République tunisienne
Ministère de l'Enseignement Supérieur

Université de Sfax
Ecole Nationale d'Ingénieurs de Sfax
Département de Génie Electrique

Cycle de Formation Doctorale dans
la Discipline *Génie Electrique*
Automatique-Informatique Industrielle

Mémoire de *DEA*
N° d'ordre: A-II 01

MEMOIRE

présenté à

l'Ecole Nationale d'Ingénieurs de Sfax
(Département de Génie Electrique)

en vue de l'obtention du

Diplôme d'Etudes Approfondies

dans la discipline *Génie Electrique*

Automatique-Informatique Industrielle

Option: *Systèmes Informatiques et Automatismes*

par

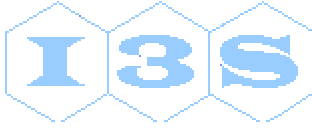
Fakhreddine GHAFFARI
Ingénieur – *Génie Electrique*
Option: Micro-Informatique

Encadré par Michel Auguin et Maher Ben Jemaa

Etude du partitionnement logiciel/matériel d'applications à distribution variable de charge de calcul

soutenu le 29 juin 2002, devant la commission d'examen:

M.	**Salem NASRI**	*Président*
M.	**Maher BEN JEMAA**	*Membre*
M.	**Mohamed ABID**	*Membre*



Laboratoire Informatique Signaux
et Système de Sophia-Antipolis

Ecole Nationale d'Ingénieurs de
Sfax

Centre National de la recherche
Scientifique

MEMOIRE

présenté à

l'Ecole Nationale d'Ingénieurs de Sfax

(Département de Génie Electrique)

en vue de l'obtention du

Diplôme d'Etudes Approfondies

dans la discipline *Génie Electrique*

Automatique - Informatique Industrielle

Option: *Systèmes Informatiques et Automatismes*

par

Fakhreddine GHAFFARI

Ingénieur – *Génie Electrique*
Option: Micro-Informatique

Encadré par Michel Auguin et Maher Ben Jemaa

Etude du partitionnement logiciel/matériel d'applications à distribution variable de charge de calcul

soutenu le 29 juin 2002, devant la commission d'examen:

M.	**Salem NASRI**	*Président*
M.	**Maher BEN JEMAA**	*Membre*
M.	**Mohamed ABID**	*Membre*

AVANT PROPOS

Le travail présenté dans ce mémoire a été effectué au laboratoire "Informatique Signaux et Systèmes de Sophia-Antipolis" *(I3S)* dans le cadre d'une étroite collaboration entre l'Ecole Nationale d'Ingénieurs de Sfax, et le laboratoire I3S. Il a été élaboré au sein du projet "Modélisation et Synthèse d'Architectures de Traitement de Signal" *(MOSARTS)*. Ce projet a pour objectif de réduire significativement le 'fossé de conception' entre les outils et les besoins qui peut être un facteur pénalisant vis à vis de l'innovation en matière de services de télécommunication.

Mes vifs remerciements vont à Mr Maher Ben Jemaa (ENIS) et Mr Michel AUGUIN (I3S) qui ont bien voulu diriger, avec beaucoup de bienveillance, mes travaux. Je tiens à leurs exprimer ma profonde reconnaissance, pour la grande sollicitude avec laquelle ils ont dirigé ce travail et pour les discussions nombreuses et fructueuses qu'ils ont toujours accueillies avec une grande disponibilité.
Ma reconnaissance envers eux est immense pour les conseils et les remarques constructives qu'ils m'ont fournis durant toute la période du projet.

Ma gratitude, mon profond respect et mes remerciements s'adressent à tous les membres de jury pour leur attention consacrée pour examiner mon présent mémoire et d'accepter de juger mon travail.

