les instances de la tâche devant être placée. Ces choix sont effectués en utilisant une heuristique. Nous avons choisi de nous inspirer de l'algorithme *quad corner* [49].

Les auteurs de [49] s'intéressent au placement de tâches sur une zone reconfigurable homogène. Ils proposent cependant une heuristique intéressante pour déterminer l'emplacement d'une tâche. L'objectif de cette heuristique est de conserver un grand espace contigu au centre de la surface. Les tâches sont donc placées prioritairement aux bords de la zone reconfigurable. Ainsi, l'algorithme qu'ils proposent est nommé *quad corner* car les tâches sont prioritairement placées aux quatre coins de la zone. De plus, les tâches sont placées dans un coin en fonction de leur taille. Ainsi, chaque coin accueille des tâches ayant une taille semblable afin de limiter la fragmentation.

Parce que nous considérons une zone reconfigurable hétérogène, la liberté de placement est plus restreinte que dans les travaux de Marconi et al. où ils considèrent une zone homogène. Ainsi, nous nous contentons de choisir des positions le plus au bord de la zone sans considérer la taille des tâches. Afin d'estimer les positions des instances, nous déterminons leur centre de gravité g, $g_{\tau_{i,j,k}}$ dénotant le centre de gravité de l'instance $\tau_{i,j}$ placée verticalement à la position v_k. Nous notons g_{ard} le centre de gravité de la zone reconfigurable. La position verticale la plus au bord de la zone reconfigurable étant la position la plus eloignée du centre de gravité g_{ard}, nous sélectionnons la position la plus eloignée du centre de gravité g_{ard}. Bien entendu, seules les positions n'occupant pas de régions déjà utilisées peuvent être sélectionnées. Pour une instance $\tau_{i,j}$, décrite par le triplet $(h_{\tau_{i,j}}, l_{\tau_{i,j}}, V_{\tau_{i,j}})$, nous souhaitons donc sélectionner l'instance positionnée

$$\tau_{i,j,k} \mid \max(d(g_{\tau_{i,j,k}}, g_{ard})) \ \forall \ k, \tag{4.9}$$

qui n'utilise pas de régions occupées par d'autres instances.

La figure 4.20 illustre le fonctionnement de notre heuristique. Nous considérons que la tâche τ_1 décrite par la figure 4.18, doit être placée. il s'agit donc de sélectionner des instances positionnées pour les instances $\tau_{1,1}$ et $\tau_{1,2}$. La figure 4.20.a décrit l'état de la zone reconfigurable au moment où les positions des instances $\tau_{1,1}$ et $\tau_{1,2}$ doivent être déterminées. Les régions hachurées sont occupées par des tâches en cours d'exécution. Elles ne peuvent donc pas être utilisées. Les figures 4.20.b et 4.20.c décrivent respectivement les positions des instances $\tau_{1,1}$ et $\tau_{1,2}$ déterminées par notre heuristique.

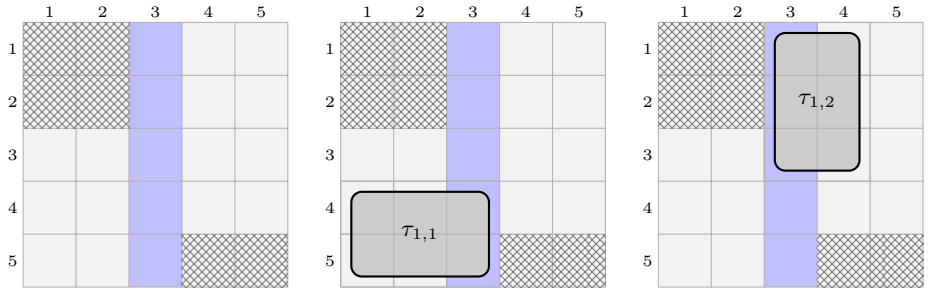


FIGURE 4.20 – Détermination des positions en utilisant l'heuristique issue de l'algorithme *quad corner*. (a) L'état de la zone reconfigurable. Les régions hachurées sont occupées par d'autres tâches. (b) Position obtenue pour l'instance $\tau_{1,1}$. (c) Position de l'instance $\tau_{1,2}$

Sélection d'une instance. Lorsque les instances d'une tâche devant être placée se sont vues attribuer des positions, il convient d'en choisir une. À nouveau, différentes heuristiques peuvent être utilisées pour effectuer ce choix. Dans notre proposition, nous visons à préserver un espace libre au centre de la zone reconfigurable. Ainsi, nous choisissons l'instance la plus éloignée du centre de la zone. Nous considérons le placement de la tâche τ_i. L'étape précédente de l'algorithme a sélectionné une instance positionnée pour chaque instance d'une tâche τ_i, ce qui constitue le vecteur d'instances positionnées $P = [\tau_{i,1,k_1}, \tau_{i,2,k_2}, ..., \tau_{i,j,k_j}]$. Cette étape de l'algorithme consiste à sélectionner l'instance positionnée

$$\tau_{i,l,k_l} \mid \max(d(g_{\tau_{i,l,k_l}}, g_{ard})) \; \forall \; l. \tag{4.10}$$

En considérant l'exemple décrit par la figure 4.20, l'instance positionnée $\tau_{1,1,4}$ serait sélectionnée.

Remarque. Le comportement de l'algorithme *SUP Fit* que nous avions présenté précédemment peut être équivalent à celui de l'algorithme *quad corner*. Si l'on considère une surface homogène où une tâche peut être placée n'importe où, c'est à dire qu'il existe une instance pour toute position, alors, l'algorithme *SUP Fit* placera les tâches dans les coins. En effet, les régions au bord de la surface sont les régions utilisées par le moins d'instances et donc les régions privilégiées lors des premiers placements.

4.2.3.2 Suppression d'une tâche

Lorsqu'une tâche a terminé son exécution, il est nécessaire de libérer l'espace qu'elle occupait sur la ressource. Cette étape est dépendante de l'implémentation de l'algorithme de placement. Lors du placement d'une tâche, il est nécessaire de disposer d'une structure de données capturant l'état de la zone reconfigurable. L'étape de suppression d'une tâche consiste simplement à mettre à jour cette structure de données. Par

exemple, si la zone reconfigurable est modélisée par une matrice dont chaque élément représente l'état d'une région par une variable binaire, la suppression d'une tâche consiste à modifier l'état des éléments de la matrice décrivant les régions utilisées par cette tâche.

Compactage des tâches. Notons qu'il serait possible de compacter la zone reconfigurable suite à la suppression d'une tâche. Cependant, cette opération nécessite de déplacer des tâches en cours d'exécution, ce qui requiert une sauvegarde du contexte des tâches. Or, il s'avère particulièrement difficile d'obtenir le contexte d'une tâche matérielle [56]. Notre algorithme n'effectue donc pas de compactage.

4.2.4 Expérimentations

Cette section présente des simulations de notre algorithme de placement en colonne. Pour l'évaluer, des scénarios d'exécution sont aléatoirement générés. L'algorithme précédemment exposé est alors appliqué à chaque *slot* du scénario. Le pourcentage de tâches placées par notre algorithme est alors comparé à celui obtenu par l'algorithme *SUP Fit*. Finalement, une comparaison du nombre d'instances nécessaires à l'obtention de résultats similaires entre *SUP Fit* et notre algorithme illustrera les avantages de ce dernier.

4.2.4.1 Scénario d'exécution

La figure 4.21 présente un exemple de scénario d'exécution d'une application composée de six tâches. Dans ces expérimentations, nous considérons que chaque tâche de l'application peut être exécutée plusieurs fois et que chaque exécution d'une même tâche peut nécessiter un nombre différent de *slots*. Par exemple, la tâche τ_2 est exécutée au temps $t = 8$ durant deux *slots* et au temps $t = 18$ durant un *slot*.

Un scénario permet d'évaluer le comportement de l'algorithme de placement en fonction des ajouts et suppressions de tâches. Plus précisément, les placements successifs des tâches peuvent engendrer une fragmentation de la zone reconfigurable, ce que l'algorithme de placement doit limiter.

En considérant l'exemple illustré par la figure 4.21, au temps $t = 0$, la tâche τ_3 doit être placée. L'algorithme de placement détermine une position pour une instance de la tâche τ_3. La zone reconfigurable est partiellement occupée par cette instance jusqu'au temps $t = 16$, où la tâche τ_3 termine son exécution. L'espace de la zone reconfigurable occupé par cette instance est alors libéré.

Le choix d'une instance et d'une position dépend de l'état de la zone reconfigurable. Par exemple, à la première exécution de la tâche τ_0 (à $t = 4$), la zone reconfigurable est occupée par les tâches τ_1 et τ_3. Lors de la deuxième exécution de la tâche τ_0 (à $t = 11$), la zone reconfigurable est occupée par les tâches τ_1, τ_3 et τ_4. L'état de la zone est donc différent entre ces deux exécutions. L'algorithme sera donc probablement amené à choisir des instances et/ou des positions différentes.

Dans la suite des expérimentations, nous générons aléatoirement des scénarios en paramétrant le nombre de tâches ainsi que que le nombre de *slots* du scénario.

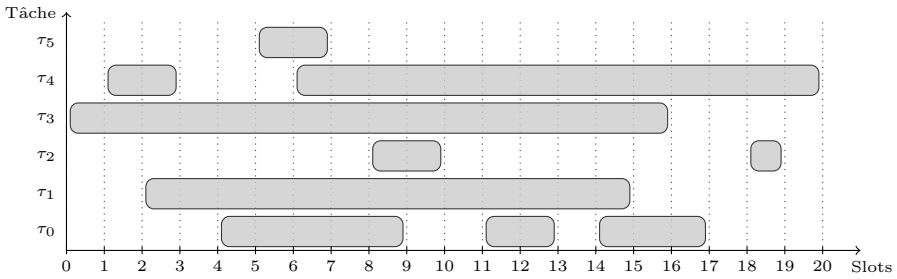


FIGURE 4.21 – Exemple d'un scénario d'exécution de tâches.

4.2.4.2 Comparaison du nombre de tâches rejetées

Dans cette section, une évaluation de la qualité de notre algorithme est présentée. Afin d'estimer sa qualité, nous le comparons à l'algorithme *SUP Fit* sur un ensemble de scénarios générés aléatoirement. Ces scénarios comportent 40 *slots* et à chaque *slot* correspond une action d'ajout ou de suppression de tâches.

Dans ces expérimentations, une zone reconfigurable modélisée par une grille de région de dimension 30×30 est utilisée. Chaque jeu de tâches possède 20 tâches, chacune étant décrite par un nombre d'instances spécifié dans les résultats que nous présentons. La taille et la position des instances sont déterminées aléatoirement.

La figure 4.22 présente des résultats de simulation de scénarios. La qualité de ces simulations est mesurée par le pourcentage de tâches placées. La qualité du placement dépendant du nombre d'instances des tâches, la figure 4.22 présente des résultats pour des tâches disposant de une à neuf instances, et le placement de chacun de ces jeux de tâches est effectué par trois algorithmes, à savoir, l'algorithme Colonne décrit précédemment, l'algorithme *SUP Fit*, noté *SUP Fit* 1 dans la figure 4.22 et une variante de l'algorithme *SUP Fit*, noté *SUP Fit* *, où le nombre d'instances est augmenté artificiellement. Ainsi, pour une instance d'une tâche, nous générons les instances décrivant toutes les positions verticales valides de cette instance. L'algorithme *SUP Fit* * dispose ainsi des mêmes possibilités de placement que notre algorithme de placement en colonne.

La figure 4.22 présente donc les résultats de ces trois algorithmes en fonction du nombre d'instances. Les résultats des différents algorithmes dépendant des tailles et positions des instances, nous testons chaque algorithme sur 100 jeux de tâches. Par exemple, 100 jeux de tâches, dont chaque tâche est décrite par une instance, ont été générés. La figure 4.22 présente le pourcentage moyen de tâches placées.

De manière générale, nous pouvons observer que le pourcentage de tâches placées croît lorsque le nombre d'instances augmente. En effet, plus il y a d'instances de tâches, plus l'algorithme de placement a de liberté lors du choix des positions des tâches à placer.

Nous pouvons également constater que l'algorithme *SUP Fit* 1 obtient des résultats très nettement inférieurs aux deux autres algorithmes. Le pourcentage moyen de

tâches placées obtenu par l'algorithme *SUP Fit* 1, sur l'ensemble des jeux de tâches présentés par la figure 4.22, est égal à 54, alors qu'il est environ égal à 78 pour les deux autres algorithmes. Cette différence de qualité provient essentiellement du faible nombre d'instances considérées par l'algorithme *SUP Fit* 1. En effet, lorsque le nombre d'instances est plus important, l'algorithme *SUP Fit* fournit des résultats similaires à notre algorithme de placement en colonnes. Ainsi, le pourcentage moyen de tâches placées, sur l'ensemble de jeux de tâches, par l'algorithme *SUP Fit* * est égal à 78.2, et celui de l'algorithme Colonne à 77.5.

Les résultats obtenus par l'algorithme *SUP Fit* * et Colonne indiquent que les heuristiques issues de l'algorithme *SUP Fit* et de l'algorithme *quad corner* fournissent des résultats comparables. Cependant, le nombre d'instances considérées par ces deux algorithmes n'est pas du tout identique. Dans la section suivante, nous présentons en détail cette différence.

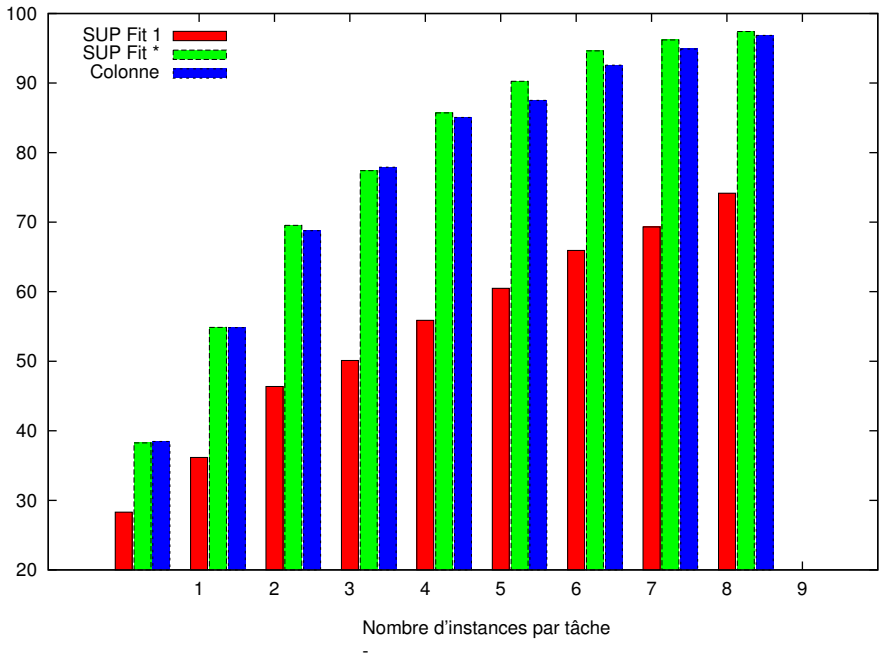


FIGURE 4.22 – Comparaison du pourcentage de tâches placées par l'algorithme *SUP Fit* 1,*SUP Fit* * et Colonne. Dans ces simulations, nous considérons des scénarios comportant 20 tâches et 40 *slots*. Les tâches disposent d'un nombre variable d'instances indiquées en abscisse.

4.2.4.3 Comparaison du nombre d'instances réelles

Dans la section précédente, nous avons constaté que les algorithmes *SUP Fit* * et Colonne fournissent des résultats similaires. Le problème majeur de l'algorithme *SUP Fit* est que les instances de tâches qu'il considère ont toutes une position fixe,

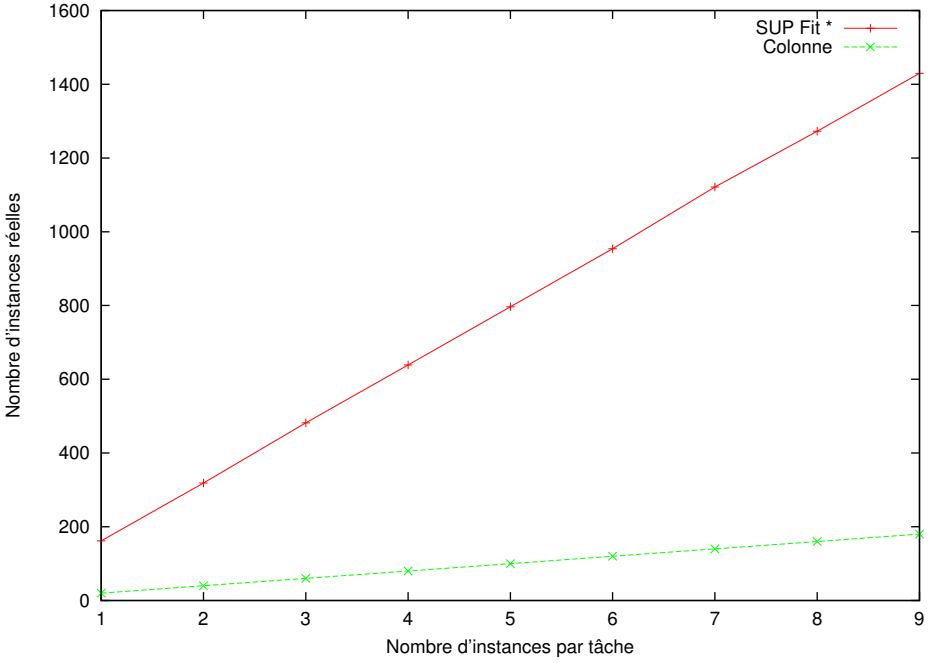


FIGURE 4.23 – Comparaison du nombre d'instances utilisées par l'algorithme *SUP Fit* * et Colonne lors des simulations présentées par la figure 4.22.

contrairement à l'algorithme Colonne qui dispose de la possibilité de placer une instance dans toute la colonne de la zone reconfigurable.

Nous avons vu que l'algorithme *SUP Fit* * est une variante de l'algorithme *SUP Fit* où nous avons créé des instances supplémentaires afin d'obtenir une liberté de placement équivalente à celle de l'algorithme Colonne. La figure 4.2.4.3 présente le nombre d'instances réelles utilisées par l'algorithme *SUP Fit* * comparé à celui de l'algorithme Colonne. Plus précisément, cette figure présente le nombre moyen d'instances sur l'ensemble des 100 jeux, pour chaque type de jeux de tâches.

Nous pouvons observer que le nombre d'instances considérées par l'algorithme *SUP Fit* * est très nettement plus important que celui de l'algorithme Colonne. Par exemple, pour les jeux décrits par neuf instances, le nombre d'instances moyen est égal à 180 pour l'algorithme Colonne (20 tâches étant chacune décrite par neuf instances). Pour les mêmes jeux de tâches, le nombre moyen d'instances réelles est égal à 1429, ce qui correspond à un facteur d'environ huit.

En fait, ce facteur est lié à la hauteur des instances et des colonnes. Par exemple, si l'on considère l'exemple illustré par la figure 4.17, c'est à dire une zone reconfigurable ayant un nombre de régions verticales égal à cinq et une instance nécessitant deux régions verticales, il est nécessaire d'ajouter trois instances. Sur cet exemple, le facteur serait égal à trois.

4.2.5 Conclusion

Dans cette section, nous avons présenté un nouveau modèle de tâches utilisable par un ordonnanceur spatial pour architectures hétérogènes reconfigurables. Malgré l'utilisation d'un algorithme simple, les expérimentations ont montré que les résultats sont équivalents à ceux obtenus au moyen de l'algorithme *SUP Fit*. En revanche, lorsque la taille de la zone reconfigurable augmente, l'algorithme *SUP Fit* requiert un nombre d'instances beaucoup plus important que notre algorithme. Ainsi, les structures de données manipulées par notre algorithme sont bien moins importantes et favorisent une utilisation en ligne de l'algorithme de placement. Nous avons donc présenté un modèle de tâches permettant de développer de nouveaux algorithmes de placement pour architectures reconfigurables hétérogènes.

Chapitre 5

Optimisation et tolérance aux fautes des réseaux de neurones de Hopfield

Ce chapitre présente des travaux relatifs aux réseaux de neurones de Hopfield dans un cadre général. Nous proposons ainsi des idées applicables aux réseaux utilisés dans les chapitres précédents. La première partie de ce chapitre présente une méthode de parallélisation de l'évaluation ayant pour but d'optimiser le temps d'évaluation d'un réseau. La seconde partie présente quant à elle des propriétés de tolérance aux fautes des réseaux de Hopfield.

Ces deux techniques ont un intérêt à la fois dans un contexte général, mais également dans un contexte d'implémentation matérielles au sein des architectures SOC. L'évaluation parallèle de neurones va permettre d'améliorer les temps d'exécution des réseaux de neurones, ce qui s'avère très intéressant dans un contexte d'exécution en-ligne. De plus, l'évolution des architectures et des technologies conduit à des systèmes dont on maîtrise de moins en moins bien le bon fonctionnement. Typiquement, les technologies futures conduiront probablement à des circuits pour lesquels la présence de fautes devra être gérée par le circuit lui-même. La définition de systèmes tolérants aux fautes devient alors un challenge important qui va permettre d'envisager « vivre avec des fautes »dans le système.

5.1 Évaluation parallèle d'un réseau de neurones de Hopfield

Dans cette section, nous présentons une méthode permettant de réduire le temps de convergence d'un réseau de neurones de Hopfield tout en préservant sa convergence. Cette accélération est obtenue en évaluant plusieurs neurones simultanément [23].

La première section présente le mode d'évaluation parallèle et notament l'équation d'évaluation de ce mode. Ces équations permettent de developper une preuve de

convergence d'un réseau de neurones de Hopfield lorsque des neurones sont évalués en parallèle. De cette preuve découlent des contraintes spécifiant quels neurones peuvent être évalués simultanément. Ces neurones sont alors groupés en paquets de neurones indépendants. La deuxième section présente la construction des paquets sur un exemple simple d'ordonnanceur temporel. Cet exemple est utilisé dans la section suivante afin d'évaluer l'accélération obtenue par notre méthode.

5.1.1 Mode d'évaluation parallèle

Dans la suite de cette section, les scalaires sont notés en minuscule, les vecteurs en majuscule et les matrices en gras. Par exemple, x_1 dénote le premier élément d'un vecteur X, alors que X_1 dénote le premier sous vecteur d'un vecteur X.

Afin d'évaluer des neurones simultanément, le vecteur d'état X est partitionné en k sous-vecteurs, que nous nommerons *blocs*, tel que

$$X^T = \left[X_1^T, X_2^T, \ldots, X_k^T\right], \tag{5.1}$$

où X^T est le vecteur transposé du vecteur X. Le vecteur T et la matrice de connexion $\mathbf{W}$ sont partitionnés de la même manière.

$$T = \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_k \end{bmatrix} \mathbf{W} = \begin{bmatrix} \mathbf{W}_{11} & \mathbf{W}_{12} & \ldots & \mathbf{W}_{1k} \\ \mathbf{W}_{21} & \mathbf{W}_{22} & \ldots & \mathbf{W}_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{W}_{k1} & \mathbf{W}_{k2} & \ldots & \mathbf{W}_{kk} \end{bmatrix} \tag{5.2}$$

Les neurones appartenant à un même bloc X_b ($b \in [1, k]$) sont tous évalués simultanément. Le parallélisme apparaît donc au sein de l'évaluation d'un bloc. Les blocs, quant à eux, sont évalués séquentiellement, et l'ordre d'évaluation des blocs n'impacte pas la propriété de convergence du réseau, mais peut influer sur le temps de convergence. Comme dans le cas d'une évaluation purement séquentielle, nous pouvons donc utiliser un ordre d'évaluation déterminé ou aléatoire. De plus, la taille des blocs peut être quelconque - le nombre de blocs l'est donc également - permettant ainsi d'adapter le grain de parallélisme. Nous verrons dans la suite que la taille des blocs est tout de même soumise à certaines contraintes.

Afin de simplifier les notations, nous considérons une évaluation séquentielle déterminée des blocs, définie par leur numérotation. Les blocs sont donc évalués en commençant par le bloc X_1 jusqu'au bloc X_k.

Parce que les notations introduites par ce mode d'évaluation sont relativement complexes, nous présentons l'évaluation d'un bloc sur un exemple simple, en l'occurence, un réseau composé de trois neurones et partitionné en deux blocs. Nous généraliserons ensuite à un réseau de taille n partionné en k blocs.

5.1.1.1 Sur un exemple simple

Dans un premier temps, nous illustrons la mise à jour d'un bloc sur l'exemple d'un réseau contenant trois neurones décrits par les objets

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad T = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \qquad \mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix}, \tag{5.3}$$

avec $w_{i,j}$ indiquant le poids de la connexion du neurone x_i au neurone x_j.

Ce réseau est partitionné en deux blocs, le premier bloc est composé des deux premiers neurones, le deuxième bloc, du dernier neurone. L'équation (5.4) présente un vecteur d'état X composé de deux sous vecteurs ainsi que le vecteur de seuil T et la matrice de connexion $\mathbf{W}$ correspondant à ce partitionnement.

$$X = \begin{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ \begin{bmatrix} x_3 \end{bmatrix} \end{bmatrix} \qquad T = \begin{bmatrix} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \\ \begin{bmatrix} t_3 \end{bmatrix} \end{bmatrix} \qquad \mathbf{W} = \begin{bmatrix} \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix} & \begin{bmatrix} w_{1,3} \\ w_{2,3} \end{bmatrix} \\ \begin{bmatrix} w_{3,1} & w_{3,2} \end{bmatrix} & \begin{bmatrix} w_{3,3} \end{bmatrix} \end{bmatrix} \tag{5.4}$$

Afin de bien comprendre les équations caractérisant le mode d'évaluation parallèle, nous les construisons pas à pas. L'équation (5.5) décrit l'évaluation du premier bloc (contenant les neurones x_1 et x_2). L'évaluation de ces neurones est basée sur l'équation (2.7). Ainsi, les neurones x_1 et x_2 sont évalués à l'instant $t+1$ en parallèle, en utilisant l'état de tous les neurones constituant le réseau à l'instant t.

$$\begin{bmatrix} x_1(t+1) & x_2(t+1) \end{bmatrix} = H' \left(\begin{bmatrix} x_1(t) * w_{1,1} + x_2(t) * w_{2,1} + x_3(t) * w_{3,1} - t_1 \\ x_1(t) * w_{1,2} + x_2(t) * w_{2,2} + x_3(t) * w_{3,2} - t_2 \end{bmatrix}^T \right) \tag{5.5}$$

où la fonction H' est simplement la fonction H (2.1) appliquée à des vecteurs, et définie par

$$H'(V) = [H(v_1) \;\; H(v_2) \;\; \ldots \;\; H(v_n) \;], \tag{5.6}$$

avec V est un vecteur de dimension n. L'équation (5.5) peut être réecrite sous forme matricielle

$$\begin{bmatrix} x_1(t+1) & x_2(t+1) \end{bmatrix} = H' \Bigg(\begin{bmatrix} x_1(t) & x_2(t) & x_3(t) \end{bmatrix} * \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{bmatrix} - \begin{bmatrix} t_1 & t_2 \end{bmatrix} \Bigg). \tag{5.7}$$

Dans l'équation (5.7), les sous vecteurs et sous matrices du partitionnement décrit par l'équation (5.4) apparaissent. En réecrivant les objets caractérisant ce réseau sous la forme

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}, \qquad T = \begin{bmatrix} T_1 \\ T_2 \end{bmatrix}, \qquad \mathbf{W} = \begin{bmatrix} \mathbf{W}_{1,1} & \mathbf{W}_{1,2} \\ \mathbf{W}_{2,1} & \mathbf{W}_{2,2} \end{bmatrix}, \tag{5.8}$$

où

$$X_1 = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad X_2 = \begin{bmatrix} x_3 \end{bmatrix} \quad T_1 = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix}, \quad T_2 = \begin{bmatrix} t_3 \end{bmatrix}, \tag{5.9}$$

$$\begin{aligned} \mathbf{W}_{1,1} &= \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}, \quad \mathbf{W}_{1,2} = \begin{bmatrix} w_{1,3} \\ w_{2,3} \end{bmatrix}, \\ \mathbf{W}_{2,1} &= \begin{bmatrix} w_{3,1} & w_{3,2} \end{bmatrix}, \quad \mathbf{W}_{2,2} = \begin{bmatrix} w_{3,3} \end{bmatrix}, \end{aligned} \tag{5.10}$$

l'équation (5.7) est alors exprimée par

$$X_1^T(t+1) = H'\left(\begin{bmatrix} X_1^T(t) & X_2^T(t) \end{bmatrix} \times \begin{bmatrix} \mathbf{W}_{1,1} \\ \mathbf{W}_{2,1} \end{bmatrix} - T_1^T \right). \tag{5.11}$$

En développant l'équation (5.11), l'évaluation du bloc X_1 devient

$$X_1^T(t+1) = H'\left(X_1^T(t) \times \mathbf{W}_{1,1} + X_2^T(t) \times \mathbf{W}_{2,1} - T_1^T \right). \tag{5.12}$$

5.1.1.2 Généralisation

Afin de généraliser ce développement, il est nécessaire de définir une opération d'addition agissant sur des vecteurs, notée $\sum$. En considérant un vecteur V de dimension n dont les éléments sont tous des vecteurs de même dimension, nous définissons l'opération $\sum$ comme suit

$$\sum_{i=1}^{n} V_i = V_1 + V_2 + \ldots + V_n. \tag{5.13}$$

En introduisant l'opérateur $\sum$, l'équation (5.12) devient

$$X_1^T(t+1) = H'\left(\sum_{i=1}^{2} X_i^T(t) \times \mathbf{W}_{i,1} - T_1^T \right). \tag{5.14}$$

Afin de généraliser l'équation d'évaluation à un réseau composé d'un nombre arbitraire de blocs, nous considérons le vecteur d'état décrit par l'équation (5.1), ainsi que le vecteur de seuil et la matrice de connexion décrit par l'équation (5.2).

En considérant un ordre d'évaluation basé sur les indices des blocs, l'équation d'évaluation des neurones d'un bloc quelconque X_b est définie par

$$\begin{aligned} X_b^T(t+1) = H'\Bigg(&\sum_{i=1}^{b-1} X_i^T(t+1) \times \mathbf{W}_{ib} \\ &+ \sum_{i=b}^{k} X_i^T(t) \times \mathbf{W}_{ib} - T_b^T \Bigg), \end{aligned} \tag{5.15}$$

où $X_b(t)$ dénote l'évaluation du bloc X_b à l'itération t. L'évaluation du bloc X_b au temps $t+1$ requiert les valeurs des blocs $X_1, \ldots, X_{b-1}$ au temps $t+1$ et les valeurs des blocs $X_b, \ldots, X_k$ au temps t.

5.1.2 Convergence dans le cas parallèle

Dans le chapitre 2, nous avons vu que, sous certaines contraintes, la convergence d'un réseau de neurones de Hopfield est prouvée. Nous pouvons rappeler que dans le cas d'une évaluation séquentielle, la convergence est assurée [35]

– si la matrice de connexion est symétrique, et
– si ses éléments diagonaux sont positifs ou nuls.

Parce que l'équation d'évaluation des neurones est différente du cas séquentiel, il est nécessaire de s'assurer de la convergence du réseau sous cette méthode d'évaluation. Dans le contexte des mémoires associatives, les auteurs de [41] prouvent la convergence d'un réseau de Hopfield évalué parallèlement. Ils énoncent ainsi le théorème suivant : « lorsque le réseau est évalué parallèlement, la convergence est assurée si la matrice $\mathbf{W}$ est symétrique et si les blocs diagonaux $\mathbf{W}_{bb}$ sont positifs ou égaux à zéro ».

De même que dans le chapitre 2, le théorème de Lyapunov est utilisé pour prouver la convergence du réseau en montrant la décroissance de la fonction d'énergie associée au réseau.