Fakhreddine

Table des Matières

Introduction générale 1

Chapitre 1 : Etat de l'art du partitionnement logiciel/matériel 6

I.1 Introduction ………………………………………………………………… 7
I.2 Partitionnement logiciel/matériel dans le cadre du CODESIGN…………… 8
I.3 Approche de partitionnement……………………………………………….. 10
I.3.1 Architecture cible ………………………………………………. 11
I.3.2 Méthode de partitionnement……………………………………. 12
I.3.3 Les critères d'optimisation ……………………………………… 14
I.4 Limites des approches classiques ………………………………………….. 14
I.5 Conclusion ………………………………………………………………. 15

Chapitre 2 : Applications de traitement d'images et choix d'architecture correspondante 16

II.1 Introduction …………………………………………………………….. 17
II.2 Modélisation des applications de traitement d'images ……………………. 18
II.3 Détection du mouvement sur un fond d'image fixe……………………….. 18
II.3.1 Schéma fonctionnel ……………………………………………. 19
II.3.2 Principe de fonctionnement ……………………………………. 19
II.3.3 Organigramme du processus de détection du mouvement …….. 20
II.3.4 Algorithme de la détection du mouvement …………………….. 21
II.4 Architecture cible : hétérogène (processeur/FPGA) …….………………. 27
II.5 Conclusion ………………………………………………………………. 30

Chapitre 3 : Nouvelle approche de partitionnement pour les applications à distribution variable de charge de calcul 31

III.1 Introduction ……………………………………………………………… 32
III.2 Profiling de l'application considérée ………………………………….. 32
III.3 Traitement des résultats du Profiling ……………………………………….. 50

III.4 Graphe de flots de données conditionné ………………………………………. 53
III.5 Exploration automatique de l'ensemble des configurations …………………. 56
III.5.1 Structure du programme ………………………………………… 57
III.5.2 Données d'entrée ………………………………………………… 58
III.5.3 Recherche des configurations réalisables ………………………… 59
III.5.4 Exemple d'application : la détection de mouvement sur un fond d'image fixe ………………………………………………… 59
III.6 Idée de réalisation ……………………………………………………………… 59
III.7 Conclusion ……………………………………………………………………… 60

Chapitre 4 : Technique de partitionnement basée sur un Algorithme génétique 62

IV.1 Introduction …………………………………………………………………… 63
IV.2 Modèle d'application retenu …………………………………………………… 63
IV.3 Présentation des algorithmes génétiques …………………………………… 65
IV.3.1 Principes généraux ……………………………………………… 65
IV.3.2 Description détaillée …………………………………………….. 67
IV.4 Application de l'algorithme génétique au problème de partitionnement …… 70
IV.4.1 Le codage choisi ………………………………………………… 70
IV.4.2 L'évaluation ……………………………………………………… 72
IV.4.2.1 L'algorithme de Clustering …………………………….. 72
IV.4.2.2 L'évaluation d'un individu …………………………….. 74
IV.4.2.3 La fonction coût ……………………………………….. 75
IV.4.3 Paramètres de réglage de l'algorithme génétique ……………….. 76
IV.5 Conclusion …………………………………………………………………….. 77

Chapitre 5 : Résultats et analyses 78

V.1 Introduction ……………………………………………………………………. 79
V.2 Résultats de l'outil du partitionnement ……………………………………… 79
V.3 Temps d'exécution total ……………………………………………………….. 81
V.4 Temps de communication …………………………………………………… 82
V.5 Utilisation des ressources matérielles ………………………………………. 83

V.6 Travaux futurs ………………………………………………………………. 83
V.7 Conclusion ………………………………………………………………… 84

Conclusion générale 85

Références 88

Annexe 1 91

Table des Figures

1 Organigramme de dépendance du projet EPICURE 4

1.1 La stratégie de conception d'un système 7

1.2 Processus d'exploration de l'espace de conception dans le CODESIGN 9

1.3 Le partitionnement au niveau système 10

2.1 Graphe de flots de données (DFG) 18

2.2 Architecture d'une caméra intelligente 19

2.3 Principe de la détection du mouvement 20

2.4 Organigramme du processus de détection de mouvement 20

2.5 Structure du programme 21

2.6 Schéma de principe du moyennage 22

2.7 Schéma de principe de la soustraction et de la valeur absolue.................... 23

2.8 Schéma du principe du seuillage 24

2.9 Masque de convolution 24

2.10 Schéma de principe des traitements morphologiques 25

2.11 Coordonnées d'un objet dans une image 26

2.12 Principe de mise à jour de l'image de référence.................................... 26

2.13 Capacité des circuits FPGA et taille de leur mémoire de configuration............ 28

2.14 Processus de configuration d'un circuit 28

2.15 Système reconfigurable à interface externe 29

2.16 Système EXCALIBUR d'ALTERA... 30

3.1 Images test de l'érosion 42

3.2 Principe de prédiction de branchement à 4 états d'un processeur 43

3.3 Traitement de la fonction : construction de l'enveloppe englobante................. 51

3.4 Traitement de la fonction : Etiquetage 52

3.5 Traitement de la fonction : Ouverture par reconstruction 52

3.6 DFG avec tâches à temps d'exécution variable 54

3.7 DFG non conditionné de la combinaison 13 55

3.8 Temps d'exécution en fonction d'un paramètre de corrélation 56

3.9 Organigramme du processus d'exploration 57

3.10 Idée de réalisation de l'approche de partitionnement 67

4.1 Transit des données par la mémoire d'interface …………………………………….. 64
4.2 Principe général des algorithmes génétiques ……………………………………….. 66
4.3 Slicing Crossover ………………………………………………………………… 68
4.4 Principe de l'opération de mutation ………………………………………………. 69
4.5 Exemple de DFG avec les implantations des tâches ………………………………. 71
4.6 Exemple d'ordonnancement d'une solution sur l'architecture ……………………. 71
4.7 Exemple d'affectation des priorités dans un Clustering d'un DFG………………… 73
4.8 Définition des contextes d'un DFG ……………………………………………….. 74
5.1 Résultats du partitionnement de configuration 3 ………………………………… 79
5.2 Résultats du partitionnement de configuration 6 ………………………………… 80
5.3 Temps d'exécution total de toutes les configurations ……………………………. 81
5.4 Principe du temps libre pendant l'exécution de l'application …………………….. 82
5.5 Temps de communication total pour toutes les configurations ………………….. 82
5.6 Nombre total des ressources pour toutes les configurations ……………………… 83

Introduction générale

Introduction

L'analyse de mouvement dans les séquences d'images est un domaine de recherche active à l'heure actuelle en raison de son importance dans de nombreuses applications : télésurveillance, compression pour les télécommunications ou l'archivage, diagnostic médical, météorologie, contrôle non destructif, robotique mobile, etc. Cependant, ces applications requièrent des puissances de calcul adaptées.

Le problème s'accentue lorsque nous voulons travailler sur des systèmes embarqués qui seront soumises à des contraintes de temps réel, de consommation de surface,… A titre d'exemple, les algorithmes de traitement de la vidéo peuvent demander de 0.1 à 10 milliards d'instructions par seconde (GIPS)[2]. Intégrer une telle puissance de traitement dans un système enfoui n'est pas toujours aisé et le problème de la mise en œuvre optimisée de ces applications se pose de façon aiguë.

Aujourd'hui, l'évolution des applications vers des systèmes à la fois plus performants et plus complexes conduit à l'élaboration de dispositifs hétérogènes, c'est à dire constitués d'unités de nature différente (i.e. logicielle, matérielle). Parmi les unités logicielles, nous pouvons distinguer deux grandes classes de circuits numériques : les processeurs d'usage général (GPP) et les processeurs à usage spécifique (ASIP, DSP). Les processeurs d'usage général peuvent être programmés pour exécuter n'importe quelle classe d'application, alors que les processeurs à usage spécifique sont dédiés à une classe d'application (exemple : Traitement d'image, traitement du signal, cryptographie).

Un certains nombre d'applications numériques peuvent être réalisées de façon uniquement logicielle, solution intéressante par la flexibilité qu'elle apporte due à la possibilité de reprogrammation. La motivation principale de l'utilisation de processeurs dédiés (à usage spécifique) réside elle dans le respect des contraintes de performances ou dans la confidentialité de la solution implémentée.

Un autre facteur important dans l'évolution des systèmes modernes est l'apparition de nouvelles architectures exploitant la synergie entre le matériel et le logiciel, basées sur la programmation des circuits matériels tels que les composants FPGAs (Field Programmable Gate Array). Ces composants sont principalement utilisés pour l'accélération de calculs spécifiques ou pour faire du prototypage d'ASIC (Application Specific Integrated Circuit).

Bien que les solutions uniquement logicielles soient préférables, des applications, notamment dans le domaine des télécommunications et du multimédia, impliquent une architecture hétérogène logicielle/ matérielle. Dans ce cas, des nouvelles méthodologies de conception qualifiées de conception conjointe logiciel/ matériel (Codesign) sont nécessaires. L'approche codesign consiste alors à définir l'ensemble des sous tâches d'une application à intégrer et à effectuer leur répartition sur des cibles logicielles ou matérielles. L'intérêt majeur de cette méthodologie réside dans la recherche d'une adéquation application/ architecture satisfaisant les nombreuses contraintes de conception telles que le coût, les performances, la surface, la consommation, les temps de conception et de développement (Time To Market), l'évolutivité,… La conception efficace de ces systèmes hétérogènes nécessite une approche globale dans laquelle les parties matérielles et logicielles sont conçues en parallèle et de façon interactive [3].