Pour montrer la décroissance de la fonction d'énergie, le signe de la différence entre deux itérations est déterminé. Sans aucune perte de généralités, nous considérons que le premier bloc du vecteur d'état X est évalué. Afin de simplifier les notations, nous réécrivons le vecteur d'état X, le vecteur de seuil T ainsi que la matrice de connexion $\mathbf{W}$ de la manière suivante

$$X = \begin{bmatrix} X_1 \\ X' \end{bmatrix}, \mathbf{W} = \begin{bmatrix} \mathbf{W}_{11} & \mathbf{C}^T \\ \mathbf{C} & \mathbf{W}' \end{bmatrix}, T = \begin{bmatrix} T_1 \\ T' \end{bmatrix}. \tag{5.16}$$

Il est important de remarquer que nous considérons la matrice $\mathbf{W}$ comme étant symétrique. Cette symétrie est exprimée par les sous-matrices $\mathbf{C}$ et $\mathbf{C}^T$.

À partir de l'équation d'énergie (2.12), et de l'équation (5.16), nous exprimons la différence d'énergie entre deux itérations successives t et $t+1$ sous la forme :

$$\begin{aligned}
\Delta(E(X)) &= E(X(t+1)) - E(X(t)) \\
&= -\tfrac{1}{2}\left(\left[X_1^T(t+1), X'^{\,T}(t)\right] \times \begin{bmatrix} \mathbf{W}_{11} & \mathbf{C}^T \\ \mathbf{C} & \mathbf{W}' \end{bmatrix} \times \begin{bmatrix} X_1(t+1) \\ X'(t) \end{bmatrix}\right) \\
&\quad - \left[X_1^T(t+1), X'^{\,T}(t)\right] \times \begin{bmatrix} T_1 \\ T' \end{bmatrix} \\
&\quad - \left(-\tfrac{1}{2}\left(\left[X_1'(t), X'^{\,T}(t)\right] \times \begin{bmatrix} \mathbf{W}_{11} & \mathbf{C}^T \\ \mathbf{C} & \mathbf{W}' \end{bmatrix} \times \begin{bmatrix} X_1(t) \\ X'(t) \end{bmatrix}\right)\right. \\
&\quad \left. - \left[X_1^T(t), X'^{\,T}(t)\right] \times \begin{bmatrix} T_1 \\ T' \end{bmatrix}\right) \\
&= -\overbrace{\left[X_1^T(t+1) - X_1^T(t)\right]}^{A_1}\overbrace{\left[\mathbf{W}_{11}X_1(t) + \mathbf{C}^T X'(t) + T_1\right]}^{A_2} \\
&\quad -\tfrac{1}{2}\overbrace{\left[X_1^T(t+1) - X_1^T(t)\right]}^{B_1^T}\mathbf{W}_{11}\overbrace{\left[X_1(t+1) - X_1(t)\right]}^{B_1}.
\end{aligned} \tag{5.17}$$

Si la différence $\Delta(E(X))$ est négative, alors la convergence du réseau est prouvée dans le cas d'une évaluation parallèle de neurones groupés selon l'équation (5.16). Nous allons donc étudier les différents cas. Dans un premier temps, nous éliminons le cas où $X_1^T(t+1) = X_1^T(t)$ car il existe au moins une valeur de k tel que $X_k^T(t+1) \neq X_k^T(t)$ sans quoi un point fixe serait déjà atteint. Dans la suite, nous considérons donc que $X_1^T(t+1) \neq X_1^T(t)$, ainsi, $X_1^T(t+1) - X_1^T(t)$ possède des éléments égaux à -1 ou à 1.

Si les produits $A_1 \times A_2$ et $B_1^T \times W_{11} \times B_1$ sont tous deux positifs, alors la différence $\Delta(E(X))$ est négative et la fonction $E(X)$ est donc prouvée décroissante. Dans la suite, nous étudions séparément le signe de ces deux produits.

5.1.2.1 Signe du produit $A_1 \times A_2$ de l'équation (5.17)

Afin de déterminer le signe du produit $A_1 \times A_2$, nous allons étudier deux cas : lorsqu'un élément de $X_1^T(t+1)$ est égal à 1 et lorsqu'un élément de $X_1^T(t+1)$ est égal à 0. À partir de l'équation (5.15), nous avons

$$X_1(t+1) = H'(\mathbf{W}_{11}X_1(t) + C^T X'(t) + T_1) = H'(A_2). \tag{5.18}$$

D'après l'équation (5.6), si un élément i de $X_1^T(t+1)$ est égal à 0, alors l'élément i de A_2 est négatif. Parce que nous supposons que $X_1^T(t+1) \neq X_1^T(t)$, quand un élément i de $X_1^T(t+1)$ est égal à 0, l'élément i de $X_1^T(t)$ est égal à 1, et l'élément i de A_1 est alors négatif (égal à -1). Dans ce cas, A_1 et A_2 sont tous deux négatifs et leur produit est positif.

Nous avons montré que si un élément i de $X_1^T(t+1)$ est égal à 0, alors le produit $A_1 \times A_2$ est positif. Un raisonnement analogue peut être appliqué à un élément i de $X_1^T(t+1)$ égal à 1.

Nous pouvons donc conclure que si $X_1^T(t+1) \neq X_1^T(t)$, les éléments de $A_1 \times A_2$ sont tous positifs.

5.1.2.2 Signe du produit $B_1^T \times \mathbf{W}_{11} \times B_1$ de l'équation (5.17)

Le signe d'un élément du produit $B_1^T \times \mathbf{W}_{11} \times B_1$ dépend du signe de $\mathbf{W}_{11}$. Si tous les éléments de $\mathbf{W}_{11}$ sont positifs, alors les éléments du produit $B_1^T \times \mathbf{W}_{11} \times B_1$ sont également positifs. Nous voyons qu'il a été nécessaire d'introduire une contrainte supplémentaire sur les blocs diagonaux de la matrice de connexion $\mathbf{W}$. En effet, pour garantir la convergence du réseau, nous imposons que les blocs diagonaux de $\mathbf{W}$ soient positifs.

Dans les deux sous-sections précédentes, nous avons montré que les produits $A_1 \times A_2$ et $B_1^T \times \mathbf{W}_{11} \times B_1$ sont positifs (sous certaines contraintes). Ainsi, le signe de $\Delta(E(X))$ est négatif et donc la fonction d'énergie est strictement négative jusqu'à ce qu'un point fixe soit atteint. Par le théorème de Lyapunov, nous pouvons conclure que le réseau, dont des blocs sont évalués parallèlement, converge

– si la matrice de connexion $\mathbf{W}$ est symétrique, et
– si les blocs diagonaux $\mathbf{W}_{bb}$ sont positifs.

Ce théorème permet d'établir des conditions suffisantes à la garantie de la convergence, mais ces conditions ne sont pas nécessaires. En effet, il est possible que des conditions moins contraignantes permettent également d'assurer la convergence.

5.1.3 Construction des paquets

La section précédente a permis d'établir des contraintes assurant la convergence d'un réseau de Hopfield dont des neurones sont évalués parallèlement. Nous avons pu observer que ces contraintes étaient relativement proches des contraintes associées à une évaluation classique. Comme lors d'une évaluation purement séquentielle, la matrice doit être symétrique. Cependant, alors qu'il était suffisant que les éléments diagonaux de la matrice soient nuls (absence d'auto-connexion), dans le cas d'une évaluation parallèle, il faut que les blocs diagonaux de la matrice de connexion soient positifs ou nuls.

La figure 5.1 présente un exemple de matrice de connexion. Nous pouvons observer que cette matrice possède des blocs diagonaux dont les éléments sont égaux à zéro. D'après la section précédente, des neurones ayant des connexions positives ou nulles peuvent être évalués simultanément. Nous pouvons donc construire trois paquets de neurones : $\{1, 2\}$, $\{3\}$ et $\{4, 5, 6\}$. Les neurones appartenant à un même paquet pourront être évalués simultanément tandis que les paquets sont évalués séquentiellement.

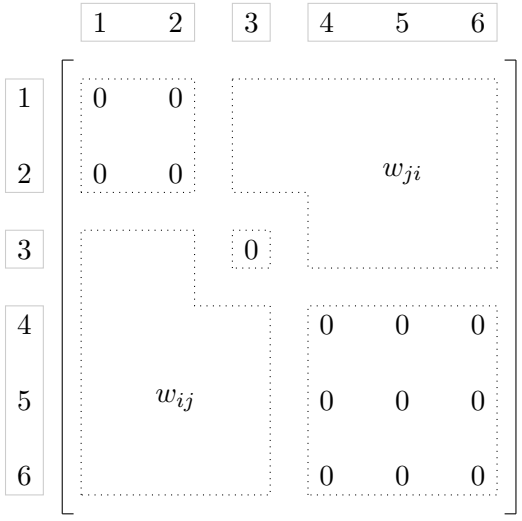


FIGURE 5.1 – Exemple d'une matrice de connexion. Les éléments diagonaux ont des valeurs égales à zéro tandis que les autres elements w_{ij} sont négatifs ou nuls.

Généralement, lors de la construction d'un réseau de Hopfield, nous ne créons pas de connexions positives. Dans la suite, les matrices de connexions considérées ne posséderont que des éléments ayant des valeurs négatives ou nulles. Parce qu'une connexion entre deux neurones ayant une valeur nulle est équivalente à une absence de connexion, il s'agira donc de déterminer des ensembles de neurones n'étant pas connectés. La section suivante présente la construction de paquets sur un problème d'ordonnancement.

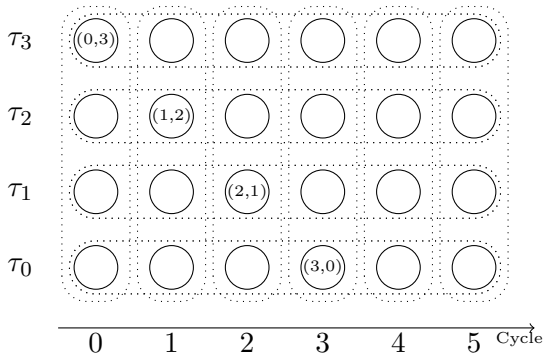


FIGURE 5.2 – Représentation graphique des règles k-de-n appliquées aux réseaux.

5.1.3.1 Application à un problème d'ordonnancement

Afin d'illustrer l'évaluation parallèle d'un réseau de neurones de Hopfield, nous considérons dans la suite un problème d'ordonnancement temporel pour architectures mono-processeur tel que défini dans [59]. Ce problème d'ordonnancement consiste simplement à déterminer à quel *slot* doivent être exécutées des tâches préemptives soumises au système. Ainsi, l'ordonnanceur doit attribuer à chaque tâche τ_i un nombre de *slots* égal au WCET C_i. De plus, l'ordonnanceur doit veiller à n'exécuter qu'une tâche à chaque *slot*. Afin de pouvoir ordonnancer toutes les tâches, nous considérons une fenêtre d'ordonnancement égale à la somme des WCET de toutes les tâches soumises.

La figure 5.2 présente un exemple de réseau de neurones de Hopfield modélisant un problème d'ordonnancement disposant d'une fenêtre d'ordonnancement égal à six *slots* et de quatre tâches. Le réseau de neurones associé à ce problème possède 6×4 neurones. L'activation d'un neurone est interprétée comme l'exécution de la tâche lui étant associée, au *slot* correspondant.

Dans ces expérimentations, nous réutilisons le réseau décrit à la section 3.1.1. Nous appliquons donc des règles k-de-n représentées par des rectangles pointillés sur la figure 5.2. Ainsi, les neurones appartenant à un même rectangle sont tous connectés les uns aux autres.

Dans la section 5.1.3, nous avons vu que les neurones pouvant être évalués en parallèle sans altérer la convergence du réseau doivent être non connectés. La figure 5.2 nous permet de constater que les neurones placés diagonalement ne sont pas connectés. Nous pouvons donc grouper ces neurones au sein de paquets. Par exemple, les neurones $(3, 0)$, $(2, 1)$, $(1, 2)$ et $(0, 3)$ peuvent former un paquet et ces neurones seront évalués simultanément.

5.1.3.2 Construction des paquets : généralisation

Dans cette section, nous généralisons la méthode de construction des paquets. La création des paquets peut être exprimée à travers un problème de nombre de cliques couvrantes [53].

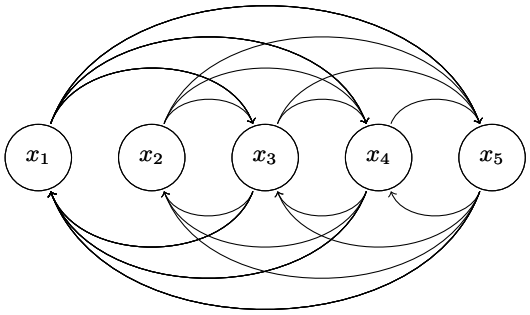


FIGURE 5.3 – Exemple d'un réseau de neurones composé de cinq neurones. Les neurones x_1 et x_2 ne sont pas connectés entre eux.

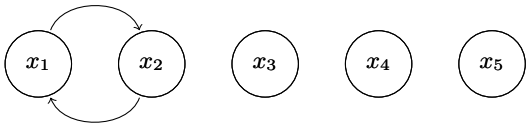


FIGURE 5.4 – Graphe complémentaire à celui décrit par la figure 5.3. Seuls les neurones x_1 et x_2 sont connectés.

En guise d'exemple, nous considérons un réseau composé de cinq neurones. La figure 5.3 représente ces cinq neurones ainsi que leurs connexions. Nous pouvons constater que les neurones x_1 et x_2 ne sont pas connectés entre eux. Dans la suite, nous notons G le graphe associé à ce réseau de neurones.

La figure 5.4 présente le graphe complémentaire G' du graphe G décrit par la figure 5.3. Les graphes G et G' sont complémentaires si et seulement si deux sommets adjacents du graphe G ne sont pas adjacents dans le graphe G'. Ainsi, dans G', seuls les neurones x_1 et x_2 sont connectés. À partir du graphe complémentaire G', nous pouvons effectuer une recherche de cliques pour obtenir des paquets de neurones non connectés.

Cet exemple de réseau étant particulièrement simple, nous obtenons quatre paquets de neurones non connectés tels que représentés par la figure 5.5. Les neurones ayant la même couleur peuvent être évalués simultanément sans altérer la convergence du réseau. Nous obtenons donc quatre paquets formés par les neurones $\{x_1, x_2\}$, $\{x_3\}$, $\{x_4\}$ et $\{x_5\}$.

Le problème de création de paquets correspond à un problème de découverte de cliques. Plus précisément, si l'on dispose d'un nombre infini de ressources d'exécution, il s'agit de trouver le nombre minimal de cliques dans le graphe complémentaire G'. En effet, moins il y a de cliques, plus le réseau est évalué rapidement. Par exemple, si l'on considère le réseau décrit par la figure 5.2 et composé de cinq neurones, le nombre minimal de cliques est égal à quatre. Il y a ainsi quatre paquets de neurones à évaluer à chaque itération. Il serait également possible d'évaluer chaque neurone séparement

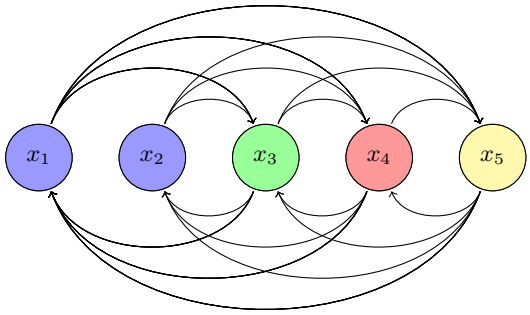


FIGURE 5.5 – Représentation des paquets de neurones. Les neurones ayant la même couleur peuvent être évalués simultanément. Des neurones de couleur différentes sont évalués séquentiellement.

mais dans ce cas, chaque itération nécessite l'évaluation de cinq paquets (chacun étant composé d'un unique neurone). Nous voyons donc que le nombre minimal de cliques permet de réduire le nombre d'évaluation séquentielles nécessaires à la réalisation d'une itération.