Une des phases clés de l'approche Codesign est le partitionnement logiciel/ matériel ; qui consiste à effectuer le choix pour une implémentation logicielle, matérielle ou mixte des différentes parties du système.

Notre travail fait partie des activités du groupe MOSARTS du Laboratoire **I3S** (Informatique Signaux et Systèmes de Sophia-Antipolis), il s'inscrit dans le cadre d'un projet exploratoire soutenu par le ministère Français de la recherche par le biais du Réseau National des Technologies Logicielles (RNTL). Ce projet s'appelle EPICURE, sa durée est de deux ans, les partenaires de L'**I3S** qui participent dans ce travail sont : Le **CEA** (Commissariat à l'Energie Atomique), **THALES**, **ESTEREL TECHNOLOGIES** et le **LESTER** de l'Université de Bretagne Sud. (Voir annexe 1)

Dans le cadre de ce projet il est proposé d'étudier un environnement (méthodologie et outils associés) destiné à faciliter la phase de partitionnement d'une application programmée et réconfigurable.

Cette approche (voir figure 1) abordera plusieurs aspects essentiels lors du développement des applications en particulier :

- ☞ Une plus grande rapidité de conception.
- ☞ Une meilleure adéquation aux spécificités de l'application.
- ☞ Une meilleure évolutivité des applications.
- ☞ Un accroissement de la sûreté de fonctionnement des produits.
- ☞ Une grande pérennité (repousse le mur de l'obsolescence).

Dans le cadre d'EPICURE, le **CEA** et ses partenaires utilisent comme application test une caméra intelligente qui vise le domaine de la vidéo-surveillance pour le contrôle qualité dans l'industrie et pour la sécurité des personnes et des biens.

Le principe de cette caméra consiste à détecter les mouvements sur un fond d'image fixe. Une telle application fait appel à des séquences d'opérations de traitement d'images. Parmi ces opérations nous distinguons celles qui conservent un temps d'exécution fixe d'une image à une autre de celles dont le temps de calcul varie suivant la nature des images.

C'est sur ce dernier type d'applications que nous focalisons notre étude et sur lesquelles il s'agit de tenir compte de la distribution variable de la charge de calcul lors du partitionnement logiciel/matériel des tâches.