Le nombre de neurones composant un paquet peut être trop important par rapport aux nombres de ressources nécessaires à l'évaluation de l'ensemble des neurones d'un paquet. Dans ce cas, il convient de restreindre la taille des paquets afin d'être en mesure d'évaluer simultanément tous les neurones d'un paquet. Nous voyons donc que différentes stratégies de création de paquets peuvent être utilisées. Dans la suite, nous chercherons systématiquement le plus petit nombre de paquets. Nous supposerons donc disposer d'un nombre suffisant de ressources.

5.1.4 Expérimentation

Dans cette section, nous présentons une comparaison entre l'évaluation séquentielle et parallèle d'un réseau de neurones de Hopfield afin de montrer les améliorations apportées par notre méthode de parallélisation. En guise d'exemple, nous considérons le problème d'ordonnancement présenté dans la section 5.1.3.1.

La première étape consiste à déterminer des paquets de neurones pouvant être évalués simultanément. La figure 5.6 présente la taille des paquets en fonction du nombre de neurones. Dans notre exemple, la taille optimale des paquets correspond au nombre de tâches soumises au système. Dans ces expérimentations, nous générons des jeux de n tâches, où chacune de ces n tâches possède un WCET déterminé aléatoirement dans l'intervalle $[1, 5]$. La fenêtre d'ordonnancement est égale à la somme des WCET de l'ensemble des tâches soumises, afin de garantir l'existence d'un ordonnancement valide. La fenêtre d'ordonnancement possède donc un nombre de *slots* appartenant à l'intervalle $[n, 5 \times n]$. Ainsi, le nombre de neurones nécessaires à la modélisation neuronale de ce

problème d'ordonnancement appartient à l'intervalle $[n^2, 5 \times n^2]$. La figure 5.6 décrit les jeux de tâches utilisés dans la suite des expérimentations.

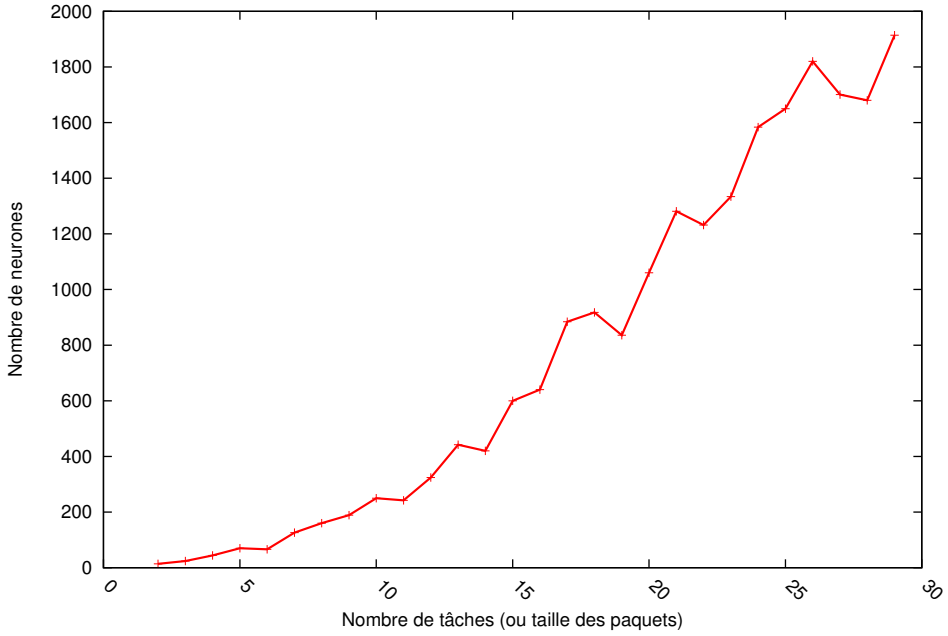


FIGURE 5.6 – Nombre de neurones par rapport à la taille des paquets. Comme la taille des paquets est égale au nombre de tâches, l'abcisse représente le nombre de tâches et la taille des paquets.

Métrique utilisée. La métrique utilisée dans nos expérimentations est le nombre d'évaluations de paquets. Lorsque l'évaluation séquentielle est utilisée, un paquet contient un unique neurone. Nous considérons donc disposer de suffisamment de ressources pour évaluer un paquet de neurones aussi rapidement qu'un neurone. Dans ce cas, le temps d'évaluation du réseau de neurones est strictement lié au nombre d'évaluations de paquets.

La figure 5.7 présente des résultats d'évaluations séquentielles et parallèles. Quel que soit le mode d'évaluation, les paquets sont évalués dans un ordre aléatoire. Chaque simulation s'arrête lorsque le réseau a atteint un état stable. Nous pouvons observer que le nombre d'évaluations n'est pas strictement croissant avec le nombre de neurones. En effet, le nombre d'itérations nécessaires à l'obtention d'une solution est également dépendant des données du problème. La tendance globale montre tout de même que le nombre d'évaluations croît avec le nombre de neurones.

La figure 5.7 montre que l'accélération est vraiment importante et croit avec l'augmentation du nombre de neurones. Par exemple, lorsque le réseau possède environ 2000 neurones, la version parallèle est 38 fois plus rapide que la version séquentielle.

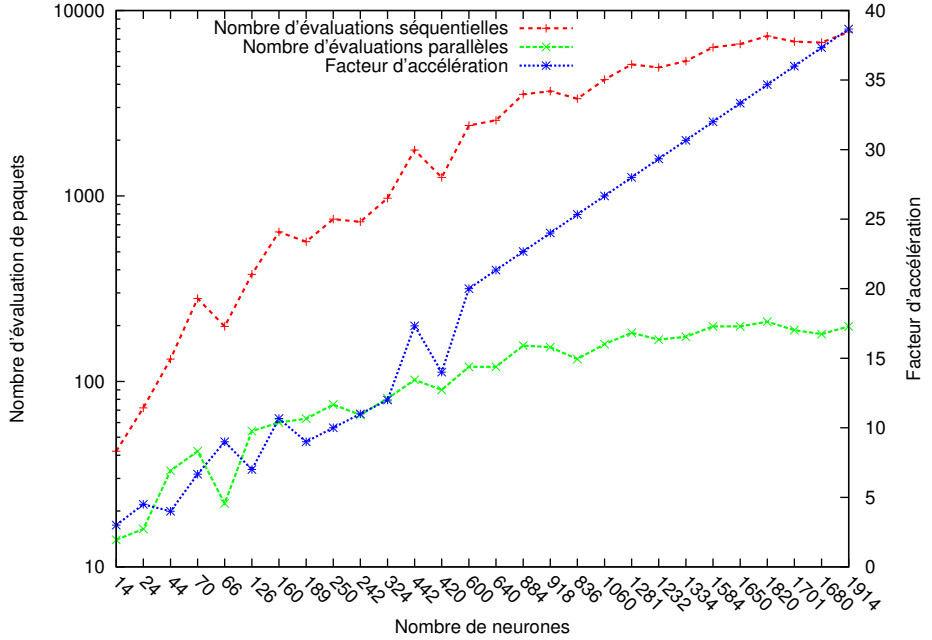


FIGURE 5.7 – Nombre d'évaluations de paquets dans les cas séquentiel et parallèle.

Le facteur d'accélération présenté par le figure 5.7 suppose que la cible d'exécution de l'ordonnanceur dispose de suffisamment de ressources pour évaluer tous les neurones simultanément, sans quoi il serait nécessaire de réduire la taille des paquets, ce qui engendrerait une augmentation du temps d'évaluation du réseau.

D'après la section 5.1.3.1, le nombre de neurones composant un paquet parallèle est égal au nombre de tâches t. En considérant un nombre n de neurones, le nombre de paquets est égal $\frac{n}{t}$. Le nombre de paquets séquentiels étant égal au nombre de neurones, la version séquentielle possède donc t fois plus de paquets. Le facteur d'accélération entre la version parallèle et la version séquentielle est donc proche d'un facteur t.

La figure 5.8 présente le facteur d'accélération par rapport au nombre de tâches. La droite représente la fonction identité, à savoir, $f(t) = t$, correspondant à une accélération linéaire fonction du nombre de tâches t. Nous pouvons observer que l'accélération s'avère supérieure, dans certains cas, à cette droite. Ce phénomène est dû au fait que notre méthode d'évaluation parallèle peut générer un ordre d'évaluation des neurones plus favorable, ce qui réduit le nombre d'itérations nécessaires à la stabilisation du réseau. En effet, l'évaluation diagonale des neurones favorise leur activation car à chaque *slot*, seul un neurone peut être activé. Un paquet étant constitué de neurones diagonaux, de nombreux neurones peuvent être activés lors des premières itérations.

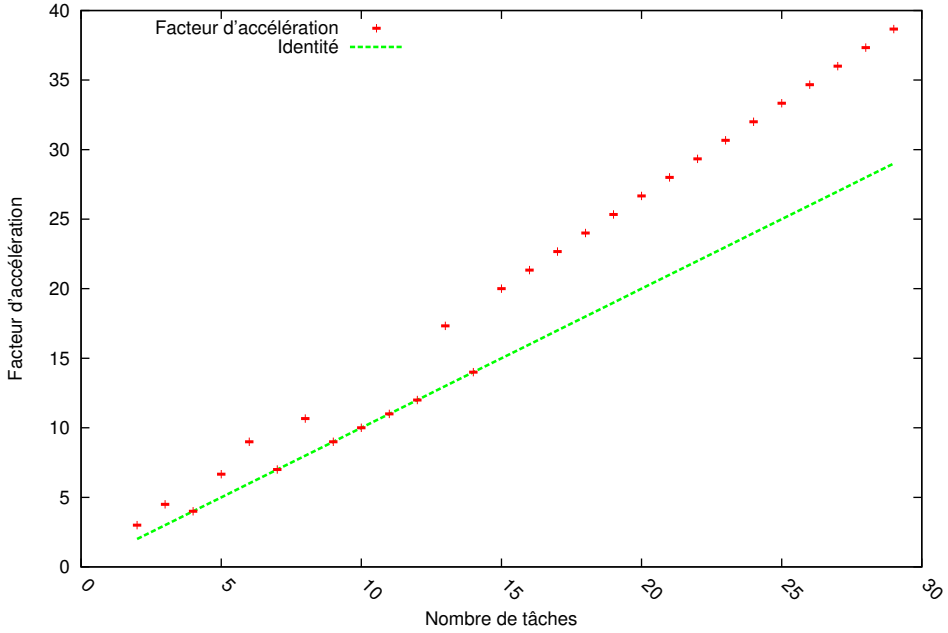


FIGURE 5.8 – Accélération par rapport au nombre de tâches.

5.1.5 Conclusion

Dans cette première partie de chapitre a été présentée une méthode permettant d'évaluer simultanément plusieurs neurones tout en conservant la convergence de celui-ci, et cette propriété a été prouvée dans le cas d'une évaluation parallèle. Nous avons montré, sur un exemple simple, qu'une accélération très importante du temps de convergence peut être obtenue à l'aide de notre méthode. De plus, la création des paquets de neurones évalués simultanément peut être automatisée en étant exprimée sous la forme d'un problème de découverte de cliques.

Cette méthode d'évaluation est applicable aux réseaux que nous avons proposés dans les chapitres précédents. Par exemple, nous avons présenté dans la section 4.1 un réseau de neurones réalisant un ordonnanceur. Les neurones associés à des instances non chevauchantes et à de tâches différentes peuvent être évalués simultanément tout en garantissant la convergence du réseau.

Les cibles d'exécution visées dans nos travaux sont des architectures pourvues d'un grain de parallélisme très fin permettant d'évaluer simultanément de nombreux neurones. Nous sommes donc en mesure d'évaluer des paquets de grande taille afin d'obtenir des facteurs d'accélération très importants.

5.1.6 Perspectives

Dans les sections précédentes, les neurones sont groupés en paquets ayant une taille fixe, et l'accélération dépend directement de la taille de ces paquets. Nous avons vu

que, pour garantir la convergence du réseau, les neurones appartenant à un même paquet sont tous non connectés. Cependant, certains réseaux peuvent posséder trop de connexions pour que le partitionnement soit efficace. Dans ce cas, il serait possible que la taille des paquets évolue au fur et à mesure des itérations.

Dans un premier temps, des paquets de neurones non connectés sont créés comme cela fut proposé dans les sections précédentes. Si ces paquets sont trop petits, alors nous fusionnons plusieurs paquets en un. Au fur et à mesure des itérations, les paquets sont divisés jusqu'à obtenir l'ensemble initial de paquets. Cette méthode d'évaluation est donc composée de deux phases. Durant la première phase, la convergence du réseau n'est pas garantie car des neurones connectés sont évalués simultanément. La deuxième phase, quant à elle, consiste en une évaluation parallèle garantissant la convergence.

Cette évolution de la taille des paquets permet d'accélérer les temps d'évaluation du réseau lors des premières itérations tout en garantissant la convergence du réseau. Cependant, l'évaluation simultanée de neurones connectés mène à des solutions non valides. La qualité des solutions générées lors des premières itérations est donc dégradée, ce qui risque d'augmenter le nombre d'itérations nécessaires à l'obtention d'une solution valide. Il s'agit donc d'effectuer des simulations pour s'assurer que l'augmentation du nombre d'itérations soit compensée par l'accélération de l'évaluation du réseau.

5.2 Tolérance aux fautes

L'évolution technologique de fabrication des circuits intégrés conduit petit à petit vers des technologies si fines qu'il devient difficile de garantir une homogénéité des transistors sur l'ensemble de la surface de silicium. Cette évolution pousse alors les concepteurs à minorer les performances des systèmes pour tenir compte de cette variabilité en se basant sur le pire cas. Malheureusement, cette variabilité va se traduire progressivement par des cas extrêmes conduisant au non fonctionnement de certains transistors du système, provoquant alors la disparition de certaines fonctionnalités ou un défaut plus général du système. L'une des questions qui se pose alors est de savoir comment contourner ce type de problème et parvenir à assurer un fonctionnement malgré l'apparition de fautes. Il est bien entendu possible d'intégrer des techniques de détection et de corrections d'erreurs, mais il va devenir de plus en plus crucial d'intégrer des techniques de tolérance aux fautes dans ces systèmes. Cette évolution amène aujourd'hui les concepteurs à proposer des architectures innovantes intégrant directement cette caractéristique.

Dans un réseau de neurones, à l'instar du cerveau humain, la défaillance d'un ou plusieurs neurones dans un système global peut généralement être corrigée par les autres neurones. Sans disposer de techniques de détection de fautes et de correction d'erreurs, les réseaux de neurones disposent en effet de cette capacité à produire des solutions valides même en présence de dysfonctionnements internes. Pour parvenir à rendre tolérant aux fautes les réseaux de neurones, deux techniques peuvent être utilisées : soit des neurones cachés sont ajoutés, soit la méthode d'apprentissage est modifiée pour tenir compte des fautes qui vont survenir [70, 40]. L'étude que nous proposons dans cette section a pour objet de montrer que les réseaux de neurones de type Hopfield sont tout à fait en mesure d'assurer la résolution d'un problème d'optimisation, et cela malgré des défaillances au sein même de la structure du réseau de neurones.

L'organisation de cette partie de chapitre est la suivante, la section 5.2.1 présente des fautes pouvant survenir dans une structure de type réseaux de neurones. La section 5.2.2 montre comment, malgré l'apparition de fautes au sein des réseaux de neurones, ceux-ci peuvent toutefois produire des solutions valides. La définition du nombre de fautes supportées, en fonction des fautes qui se produisent, est alors donnée et permet de définir deux contraintes sous lesquelles le réseau continue à converger vers une solution valide. La section 5.2.3 présente des résultats de simulation permettant de préciser le comportement d'un réseau de neurones résolvant un problème d'ordonnancement de tâches sur architecture monoprocesseur.

5.2.1 Modèle de fautes au sein d'un réseau de neurones de Hopfield

Pour illustrer les capacités de tolérances aux fautes des réseaux de neurones, nous étudions dans un premier temps la robustesse de la règle k-de-n appliquée à un ensemble de neurones subissant des défaillances. Soit un ensemble de N neurones sur lequel une règle k-de-n est appliquée. Sur un tel réseau, le nombre de neurones est égal à N, le nombre d'entrées du réseau est égal N et le nombre de connexions entre neurones est

égal à $N \times (N-1)$.