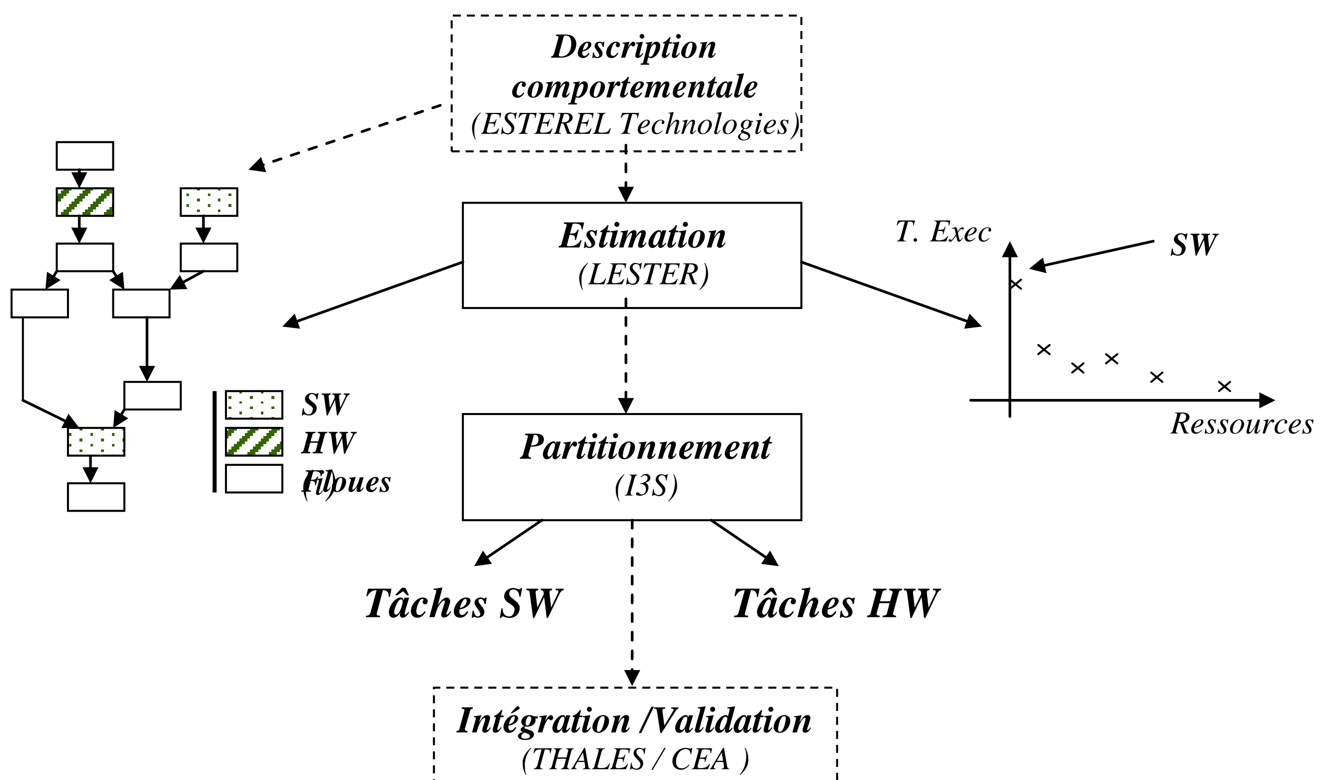


Figure 1 Organigramme de dépendance du projet EPICURE

Dans le cadre de ce mémoire, nous utilisons une approche de partitionnement basée sur un algorithme génétique avec une heuristique de clustering permettant de construire une architecture et un schéma d'exécution temps réel adaptatif de l'application. Cette approche va tenir compte des caractéristiques des données traitées, (par exemple les caractéristiques des images qui arrivent). Ces caractéristiques seront évaluées en temps réel au moment de l'exécution de la nouvelle donnée.

Pour expérimenter nos travaux, nous considérons le système EXCALIBUR d'ALTERA qui intègre sur un puce un processeur RISC et une unité reconfigurable [4].

Ce mémoire est organisé comme suit :

Dans un premier chapitre, nous présentons un état de l'art sur le partitionnement logiciel/ matériel ciblant les architectures enfouies reconfigurables. Nous montrons ainsi l'insuffisance des approches classiques de partitionnement pour certains domaines d'application tel que le traitement d'images. Le chapitre 2 introduit l'application de traitement d'image que nous avons développée durant la période de ce projet. Une idée sur l'architecture cible considérée dans notre approche est également présentée dans une deuxième partie de ce chapitre.

Le chapitre 3 est consacré à l'approche de transformation d'un graphe de flots de données, où les temps d'exécution des tâches sont dépendants des données, en un graphe de flots de données conditionné à temps d'exécution fixe. Nous expliquons toutes les étapes de cette approche ainsi qu'une idée sur l'intégration de l'application sur l'architecture cible. Dans le chapitre 4 nous revenons sur l'outil de partitionnement utilisé dans notre approche, qui est un algorithme génétique associé à une heuristique de clustering. Le dernier chapitre est relatif aux résultats. Une comparaison entre différentes techniques de reconfiguration est établie.

Enfin, une conclusion termine ce mémoire et évoque les différentes perspectives possibles par rapport au travail mené et aux résultats obtenus.

Chapitre 1

Etat de l'art du partitionnement logiciel/matériel

I.1 Introduction

Les techniques et les capacités d'intégration des circuits électroniques n'ont cessé de s'accroître au cours de ces dernières années. L'intégration de systèmes toujours plus complexes nécessite la définition de nouvelles méthodes de conception afin d'exploiter pleinement les évolutions technologiques.

Dans une approche classique la conception du logiciel et celle du matériel sont séparées et ce n'est qu'à la fin du processus de conception que les différentes parties sont testées ensemble (voir figure 1.1 (a)). Avec cette approche, une réalisation générée risque de ne pas répondre aux contraintes imposées (contraintes temps réel, coût, performances, consommation, fiabilité…). En plus, cette réalisation peut contenir des erreurs de conception aux interfaces logicielles /matérielles. Dans ce cas, il faut reprendre le cycle de conception.

Pour surmonter les limites de l'approche classique et pouvoir aboutir à des réalisations efficaces, l'approche de conception conjointe (Codesign) a vu le jour.

Dans ce chapitre nous allons introduire cette nouvelle approche ainsi que ses différentes méthodes de partitionnement adoptées dans la littérature et leurs limites face à certaines applications.

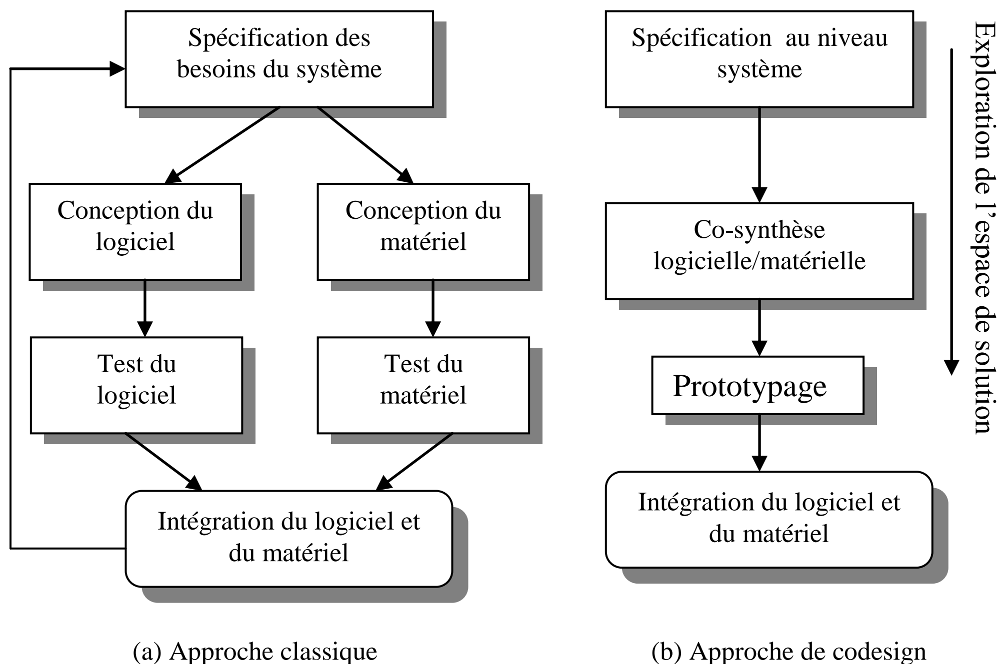


Figure 1.1 : La stratégie de conception d'un système

I.2 Partitionnement logiciel/matériel dans le cadre du Codesign

Le mariage de la conception logicielle et matérielle, appelé Codesign, permet de maîtriser la conception des systèmes complexes et de promouvoir leur pérennité afin d'augmenter la productivité. L'analyse des performances avant la fabrication permet une exploration rapide de plusieurs alternatives d'architectures ce qui offre au concepteur une meilleure visibilité et une grande réactivité vis-à-vis des changements technologiques (fiabilité, optimisation, flexibilité, etc.).

Le processus de codesign inclut donc plusieurs étapes de raffinement de la conception et des tâches de validation. Partant d'une spécification au niveau système, l'architecture sera raffinée jusqu'à la définition de l'architecture logicielle/matérielle de réalisation. Le processus de conception est donc un chemin progressif, à la fois pour déterminer une solution fonctionnellement appropriée et pour exprimer et évaluer les performances à chaque niveau de développement avec la prise en compte de l'aspect performances.

Les étapes de conception sont : (voir figure 1.2) :

- La décomposition fonctionnelle : l'exploration concerne la recherche d'une conception, qui satisfait les besoins fonctionnels, mais aussi les objectifs de performances. Quelques fonctions peuvent être réutilisables. Nous avons intérêt à construire une décomposition qui fait apparaître la même fonction autant de fois qu'elle peut exister dans le processus. Ceci nous permet d'utiliser la même implémentation pour cette fonction.
- Le partitionnement logiciel/matériel : Un des points-clefs du Codesign est d'obtenir une bonne répartition des tâches entre les unités logicielles (processeur, DSP...) et les unités matérielles (FPGAs, ASIC…) en fonction des ressources disponibles et des architectures. Le partitionnement doit prendre en considération un certain nombre des contraintes à respecter et de performances à atteindre. Nous pouvons considérer la contrainte temps réel, le coût de réalisation, la consommation, etc. Depuis plusieurs années, de nombreux travaux (DeMicheli, Ernst, Israel, Wolf, Kalavade…) ont proposé des approches pour partitionner une application. Les solutions automatiques restent cependant rares, et l'intervention des concepteurs semble nécessaire.