Dans cette étude, nous considérons des fautes matérielles provoquant un blocage de l'état des neurones. Nous considérons les trois types de fautes suivantes.

- Sur un neurone : une faute sur la fonctionnalité du neurone lui même, qui est figé à une valeur donnée. On parlera alors de faute de collage correspondant soit à un état actif permanent soit à un état inactif permanent du neurone.
- Sur une entrée : lorsqu'une faute sur une entrée d'un neurone provoque un apport d'énergie trop important, le neurone sera bloqué à l'état actif lors de sa première évaluation. Inversement, si trop peu d'énergie est apportée au neurone, celui-ci sera bloqué à l'état inactif. Sous l'hypothèse que le neurone défaillant ait été évalué au moins une fois, ces deux cas se ramènent à une faute de collage, à savoir un état actif ou inactif permanent en fonction de la défaillance survenant sur l'entrée du neurone.
- Sur une connexion : une faute sur une connexion engendrant un apport d'énergie négative trop important mène à une désactivation permanente du neurone cible. Inversement, une faute engendrant un apport d'énergie positive trop important mène à une activation permanente du neurone cible. Ainsi, un neurone évalué pourvu d'une connexion défaillante engendrant un apport très important d'énergie, qu'elle soit négative ou positive, se voit figé dans un état respectivement inactif ou actif.

Lors de la présentation précédente des fautes sur les entrées et sur les connexions, nous avons exposé des fautes pouvant être ramenées à un collage du neurone cible. Nous avions précisé que l'énergie apportée par une entrée ou une connexion doit être importante pour être ramenée à un collage. Par exemple, dans le cas d'une règle k-de-n, l'entrée non défaillante des neurones a comme valeur $2k-1$. D'après l'équation (2.2) d'évaluation d'un neurone, si la valeur de l'entrée défaillante est strictement inférieure à 0, alors le neurone possédant cette entrée défectueuse est bloqué à l'état inactif. Si la valeur de l'entrée défaillante est supérieure ou égale à $2(n-1)$, ce neurone est bloqué à l'état actif. A contrario, des entrées défaillantes ayant une valeur appartenant à l'intervalle $]0, 2(n-1)]$ peuvent engendrer des changements d'état. Nous restreignons notre étude à des entrées défaillantes n'appartenant pas à cet intervalle afin de pouvoir assimiler ces entrées défaillantes à des collages de neurones. Dans la suite, nous n'évoquerons donc plus que des collages de neurones.

5.2.2 Maintien de la convergence d'un réseau de Hopfield malgré les défaillances

Dans cette section, nous établissons le nombre de fautes qu'un réseau de neurones de Hopfield peut tolérer sans altérer sa convergence. Nous étudions les cas de collage de neurones à 0 ou 1.

Soit un ensemble ε_N composé de N neurones sur lequel une règle k-de-n est appliquée, et soit un sous ensemble $\varepsilon'_{N_{D_1}}$ composé de N_{D_1} neurones défaillants et collés à 1,

avec $\varepsilon'_{N_{D_1}} \subset \varepsilon_N$. Alors la fonction d'énergie relative à la règle k-de-n (équation (2.19)) appliquée à l'ensemble ε_N devient

$$\begin{aligned} E &= \left(\sum_{x_i \in \varepsilon_N} x_i - k \right)^2 \\ &= \left(\left(\sum_{x_i \in \left(\varepsilon_N - \varepsilon'_{N_{D_1}} \right)} x_i + \sum_{x_i \in \varepsilon'_{N_{D_1}}} x_i \right) - k \right)^2 . \end{aligned} \tag{5.19}$$

Les neurones $x_i \in \varepsilon'_{N_{D_1}}$ étant collés à l'état actif (c'est-à-dire bloqués dans l'état 1), alors $\sum_{x_i \in \varepsilon'_{N_{D_1}}} x_i = N_{D_1}$. L'équation (5.19) devient

$$E = \left(\sum_{x_i \in \left(\varepsilon_N - \varepsilon'_{N_{D_1}} \right)} x_i + N_{D_1} - k \right)^2 .$$

Sachant que seuls les neurones contenus dans l'ensemble $\varepsilon_N - \varepsilon'_{N_{D_1}}$ peuvent changer d'état, pour que cette fonction d'énergie puisse être minimisée et ramenée à zéro, il faut que le terme $\sum_{xi \in \varepsilon'_{N_{D_1}}} x_i - k$ soit négatif, c'est-à-dire que $N_{D_1} - k \leq 0$. En effet, la fonction E atteint son minimum si le système d'équation suivant est respecté

$$\begin{array}{l} \left\{ \begin{array}{lcl} \sum_{x_i \in \left(\varepsilon_N - \varepsilon'_{N_{D_1}} \right)} x_i + N_{D_1} - k & = & 0 \\ \sum_{x_i \in \left(\varepsilon_N - \varepsilon'_{N_{D_1}} \right)} x_i & \geq & 0 \end{array} \right. \\ \Rightarrow N_{D_1} - k \leq 0 \end{array} \tag{5.20}$$

Du système d'équation 5.20, nous déduisons la contrainte

$$C_1 : \qquad N_{D_1} \leq k \tag{5.21}$$

indiquant qu'il ne faut pas que le sous ensemble de neurones défaillants $\varepsilon'_{N_{D_1}}$ contienne plus de k neurones.

Le même raisonnement peut être mené pour des défaillances de collage dans l'état inactif des neurones. Soit un sous ensemble $\varepsilon'_{N_{D_0}}$ composé de N_{D_0} neurones défaillants et collés à 0, avec $\varepsilon'_{N_{D_0}} \subset \varepsilon_N$. Alors la fonction d'énergie relative à la règle k-de-n

appliquée à l'ensemble ε_N s'écrit alors :

$$\begin{aligned} E &= \left(\sum_{xi \in \varepsilon_N} x_i - k \right)^2 \\ &= \left(\left(\sum_{xi \in \left(\varepsilon_N - \varepsilon'_{N_{D_0}} \right)} x_i + \sum_{xi \in \varepsilon'_{N_{D_0}}} x_i \right) - k \right)^2 . \end{aligned} \tag{5.22}$$

Les neurones $x_i \in \varepsilon'_{N_{D_0}}$ étant collés à l'état inactif (c'est-à-dire dans l'état 0), alors $\sum_{xi \in \varepsilon'_{N_{D_0}}} x_i = 0$. Sachant que seuls les neurones contenus dans l'ensemble $\varepsilon_N - \varepsilon'_{N_{D_0}}$ peuvent changer d'état, pour que cette fonction d'énergie puisse être minimisée et ramenée à zéro, il faut que le terme $\sum_{xi \in \left(\varepsilon_N - \varepsilon'_{N_{D_0}} \right)} x_i - k$ puisse être annulé. Pour assurer cela, il faut que l'ensemble $\varepsilon_N - \varepsilon'_{N_{D_0}}$ contienne au moins k éléments. Cela se traduit par $Card(\varepsilon_N) - Card(\varepsilon'_{N_{D_0}}) \geq k$, soit $N - N_{D_0} \geq k$. La contrainte s'écrit donc

$$C_2 : \qquad N_{D_0} \leq N - k \tag{5.23}$$

Toute combinaison de défaillances pour laquelle les contraintes C_1 et C_2 sont respectées permet d'assurer la convergence du réseau vers une solution possible au problème de l'activation de k neurones parmi N.

5.2.3 Résultats

Dans cette section, nous exposons quelques expérimentations relatives à la tolérance aux fautes telle que présentée dans les sections précédentes. Nous considérons un réseau réalisant une règle k-de-n dont nous faisons évoluer le paramètre k. Nous présentons dans un premier temps le nombre de fautes tolérées en fonction de différentes valeurs de k. Ensuite, le nombre d'itérations nécessaire à la convergence du réseau est mis en relation avec le nombre de fautes dans le réseau.

5.2.3.1 Nombre de fautes tolérées

Dans la figure 5.9, nous montrons le nombre de défaillances de type collage à 0 ou collage à 1 qu'un réseau composé de 100 neurones est capable de supporter. Comme le stipule les contraintes C_1 (5.21) et C_2 (5.23), le nombre de défaillances tolérées est dépendant de k, c'est-à-dire du nombre de neurones actifs souhaités pour la convergence. La zone grisée correspond à la zone pour laquelle le nombre de défaillances ne remet pas en cause la convergence du réseau.

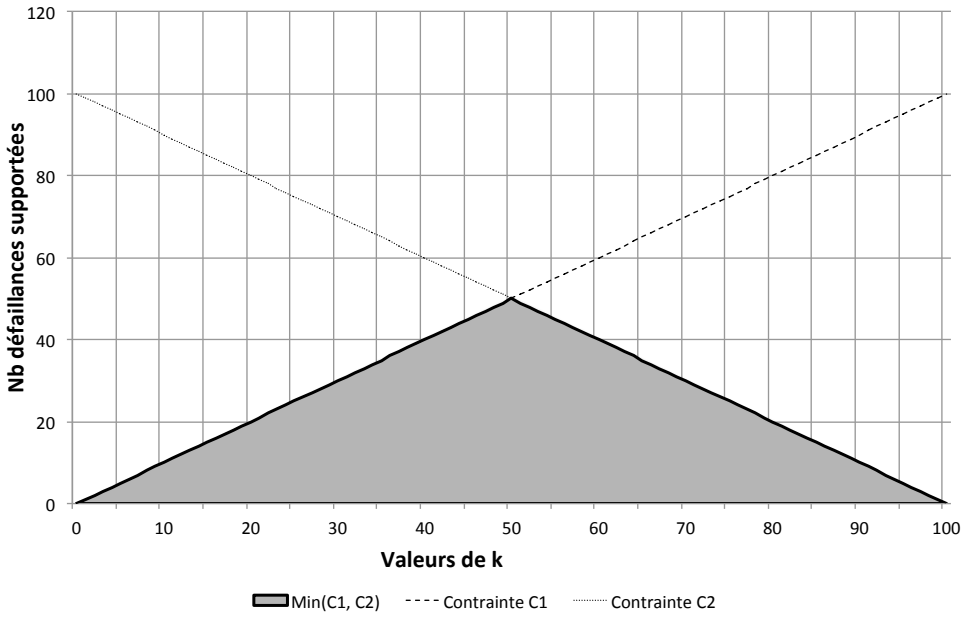


FIGURE 5.9 – Nombre de défaillances supportées par une règle k-de-n pour $N = 100$ et k variant de 0 à 100.

5.2.3.2 Impact du nombre de fautes sur le temps de convergence

La figure 5.10 illustre l'impact des défaillances sur le temps de convergence. Le réseau utilisé pour ces résultats est constitué de 100 neurones, une règle 50-de-100 lui est appliquée, et le nombre de défaillances varie de 0 à 50 défaillances de collage à 1, puis de collage à 0, puis de collage à la fois à 1 et à 0. La figure montre que lorsque le nombre de fautes reste relativement faible (quelques neurones défaillants), alors le réseau parvient sans difficulté à converger vers une solution valide. On notera aussi que le nombre de cycles pour atteindre la convergence est peu impacté, notamment lorsque les défauts sont à la fois des collages à 1 et des collages à 0, ce qui est plus probable que le cas de collages uniquement à 0 ou uniquement à 1.

5.2.4 Conclusion

Les travaux présentés dans cette partie illustrent les capacités de tolérance aux fautes des réseaux de neurones de Hopfield. Ces réseaux sont capables de produire des solutions à un problème d'optimisation pour peu que l'on sache modéliser le problème et définir l'ensemble des solutions. Dans cette étude nous avons montré que la défaillance d'une partie de la structure du réseau de neurones ne remet pas en cause la production de solutions valides. Nous avons défini le modèle de fautes qui peut survenir dans une structure de Hopfield, et nous avons illustré la capacité de tolérance aux fautes au travers d'une structure de réseau de neurones simple sur laquelle nous appliquons une règle de convergence particulière. Nous avons alors montré sous quelles conditions de

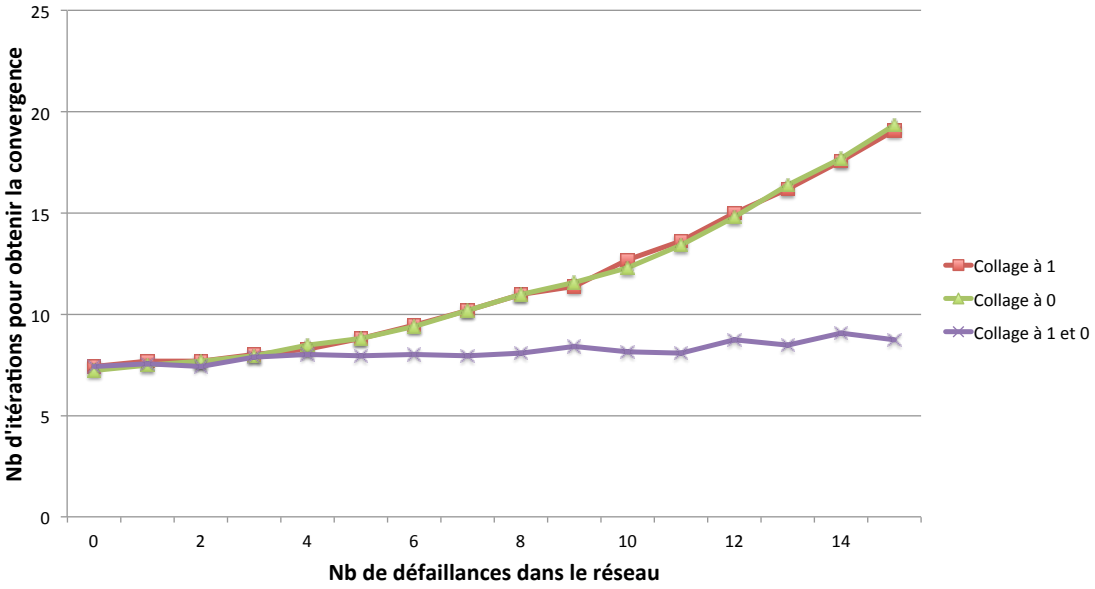


FIGURE 5.10 – Impact sur le temps de convergence en cas de défaillances au sein du réseau de neurones.

défaillance ce type de réseau peut continuer à produire des résultats valides. Un exemple de convergence de réseau de neurones en présence de fautes a été présenté et nous avons montré que la convergence est peu sensible au nombre de défaillances au sein du réseau, lorsque ce nombre de fautes reste raisonnable.

Conclusion

Synthèse des travaux

Ce document a présenté nos travaux relatifs à l'ordonnancement pour architectures hétérogènes partiellement reconfigurables. Nos ordonnanceurs étant principalement basés sur des réseaux de neurones de Hopfield, nous avons également mené des travaux sur ce type d'algorithmes. Ainsi, nous avons présenté les trois axes majeurs de nos travaux, à savoir, l'ordonnancement temporel, l'ordonnancement spatial et les réseaux de neurones de Hopfield.

Concernant l'ordonnancement temporel, nous avons proposé un ordonnanceur supportant des architectures de type SOC hétérogènes. Cet ordonnanceur est basé sur un réseau de Hopfield dont nous avons modifié le fonctionnement afin de réduire le nombre d'itérations nécessaire à sa convergence. Nous avons ainsi introduit le concept de « neurone inhibiteur ». Afin d'évaluer la qualité de notre ordonnanceur par rapport à une solution connue, nous l'avons comparé à l'algorithme d'ordonnancement pour architectures homogènes nommé Pfair. Bien que notre algorithme soit destiné à des architectures hétérogènes, nous obtenons une qualité d'ordonnancement très proche de celle obtenue avec Pfair dont l'optimalité a été démontrée. Dans le cadre d'un SOC hétérogène, nous présentons une application de notre ordonnanceur sur un exemple concret.

Notre ordonnanceur temporel supportant une architecture hétérogène, il permet d'ordonnancer temporellement une application sur un FPGA. Cependant, il s'avère encore nécessaire de déterminer un emplacement pour les tâches à exécuter. L'état de l'art des algorithmes de placement a permis de mettre en évidence le manque de travaux tenant compte de l'hétérogénéité des architectures reconfigurables. Nous proposons donc deux ordonnanceurs répondant à cette problématique. Ces deux ordonnanceurs ont pour point commun de considérer plusieurs implémentations d'une même tâche. Grâce à cette technique, le premier ordonnanceur que nous avons proposé ne requiert aucun mécanisme de relocation. Cet ordonnanceur nous permet d'envisager une implémentation neuronale résolvant le problème de placement. Le second ordonnanceur, quant à lui, utilise la relocation en la restreignant à des zones possédant les mêmes ressources. Les différentes instances d'une tâche permettant alors de la placer en des endroits composés de ressources différentes. Les résultats produits par ces deux ordonnnanceurs ont été

ensuite comparés à l'algorithme de placement SUP Fit. Concernant l'algorithme basé sur un réseau de neurones, nous obtenons un nombre de tâches placées comparable à celui obtenu par l'algorithme SUP Fit tout en bénéficiant des qualités intrinsèques - implémentation matérielle efficace et tolérance aux fautes - des réseaux de Hopfield. Concernant l'algorithme *Colonne*, celui-ci permet de réduire de façon très importante le nombre d'instances nécessaire à l'obtention d'une solution de qualité identique.

En vue d'améliorer la rapidité d'exécution de nos ordonnnanceurs, nous avons mené des travaux visant à réduire les temps d'évaluation d'un réseau de neurones de Hopfield. Nous avons étudié deux axes. Le premier axe concerne l'évaluation simultanée de plusieurs neurones. Nous avons précisé les contraintes à respecter pour garantir la convergence du réseau. Nous avons ensuite illustré ce mode d'évaluation sur un exemple simple où nous avons obtenu une réduction très importante du nombre d'itérations nécessaire à la convergence. Le second axe présente des propriétés de tolérance aux fautes des réseaux de Hopfield. Nous avons en effet observé qu'en dessous d'un certain taux de fautes, telles que des collages de neurones, le réseau continue de converger vers des solutions valides. Nous avons donc défini un modèle de fautes et déterminé le nombre de fautes tolérables sur un exemple simple. L'augmentation de la finesse de gravure des circuits engendrant un nombre croissant de fautes, la propriété de tolérance aux fautes des réseaux de Hopfield permettrait donc d'exploiter des architectures pouvant être partiellement défectueuses tout en garantissant la validité des solutions obtenues, et ce, sans recourir à des mécanismes complexes de détection/correction d'erreurs.

Perspectives

Nous présentons dans cette dernière section trois pistes de travaux futurs. Dans la continuité des travaux présentés dans ce document, l'implémentation matérielle de réseaux de Hopfield est sans doute la piste prioritaire. Nous avons ainsi débuté des travaux présentés brièvement dans la suite. La seconde piste concerne le placement de plusieurs tâches simultanément en vue d'améliorer la qualité du placement. Finalement, nous évoquons une extension de notre réseau de neurones de placement en vue d'allouer aux tâches les ressources d'une architecture SOC 3D.

Implémentation matérielle d'un réseau de neurones de Hopfield

Les ordonnanceurs que nous avons développés à partir de réseaux de neurones visant à être implémentés matériellement, nous avons débuté des travaux d'implémentation sur FPGA [51]. Dans des travaux antérieurs, nous avions développé un ordonnanceur temporel pour architectures partiellement reconfigurables, nommé RANN (*Reconfigurable Artificial Neural Network*) [14]. Cet ordonnanceur se distingue d'un ordonnanceur multiprocesseur par sa capacité à s'assurer que la surface totale des tâches exécutées ne soit pas supérieure à la surface disponible sur l'architecture considérée.

Nous avons réalisé des synthèses de réseaux RANN sur le circuit XC5VFX70T de la famille Virtex 5 de chez *Xilinx*. Ainsi, les figures 5.11 et 5.12 présente les résultats en

fréquence et en surface en fonction du nombre de neurones composant le réseau RANN.

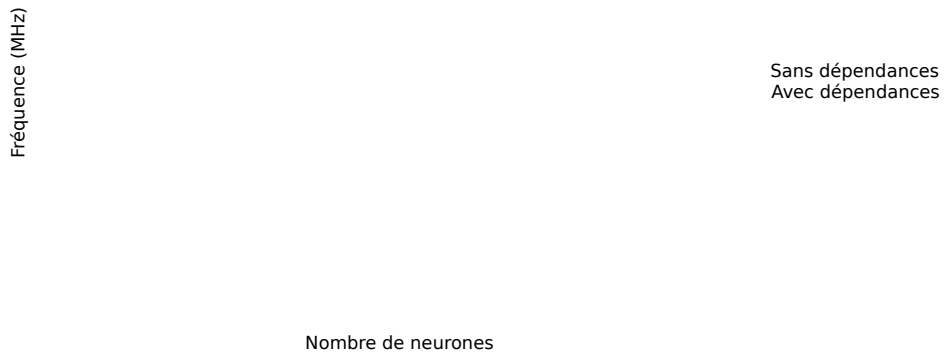


FIGURE 5.11 – Performances temporelles du réseau de neurones RANN sur un circuit Virtex 5 XC5VFX70TF.

Dans ces travaux, nous avons développé une description VHDL d'un neurone générique en s'efforçant de réduire au maximum les calculs complexes. Nous avons ainsi pu supprimer les opérations d'additions et de multiplications. Dans la continuité de ces travaux, nous souhaitons désormais implémenter notre ordonnanceur spatial afin d'obtenir une solution matérielle d'ordonnancement spatio-temporelle par réseaux de neurones de Hopfield.

Placement de tâches par ensemble

Dans un premier temps, nous exposons des situations susceptibles de bénéficier du placement simultané de plusieurs tâches. Un premier cas de placement d'un ensemble de tâches se produit lorsque plusieurs tâches d'une application ont à s'exécuter simultanément. La figure 5.13 présente l'exemple d'un graphe de tâches menant à l'exécution simultanée de deux tâches. Les tâches τ_2 et τ_3 dépendant du résultat de la tâche τ_1, lorsque la tâche τ_1 a terminé son exécution, les tâches τ_2 et τ_3 peuvent démarrer. Dans ce scénario, l'ordonnanceur peut alors placer ces deux tâches simultanément.

Une seconde opportunité apparaît dans les travaux concernant la défragmentation de la zone reconfigurable [58, 45]. À un instant donné, les tâches sont préemptées pour être déplacées sur la zone reconfigurable afin d'améliorer le placement. Dans ce contexte, il s'avère également profitable d'être capable de placer un ensemble de tâches simultanément.

Finalement, la figure 5.14 présente un scénario d'exécution illustrant la possibilité du placement d'un ensemble de tâches dans un contexte temps réel. Nous considérons les trois tâches utilisées précédemment auxquelles nous associons une durée d'exécution ainsi qu'une période. Notons que les périodes de ces trois tâches sont égales à six unités

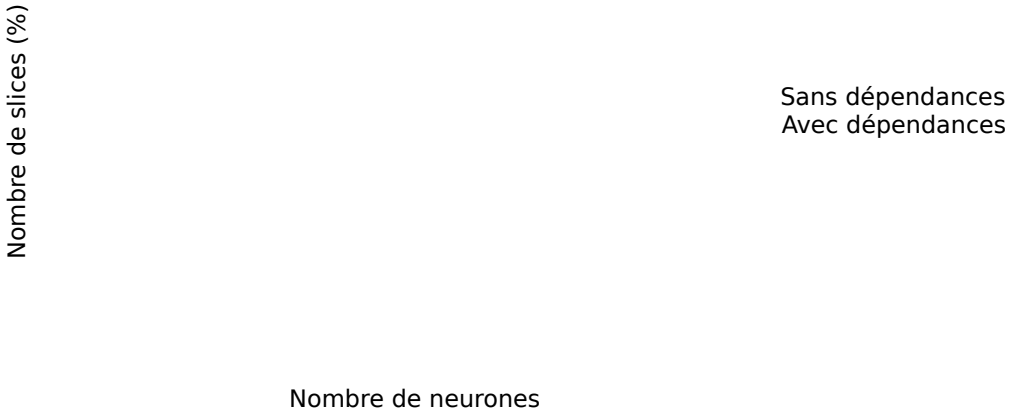


FIGURE 5.12 – Surfaces occupées par le réseau de neurones RANN sur un circuit Virtex 5 XC5VFX70TF.

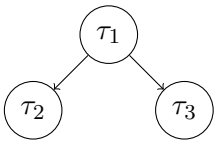


FIGURE 5.13 – Exemple d'un graphe de tâche composé de trois tâches. Les tâches τ_2 et τ_3 dépendent du résultats de la tâche τ_1.

de temps et sont en phase. Au temps $t = 2$, la tâche τ_1 est soumise au système. L'ordonnanceur détermine donc un emplacement pour τ_1 au temps $t = 2$. Au temps $t = 3$, l'ordonnanceur détermine un emplacement pour la tâche τ_2, et au temps $t = 4$ un emplacement pour la tâche τ_3. Pour l'instant, les tâches τ_1, τ_2 et τ_3 ont été placées indépendamment au fur et à mesure de leur arrivée. Au temps $t = 8$, l'ordonnanceur doit ré-exécuter la tâche τ_1. Cependant, il a désormais connaissance des moments d'exécution des tâches τ_2 et τ_3, l'ordonnanceur est donc en mesure de déterminer les emplacements de τ_1, τ_2 et τ_3 simultanément afin d'améliorer la qualité du placement comme nous l'avons vu précédemment.

Nous avons présenté des situations où il serait possible de placer plusieurs tâches simultanément. Dans la suite, nous montrons comment cette démarche peut améliorer la qualité du placement. Classiquement, le placement de tâches sur une architecture reconfigurable consiste à placer des tâches les unes après les autres. Si n tâches sont soumises simultanément au système, les algorithmes de placement classiques se comportent comme des algorithmes gloutons, à savoir, ils sont appelés successivement n fois et le placement de chaque tâche n'est jamais remis en cause. L'ordre de traitement des

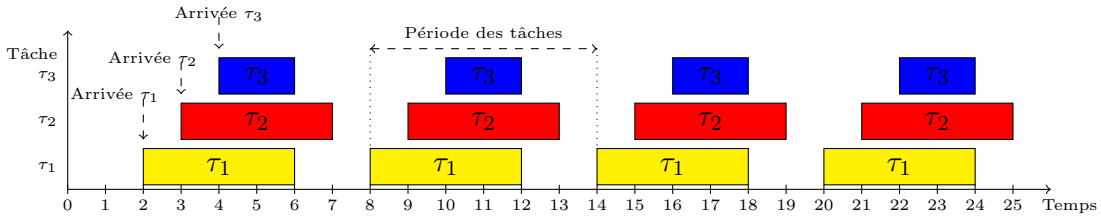


FIGURE 5.14 – Scénario d'exécution de trois tâches périodiques.

tâches influera sur la qualité du placement et à notre connaissance, aucun algorithme de placement n'intègre un mécanisme permettant de choisir un ordre adéquat de façon dynamique.

La figure 5.15 présente un exemple de placement illustrant les avantages du placement d'un ensemble de tâches. Pour cet exemple, nous considérons une heuristique simple de placement consistant à minimiser le nombre de rectangles vides. Les sous-figures (a), (b) et (c) décrivent les placements successifs des tâches τ_1, τ_2 et τ_3. La sous-figure (d) représente le placement optimal de ces trois tâches. Nous pouvons constater que le placement successif des tâches a mené à une solution sous-optimale en termes de nombre de rectangles vides. Cet exemple montre que l'ordre de placement des tâches à une importance quant à la qualité de la solution obtenue. En étant capable de placer un ensemble de tâches, notre ordonnanceur serait alors en mesure de générer une solution plus proche de la solution optimale.

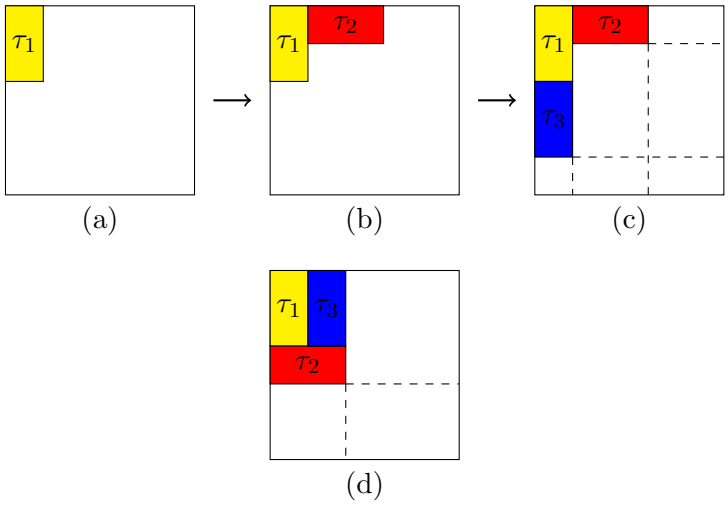


FIGURE 5.15 – Les sous-figures (a), (b) et (c) décrivent le placement successif des tâches τ_1, τ_2 et τ_3. La sous-figure (d) représente le placement optimal en termes de nombre de rectangles vides.

Allocation de ressources pour SOC 3D

L'une des évolutions architecturales des SOC envisagées concerne la conception de systèmes 3D. La figure 5.16 présente un schéma décrivant ce type d'architectures. Lors de l'exécution d'une application sur ce type d'architecture, il faudra déterminer qu'elles sont les ressources à allouer aux tâches composant l'application. Afin d'optimiser les communications entre les tâches, il convient de leur allouer des ressources géographiquement proches. Dans un premier temps, le modèle de communications considéré pour

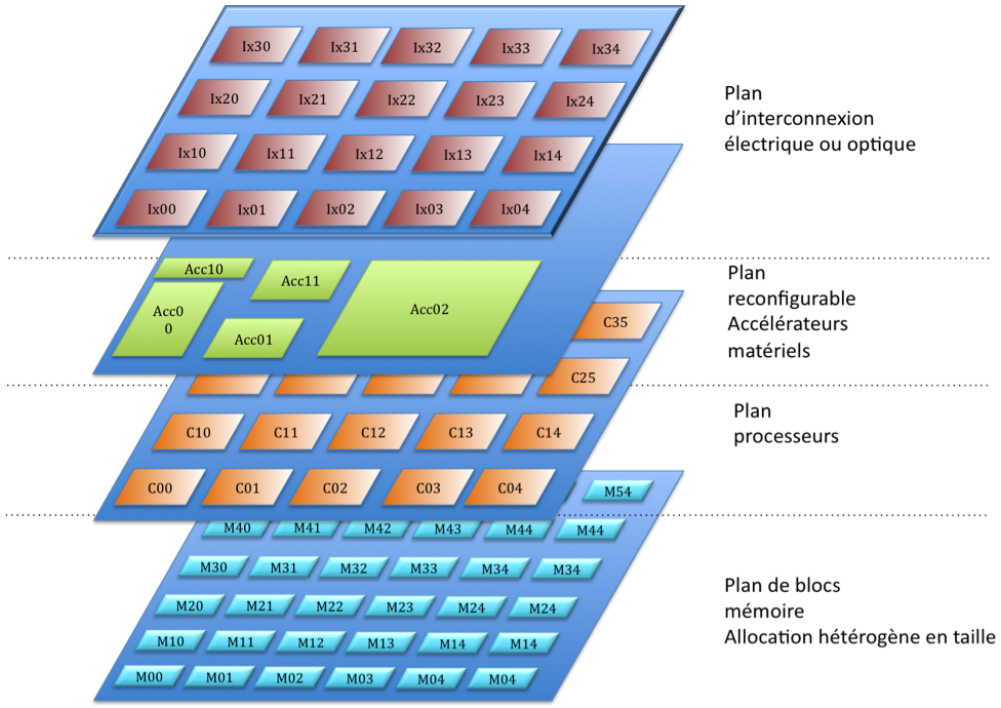


FIGURE 5.16 – Schéma général d'un circuit de type SOC 3D

le plan de processeurs est basé sur une mémoire partagée, rendant ainsi les temps de communications uniformes au sein de ce plan. En ce qui concerne le placement des tâches communicantes sur la zone reconfigurable, il demeure important qu'elles soient géographiquement proches. Ainsi, nous souhaitons enrichir notre ordonnanceur spatial par réseaux de neurones afin qu'il favorise un placement proche des tâches communicantes. Ensuite, nous souhaiterions l'étendre afin qu'il gère la dimension supplémentaire qu'est le plan de processeurs.

Glossary

API *Application Programming Interface*. 23

Bin Packing Problème NP Complet. 7, 24

BiRF *Bitstream Reconfiguration Filter*. 19

bitstream Données de configuration d'une implémentation matérielle. 14, 15, 17–20, 30, 31, 77–79, 98, 147

BRam *Block RAM*. 15, 78, 79, 99

Bus Macro Bus de communication entre PRR. 15, 18

CLB *Configurable Logic Block*. 12, 15, 78, 79, 99

collage Dysfonctionnement d'un transistor empêchant tout changement d'état. 126, 128

Colonne Algorithme de placement. 101, 106–108, 131, 150

compteur ordinal Registre contenant l'adresse de l'instruction en cours d'éxecution. 16

concepteur Concepteur d'une application implémentée matériellement. 17

DDPT Date de Démarage au Plus Tôt. 22, 23, 25, 147

design Implémentation matérielle d'une application. 17

DFPT Date de Fin au Plus Tard. 22, 23, 25, 54, 147

DRA *Dynamic Reconfigurable Accelerator*. 73

DSP *Digital Signal Processing*. 5, 7, 16, 78, 79, 99, 147

EDF *Earliest Deadline First*. 23, 24

FIFO *First In First Out*. 21, 147

FPGA *Field Programmable Gate Array*. 5–9, 12–20, 78, 131, 132, 138, 139, 147, 155

GAL *Generic Array Logic*. 12

GPP *General Purpose Processor*. 72, 73

heaviside Fonction de seuillage. 35

ICAP *Internal Access Configuration Port*. 15, 25

instance Implémentation matérielle d'une tâche. 31

IP *Intellectual Property*. 72, 73

LATTICE Un fabricant de FPGA. 12

loi de Hebb Concept dans le domaine des réseaux de neurones. 38, 39

LUT *Lookup Table*. 5, 7, 12, 147

MER *Maximum Empty Rectangle*. 26–31, 147

migration Opération visant à déplacer une tâche d'une ressource à une autre lors d'une préemption. 24

MMU *Memory Management Unit*. 16

NP Complet Classe de complexité d'un problème. 7, 137

ordonnanceur Composant choisissant les programmes à exécuter. 20–23

PAL *Programmable Array Logic*. 11, 12

PARBIT Outil de relocation. 18

période . 22, 23, 54, 67, 147

Pfair Algorithme d'ordonnancement temps réel multiprocesseur. 2, 9, 24, 53, 65, 67–69, 75, 131, 149

Power PC Processeur généraliste RISC d'IBM. 16

préemption Mécanisme visant à sauvergarder/restaurer le contexte d'un programme. 16, 24, 138

PRM *Partially Reconfigurable Module*. 15, 17, 147

processus Programme en cours d'exécution. 20

PRR *Partially Reconfigurable Region*. 14, 15, 17, 18, 137, 147

RANN *Reconfigurable Artificial Neural Network*. 132, 151

read-back Lecture des données de configuration d'un FPGA. 17

relocation Opération visant à reloger une tâche. 18, 98

REPLICA Outil de relocation. 18, 19

RMS *Rate Monotonic Scheduling*. 23

Round Robin Algorithme d'ordonnancement. 21, 147

RUP Fit *Run-time Utilization Probability Fit*. 31

scanpath Technique de test de circuit. 17

slot Durée entre deux préemptions. 21, 23, 54–56, 58, 59, 62–65, 67, 71, 73, 74, 105, 107, 117, 118, 120, 122, 148–150

SOC *System On Chip*. 5–7, 111, 131, 132, 135, 151

SUP Fit *Static Utilization Probability Fit*. 31, 131

TF *Total Fragmentation*. 29

tick Instant où le système d'exploitation peut préempter une tâche. 23, 54, 55

Virtex une famille de FPGA *Xilinx*. 13, 15, 16, 19

WCET *Worst Case Execution Time*. 22, 23, 54–57, 62, 63, 65, 67, 69, 73, 117, 120, 148, 149

Xilinx Un fabricant de FPGA http ://www.xilinx.com/. 12, 13, 132

Bibliographie

[1] A.Eiche, D. Chillet, S. Pillement, and O. Sentieys. Task placement for dynamic and partial reconfigurable region. In *Proceedings of the Conference on Design and Architectures for Signal and Image Processing*, DASIP '10, pages 82–88, Edinburgh, UK, october 2010.

[2] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass. Hthreads : A Hardware/Software Co-Designed Multithreaded RTOS Kernel. In *10th IEEE Conference on Emerging Technologies and Factory Automation*, volume 2 of *ETFA 2005*, Italy, 2005.

[3] Dalia Aoun, Anne-Marie Déplanche, and Yvon Trinquet. Pfair scheduling improvement to reduce interprocessor migrations. In Giorgio Buttazzo and Pascale Minet, editors, *16th International Conference on Real-Time and Network Systems*, RTNS 2008, Rennes, France, 2008.

[4] Atmel. AT40K FPGA Data Sheet. Technical report, Atmel Corporation, 1999.

[5] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress : a notion of fairness in resource allocation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 345–354, New York, NY, USA, 1993. ACM.

[6] S.K. Baruah. Fairness in periodic real-time scheduling. In *16th IEEE Real-Time Systems Symposium*, RTSS'95, pages 200–209, Pisa, Italy, december 1995. IEEE.

[7] K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers*, 17(1) :68–83, 2000.

[8] T. Becker, W. Luk, and P.Y.K. Cheung. Enhancing relocatability of partial bitstreams for run-time reconfiguration. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM 2007, pages 35 –44, april 2007.

[9] Imène Benkermi, Daniel Chillet, Sébastien Pillement, and Olivier Sentieys. Hardware Task Scheduling for Heterogeneous SoC Architectures. In *European Signal Processing Conference*, EUSIPCO, Poznan, Pologne, 2007.

[10] G. Brebner. The swappable logic unit : a paradigm for virtual hardware. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 77–86, 1997.

[11] C. Cardeira and Z. Mammeri. Handling precedence constraints with neural network based real-time scheduling algorithms. In *Ninth Euromicro Workshop on Real-Time Systems*, pages 207–214. IEEE, 1997.

[12] C. Cardeira and Z. Mammeri. Preemptive and non-preemptive real-time scheduling based on neural networks. In *The 13th IFAC Workshop on Distributed Computer Control Systems*, pages 67–72, September 1995.

[13] Daniel Chillet, Antoine Eiche, SÃl'bastien Pillement, and Olivier Sentieys. Real-time scheduling on heterogeneous system-on-chip architectures using an optimised artificial neural network. *Journal of Systems Architecture*, 57(4) :340 – 353, 2011.

[14] Daniel Chillet, Sébastien Pillement, and Olivier Sentieys. RANN : A Reconfigurable Artificial Neural Network Model for Task Scheduling on Reconfigurable System-on-Chip. In G. Gogniat, D. Milojevic, A. Morawiec, and A. T. Erdogan, editors, *Algorithm-Architecture Matching for Signal and Image Processing*. Springer Verlag, 2010.

[15] E.G. Coffman Jr, M.R. Garey, and D.S. Johnson. Approximation algorithms for bin packing : A survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1996.

[16] K. Compton and S. Hauck. Reconfigurable computing : a survey of systems and software. *ACM Computing Surveys*, 34(2) :171–210, 2002.

[17] S. Corbetta, M. Morandi, M. Novati, M.D. Santambrogio, D. Sciuto, and P. Spoletini. Internal and external bitstream relocation for partial dynamic reconfiguration. *IEEE Transactions on Very Large Scale Integration Systems*, 17(11) :1650–1654, 2009.

[18] J. Cui, Z. Gu, W. Liu, and Q. Deng. An efficient algorithm for online soft real-time task placement on reconfigurable hardware devices. In *IEEE International Symposium on Object/component/services-oriented Real-time distributed Computing*, ISORC, pages 321–328, 2007.

[19] K. Danne. Memory management to support multitasking on fpga based systems. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, 2004.

[20] Qingxu Deng, Fanxin Kong, Nan Guan, Mingsong Lv, and Wang Yi. On-Line placement of Real-Time tasks on 2D partially Run-Time reconfigurable FPGAs. In *Fifth IEEE International Symposium on Embedded Computing*, SEC '08, pages 20–25, 2008.

[21] M.L. Dertouzos and A.K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12) :1497–1506, 1989.

[22] R.P. Dick, D.L. Rhodes, and W. Wolf. Tgff : task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101. IEEE Computer Society, 1998.

[23] Antoine Eiche, Daniel Chillet, Sébastien Pillement, and Olivier Sentieys. Parallel Evaluation of Hopfield Neural Networks. In *NCTA, International Conference on Neural Computation Theory and Applications*, Paris, France, 2011.

[24] S. Fekete, E. Kohler, and J. Teich. Optimal FPGA module placement with temporal precedence constraints. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe conference*, pages 658–667. IEEE Press Piscataway, NJ, USA, 2001.

[25] F. Ferrandi, M. Morandi, M. Novati, M.D. Santambrogio, and D. Sciuto. Dynamic reconfiguration : Core relocation via partial bitstreams filtering with minimal overhead. In *International Symposium on System-on-Chip*, pages 1–4. IEEE, 2006.

[26] Samuel Garcia and Bertrand Granado. OLLAF : a fine grained dynamically reconfigurable architecture for OS support. *EURASIP Journal Embedded System*, January 2009.

[27] Samuel Garcia, Jean-Christophe Prévotet, and Bertrand Granado. Hardware task context management for fine grained dynamically reconfigurable architecture. In *DASIP'07*, page 1, Grenoble, France, November 2007.

[28] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann. Design of homogeneous communication infrastructures for partially reconfigurable fpgas. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'07)*, 2007.

[29] M. Handa. *Online Placement and Scheduling Algorithms and Methodologies for Reconfigurable Computing Systems*. PhD thesis, University of Cincinnati, 2004.

[30] M. Handa and R. Vemuri. Area fragmentation in reconfigurable operating systems. In *Engineering of Reconfigurable Systems and Algorithms*, volume 83. Citeseer, 2004.

[31] Reiner Hartenstein, Michael Herz, and Frank Gilbert. Designing for xilinx xc6200 fpgas. In Reiner Hartenstein and Andres Keevallik, editors, *Field-Programmable Logic and Applications From FPGAs to Computing Paradigm*, volume 1482 of *Lecture Notes in Computer Science*, pages 29–38. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0055230.

[32] S. Haykin. *Neural Networks : A Comprehensive Foundation (2nd Edition)*. Prentice Hall, 1998).

[33] DO Hebb. The organization of behavior ; a neuropsychological theory. 1949.

[34] JJ Hopfield. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *Proceedings of the National Academy of Sciences*, 79(8) :2554–2558, 1982.

[35] JJ Hopfield and DW Tank. "Neural" computation of decisions in optimization problems. *Biological cybernetics*, 52(3) :141–152, 1985.

[36] E. Horta and J.W. Lockwood. PARBIT : a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs). *Dept. Comput. Sci., Washington Univ., Saint Louis, MO, Tech. Rep. WUCS-01-13*, 2001.

[37] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert. Replica : A bitstream manipulation filter for module relocation in partial reconfigurable systems. In *19th IEEE*

International Parallel and Distributed Processing Symposium, pages 151b–151b. IEEE, 2005.

[38] H. Kalte and M. Porrmann. REPLICA2Pro : Task relocation by bitstream manipulation in Virtex-II/Pro FPGAs. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 403–412. ACM, 2006.

[39] H. Kalte, M. Porrmann, and U. Ruckert. System-on-programmable-chip approach enabling online fine-grained 1d-placement. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 141. IEEE, 2004.

[40] N. Kamiura, T. Isokawa, and N. Matsui. On improvement in fault tolerance of hopfield neural networks. In *Test Symposium, 2004. 13th Asian*, pages 406 – 411, november 2004.

[41] Y. Kamp and M. Hasler. *Recursive neural networks for associative memory*. John Wiley & Sons, Inc., 1990.

[42] D. Koch, A. Ahmadinia, C. Bobda, and H. Kalte. Fpga architecture extensions for preemptive multitasking and hardware defragmentation. In *IEEE International Conference on Field-Programmable Technology*, pages 433 – 436, december 2004.

[43] M. Koester, W. Luk, J. Hagemeyer, M. Porrmann, and U. Ruckert. Design optimizations for tiled partially reconfigurable systems. *IEEE Transactions on Very Large Scale Integration Systems*, (99) :1–14, 2010.

[44] M. Koester, M. Porrmann, and H. Kalte. Task placement for heterogeneous reconfigurable architectures. In *IEEE International Conference on Field-Programmable Technology*, pages 43–50, 2005.

[45] M. Koester, M. Porrmann, and H. Kalte. Relocation and defragmentation for heterogeneous reconfigurable systems. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 70–76. Citeseer, 2006.

[46] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm : Exact characterization and average case behavior. In *Proceedings Real Time Systems Symposium*, pages 166–171. IEEE, 1989.

[47] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61, 1973.

[48] A. M. Lyapunov. *General Problem of the Stability Of Motion*. CRC Press, 1 edition, 1992.

[49] T. Marconi, Y. Lu, K. Bertels, and G. Gaydadjiev. A Novel Fast Online Placement Algorithm on 2D Partially Reconfigurable Devices. In *International Conference on Field-Programmable Technology*, FPT 2009, pages 296–299. Citeseer, 2009.

[50] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4) :115–133, 1943.

[51] A. Pasturel, A. Eiche, D. Chillet, S. Pillement, and O. Sentieys. Implémentation matérielle d'un réseau de neurones pour l'ordonnancement temporel de tâches sur architectures multi-processeur hétérogènes. In *Symposium en Architecture de machines (SympA)*, Saint-Malo, France, May 2011.

[52] Sébastien Pillement, Olivier Sentieys, and Raphaël David. Dart : a functional-level reconfigurable architecture for high energy efficiency. *EURASIP Journal Embedded System*, 2008 :5 :1–5 :13, January 2008.

[53] Norman Pullman. Clique coverings of graphs - a survey. In Louis Casse, editor, *Combinatorial Mathematics X*, volume 1036 of *Lecture Notes in Mathematics*, pages 72–85. Springer Berlin / Heidelberg, 1983.

[54] F. Rosenblatt. The perceptron : A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6) :386, 1958.

[55] C. Rossmeissl, A. Sreeramareddy, and A. Akoglu. Partial bitstream 2-d core relocation for reconfigurable architectures. In *NASA/ESA Conference on Adaptive Hardware and Systems*, AHS 2009, pages 98 –105, 29 2009-aug. 1 2009.

[56] K. Rupnow, Wenyin Fu, and K. Compton. Block, drop or roll(back) : Alternative preemption methods for rh multi-tasking. In *17th IEEE Symposium on Field Programmable Custom Computing Machines*, FCCM '09, pages 63 –70, april 2009.

[57] Klaus Schild and Jörg Würtz. Off-line scheduling of a real-time system. In *Proceedings of the 1998 ACM symposium on Applied Computing*, SAC '98, pages 29–38, New York, NY, USA, 1998. ACM.

[58] J. Septien, H. Mecha, D. Mozos, and J. Tabero. 2d defragmentation heuristics for hardware multitasking on reconfigurable devices. *International Parallel and Distributed Processing Symposium*, 0 :175, 2006.

[59] J.B. Sidney. Optimal single-machine scheduling with earliness and tardiness penalties. *Operations Research*, 25(1) :62–69, 1977.

[60] H.T. Siegelmann and E.D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6) :77–80, 1991.

[61] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms : Online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11) :1393–1407, 2004.

[62] J. Tabero, J. Septien, H. Mecha, and D. Mozos. A Low Fragmentation Heuristic for Task Placement in 2D RTR HW Management. In *14th International Conference Field-programmable logic and applications*, FPL 2004, pages 241–250, Antwerp, Belgium, August 2004. Springer-Verlag New York Inc, Springer.

[63] GA Tagliarini, JF Christ, and EW Page. Optimization using neural networks. *IEEE Transactions on Computers*, 40(12) :1347–1358, 1991.

[64] A. Tanenbaum, J.A. Hernandez, R. Joly, and M. Dupuy. *Systèmes d'exploitation*, volume 2. Pearson Education, 2003.

[65] H. Walder, C. Steiger, and M. Platzner. Fast online task placement on FPGAs : free space partitioning and 2D-hashing. In *Proceedings of the International Parallel and Distributed Processing Symposium*, page 8, 2003.

[66] Xilinx. Early Access Partial Reconfiguration User Guide (UG208 v1.1), 2006.

[67] Xilinx. Virtex-4 FPGA Configuration User Guide (UG071 v1.11). `http://www.xilinx.com/support/documentation/user_guides/ug071.pdf`, 2009.

[68] Xilinx. Virtex-5 FPGA Configuration User Guide (UG191 v3.10). `http://www.xilinx.com/support/documentation/user_guides/ug191.pdf`, 2011.

[69] Jia Xu and David Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3) :360–369, March 1990.

[70] Zhi-Hua Zhou and Shi-Fu Chen. Evolving fault-tolerant neural networks. *Neural Computing and Applications*, 11 :156–160, 2003. 10.1007/s00521-003-0353-4.

Table des figures

1 Exemple de système sur puce composé de cinq ressources différentes ainsi que d'une infrastructure de communication. 8

2 Exemple de placement de deux tâches sur une architecture reconfigurable hétérogène de type FPGA. La tâche τ_1 requiert quatre LUT et deux DSP, la tâche τ_2 six LUT. 10

1.1 Schéma et table de vérité de la fonction logique $(\bar{A} \cdot B) + (C \oplus D)$. . . . 14

1.2 FPGA reconfigurable dynamiquement par colonne. Un module M_1 est exécuté sur la première colonne. 16

1.3 Reconfiguration par région. La sous figure (a) présente un FPGA composé de cinq régions. La sous figure (b) illustre l'exécution de deux modules sur ce FPGA. 16

1.4 Illustration de la reconfiguration par région. Deux PRR ont été instanciés sur la zone reconfigurable. Trois PRM différents peuvent être accueillis par le PRR_1 et deux PRM par le PRR_2. 17

1.5 Modèle d'exécution permettant la relocation de *bitstream* tel que présenté dans [55]. Trois modules sont en cours d'exécution et chacun de ces modules est pourvu d'une interface de communication. La région statique contient le contrôleur de reconfiguration (incluant les mécanismes de manipulation du *bitstream*), ainsi que le contrôleur de bus. 22

1.6 Exemple d'ordonnancement de trois tâches sur une architecture supportant la préemption et sur une qui ne la supporte pas. La sous figure (a) présente un ordonnancement FIFO. La sous figure (b) présente un ordonnancement *Round Robin*. 24

1.7 Exemple d'exécution d'une tâche temps réel τ_i ayant une DDPT $DDPT_i = 0$, une période $P_i = 5$, une DFPT $D_i = 4$, et un WCET $C_i = 3$. 26

1.8 FPGA reconfigurable dynamiquement par colonne. Trois tâches sont en cours d'exécution sur le FPGA. L'espace libre restant est la colonne C_2. 29

1.9 Un exemple de placement et les MER M_1 et M_2 associés à ce placement. La zone barrée verticalement correspond au MER M_1, celle barrée horizontalement au MER M_2. 30

1.10 Exemple de placement identique à la figure 1.9 où seuls des MER disjoints sont considérés [7]. 30

1.11 Représentation de l'état de la zone reconfigurable et d'une liste de sommets décrivant l'espace libre. 33

2.1 Photographie microscopique de neurones (agrandie 650x). 36
2.2 Illustration du fonctionnement d'un neurone formel à n entrées. 37
2.3 Illustration d'un perceptron à deux entrées. 39
2.4 Exemple d'un réseau de neurones de Hopfield possédant trois neurones. 39
2.5 Exemple de motifs soumis au réseau. Les pixels de chaque motif sont numérotés de un à neuf et les motifs sont numérotés de un à trois. . . . 41
2.6 Un motif partiel est soumis au réseau. Le réseau répond le motif correspondant. 41
2.7 Transformation d'un problème d'optimisation en un réseau de neurones de Hopfield. Nous supposons que le problème est codé sous une forme exploitable par un réseau de neurones. 47
2.8 Exemple de l'application d'une règle 2-de-3. Les connexions sont paramétrées à -2 et les valeurs de seuil à $2k-1=3$ avec $k=2$. 50
2.9 Exemple de l'application d'une règle au-plus-2-de-3. Les connexions sont paramétrées à -2 et les valeurs de seuils à $2k-1=3$ avec $k=2$. Les neurones x_1 x_2 et x_3 sont des neurones utilisés par le codage du problème contrairement aux neurones cachés x_4 et x_5 qui ne sont utilisés que par la règle au-plus-2-de-3 50
2.10 Exemple d'une règle de k-consécutivité de 2 neurones parmi 7. Seules les connexions des neurones x_3 et x_4 sont représentées. 52

3.1 (a) La représentation initiale d'un problème d'ordonnancement de trois tâches sur une architecture disposant d'une ressource d'exécution. Chaque ligne correspond à une tâche τ_i ayant un WCET C_i. Les colonnes correspondent à des *slots* d'exécution. (b) Un exemple de scénario d'exécution valide. 56
3.2 Application des règles nécessaires à la construction du réseau. 57
3.3 Exemple d'un problème d'ordonnancement (T tâches et R ressources) modélisé par un réseau de neurones utilisant des neurones inhibiteurs (placés dans les zones grisées). 59
3.4 Réseau de neurones comportant deux tâches décrites chacune par cinq neurones ainsi que deux neurones inhibiteurs $xh_{i,j}$ et $xh_{i,l}$. Le neurone $xh_{i,j}$ est activé par les neurones $x_{i,j,k}$ pour tout $k \in 1,\ldots,5$ et inhibe les neurones $x_{i,l,k}$ pour tout $k \in 1,\ldots,5$. 61
3.5 Exemple d'un problème d'ordonnancement (T tâches et R ressources) modélisé par un réseau de neurones utilisant des neurones inhibiteurs (placés dans les zones grisées). 62

3.6 Évolution de l'énergie d'un réseau disposant de neurones inhibiteurs. L'énergie du réseau n'est pas strictement décroissante au fil des évaluations de neurones. La première partie grisée - $[1, 8]$ - correspond à la phase d'activation des neurones. La zone blanche correspond à l'activation d'un neurone inhibiteur. La seconde partie - $[9, 30]$ - grisée correspond à la convergence du réseau suite à l'activation des neurones inhibiteurs. . . . 63
3.7 Exemple présentant un scénario d'évaluation d'un réseau de neurones modélisant l'ordonnancement de cinq tâches sur une architecture disposant de trois unités d'exécution. À chaque *slot*, le nombre de neurones activés est égal au nombre d'unités d'exécution. Les tâches fictives τ_{f_1}, τ_{f_2} et τ_{f_3} permettent de modéliser l'inactivité des unités d'exécution. . . 65
3.8 Exemple présentant un scénario d'évaluation d'un réseau de neurones modélisant l'ordonnancement de cinq tâches sur une architecture disposant de trois unités d'exécution. Les neurones noirs sont des neurones actifs, les neurones blancs sont inactifs et les gris sont des neurones inactifs dont nous analysons l'évaluation. 67
3.9 Exemple de réseaux modélisant l'ordonnancement de trois tâches (τ_1, τ_2 et τ_3) sur une architecture disposant d'une seule unité d'exécution. Les WCET des tâches τ_1, τ_2 et τ_3 sont respectivement égaux à 3, 2 et 2. (a) présente le réseau issu de la modélisation classique dans un état initial aléatoire. (b) présente ce même réseau après son évaluation. (c) présente le réseau neuronal issu de notre modélisation dans un état initial aléatoire. Sa convergence est semblable à (b). 68
3.10 Exemple d'ordonnancement de trois tâches sur une architecture disposant de deux unités d'exécution. (a) est le réseau issu de la modélisation classique dans un état quelconque. (b) est le réseau dépourvu de neurones cachés. 68
3.11 Diagramme de Gantt de l'exécution des huit tâches décrites par l'équation (3.15) sur une architecture disposant de quatre unités d'exécution. (a) Solution obtenue par l'algorithme Pfair. (b) Solution obtenue par notre algorithme. 70
3.12 Scénario illustré par la figure 3.11.b après une passe de compactage. . . 70
3.13 Exemple d'une solution générée par notre réseau de neurones. Les sept tâches décrites par la table 3.2 sont ordonnancées sur une architecture disposant de deux unités d'exécution. La période d'ordonnancement est égale à 20 *slots*. Les zones grisées contiennent les neurones inhibiteurs. . 73
3.14 Résultats de l'ordonnancement d'une application composée de 10 tâches sur un composé de cinq unités d'exécution. 76

4.1 Floorplan d'un Virtex 5 80
4.2 Modélisation d'une architecture reconfigurable. La surface modélisée possède trois types de ressources. Ces ressources sont regroupées dans trois types de régions. La surface est composée de 25 régions. 81
4.3 Instances de la tâche 1 82

4.4 Exemple de ressources utilisées et de position des instances de deux tâches. 83
4.5 Réseau de neurone associé aux tâches décrites par la figure 4.4 83
4.6 Application des rêgles 1-de-n au réseau associé aux tâches décrites par la figure 4.4 84
4.7 Illustration des connexions permettant de contrôler le chevauchement des instances des tâches décrites par la figure 4.4. 85
4.8 Réseau de neurones de Hopfield associé aux tâches décrites par la figure 4.4. 88
4.9 Exemple d'un jeu de tâches 89
4.10 Exemple d'une fonction s. La fonction s est une fonction décroissante par échelon puis constante en zéro. 90
4.11 Simulation de différentes fonction s. L'abscisse décrit les échelons utilisés tandis que l'ordonnée présente le pourcentage de tâches placées en fonction des échelons utilisés. 92
4.12 Moyennes du nombre de tâches placées. Chaque jeu contient neuf tâches, et chaque tâche possède cinq instances. 95
4.13 Comparaison des taux de succès de l'algorithme *SUP Fit* et de notre algorithme. 96
4.14 Nombre d'évaluations de neurones en fonction du nombre de tâches et d'instances. 97
4.15 Évolution du nombre de *slices* de la famille Virtex de Xilinx. Chaque barre indique le nombre minimal et maximal de *slices* de l'ensemble des modèles au sein d'une famille. 101
4.16 Illustration de la présence de symétries dans la disposition des ressources d'une architecture reconfigurable hétérogène (Xilinx Virtex 5 SX50t). . . 102
4.17 Description de toutes les positions verticales de l'instance $\tau_{1,1}$. 103
4.18 Illustration de deux instances de la tâche τ_1. 103
4.19 Illustration des étapes de l'algorithme de placement de tâches. 104
4.20 Détermination des positions en utilisant l'heuristique issue de l'algorithme *quad corner*. (a) L'état de la zone reconfigurable. Les régions hachurées sont occupées par d'autres tâches. (b) Position obtenue pour l'instance $\tau_{1,1}$. (c) Position de l'instance $\tau_{1,2}$ 106
4.21 Exemple d'un scénario d'exécution de tâches. 108
4.22 Comparaison du pourcentage de tâches placées par l'algorithme *SUP Fit* 1,*SUP Fit* * et Colonne. Dans ces simulations, nous considérons des scénarios comportant 20 tâches et 40 *slots*. Les tâches disposent d'un nombre variable d'instances indiquées en abscisse. 109
4.23 Comparaison du nombre d'instances utilisées par l'algorithme *SUP Fit* * et Colonne lors des simulations présentées par la figure 4.22. 110

5.1 Exemple d'une matrice de connexion. Les éléments diagonaux ont des valeurs égales à zéro tandis que les autres elements w_{ij} sont négatifs ou nuls. 119
5.2 Représentation graphique des règles k-de-n appliquées aux réseaux. . . . 120

5.3 Exemple d'un réseau de neurones composé de cinq neurones. Les neurones x_1 et x_2 ne sont pas connectés entre eux. 121
5.4 Graphe complémentaire à celui décrit par la figure 5.3. Seuls les neurones x_1 et x_2 sont connectés. 121
5.5 Représentation des paquets de neurones. Les neurones ayant la même couleur peuvent être évalués simultanément. Des neurones de couleur différentes sont évalués séquentiellement. 122
5.6 Nombre de neurones par rapport à la taille des paquets. Comme la taille des paquets est égale au nombre de tâches, l'abcisse représente le nombre de tâches et la taille des paquets. 123
5.7 Nombre d'évaluations de paquets dans les cas séquentiel et parallèle. . . 124
5.8 Accélération par rapport au nombre de tâches. 125
5.9 Nombre de défaillances supportées par une règle k-de-n pour $N = 100$ et k variant de 0 à 100. 131
5.10 Impact sur le temps de convergence en cas de défaillances au sein du réseau de neurones. 132
5.11 Performances temporelles du réseau de neurones RANN sur un circuit Virtex 5 XC5VFX70TF. 135
5.12 Surfaces occupées par le réseau de neurones RANN sur un circuit Virtex 5 XC5VFX70TF. 136
5.13 Exemple d'un graphe de tâche composé de trois tâches. Les tâches τ_2 et τ_3 dépendent du résultats de la tâche τ_1. 136
5.14 Scénario d'exécution de trois tâches périodiques. 137
5.15 Les sous-figures (a), (b) et (c) décrivent le placement successif des tâches τ_1, τ_2 et τ_3. La sous-figure (d) représente le placement optimal en termes de nombre de rectangles vides. 137
5.16 Schéma général d'un circuit de type SOC 3D 138

Résumé

L'évolution constante des applications, que ce soit en complexité ou en besoin de performances, impose le développement de nouvelles architectures. Parmi l'ensemble des architectures proposées se démarquent les architectures reconfigurables. Ce type d'architectures offre des performances proches d'un circuit dédié tout en proposant davantage de flexibilité. Si originellement ces architectures ne pouvaient être configurées qu'à leur démarrage, elles sont désormais configurables partiellement à tout moment. Cette fonctionnalité, nommée « reconfiguration dynamique », permet de multiplexer temporellement et spatialement l'exécution de plusieurs applications, rapprochant ainsi l'utilisation des architectures reconfigurables de celle d'un processeur généraliste. À l'instar des ordinateurs pourvus d'un processeur généraliste, il s'avère donc nécessaire de disposer d'un système d'exploitation pour architectures reconfigurables. Cette thèse se focalise sur la création d'ordonnanceurs destinés à ce type d'architectures. Parce que nous ciblons l'exécution d'applications complexes - composées de plusieurs tâches dont l'ordre d'exécution n'est pas connu à l'avance - les algorithmes d'ordonnancement doivent être exécutés en ligne et les solutions obtenues en des temps très brefs. À cette fin, nous réalisons nos algorithmes d'ordonnancement en nous basant sur des réseaux de neurones de Hopfield. Ce type de réseaux a été utilisé pour résoudre des problèmes d'optimisations (tels que des problèmes d'ordonnancement) où il a été constaté qu'ils produisaient des solutions rapidement. De plus, leur simplicité de fonctionnement (peu de structures de contrôle) permet de les implémenter efficacement sur des architectures reconfigurables de type FPGA.

Le premier chapitre de ce document introduit les concepts sur lesquels sont basés nos travaux, à savoir, les architectures reconfigurables ainsi que l'ordonnancement temporel et spatial destiné à ce type d'architectures. Le second chapitre présente les réseaux de neurones de Hopfield. Dans le troisième chapitre, nous présentons un ordonnanceur temporel pour architectures hétérogènes reconfigurables.

Le quatrième chapitre présente deux ordonnanceurs spatiaux, fondés sur des hypothèses matérielles différentes. Le premier, basé sur un réseau de Hopfield, ne requiert que des mécanismes de reconfiguration simples et est donc utilisable avec les outils fournis par les fabricants. Le second, quant à lui, nécessite des mécanismes actuellement non fournis par les fabricants, mais permet d'exploiter plus finement l'architecture. Le dernier chapitre présente des travaux relatifs aux réseaux de neurones de Hopfield, à savoir l'évaluation parallèle de neurones, ainsi que des propriétés de tolérance aux fautes de ces réseaux.