  Il est à noter ici que la méthode de partitionnement adoptée doit prendre en considération le type de l'architecture cible.
- La Co-synthèse : C'est la synthèse des partitions logicielles et matérielles ainsi que la communication. Dans cette étape aussi, le concepteur peut prendre en considération des composants SW ou HW réutilisables.

- Evaluation (co-simulation) : Il s'agit de vérifier le bon fonctionnement du système et les contraintes liées au contexte de l'application du système.

Dans la littérature plusieurs environnements de conception conjointe ont été développés. L'environnement Ptolemy a été utilisé dans [5] [6] [7] [8] [9] [10] [11]. Les approches de Codesign utilisée dans [12] [13] [14] [15] [16] utilisent plutôt le système VULCAN. Un autre environnement de Codesign appelé SpecSyn a été utilisé également dans [17] [18] [19].

Dans ce mémoire nous nous intéressons à l'étape du partitionnement logiciel/matériel. Dans la suite de ce chapitre nous allons présenter quelques approches de partitionnement qui sont déjà employées.

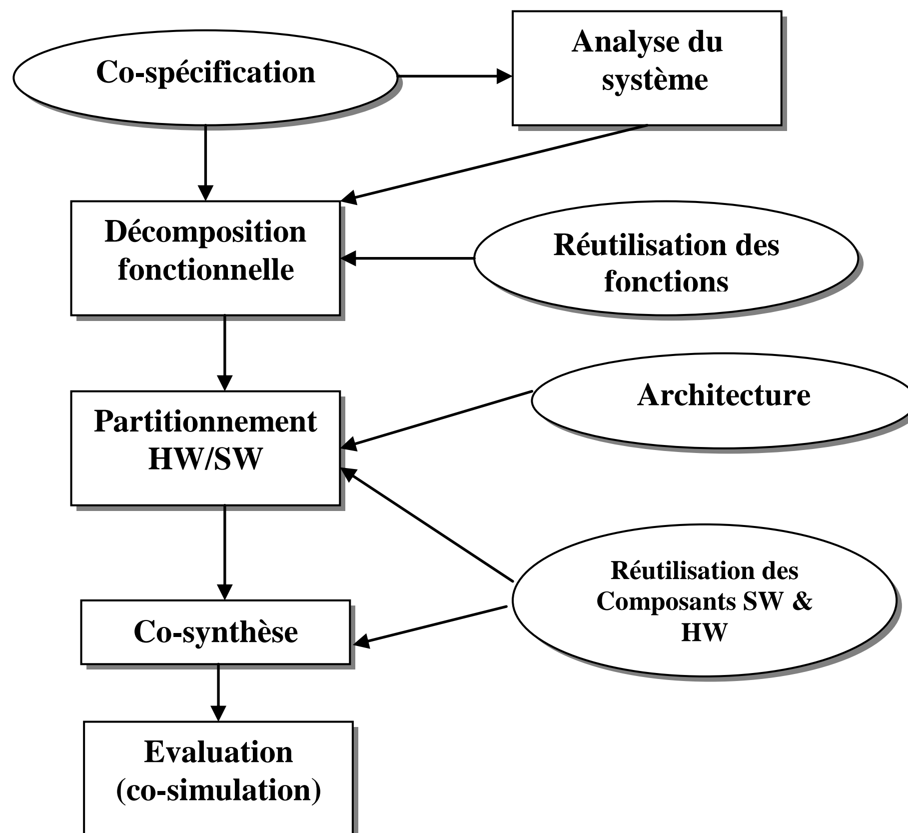


Figure 1.2. Processus d'exploration de l'espace de conception dans le CODESIGN
Source : Ernst (IEEE. D&T of computer)

I.3 Approches de partitionnement

Le partitionnement sert à distribuer les fonctionnalités d'une spécification système sur un ensemble des unités d'une architecture cible. Etant donné un graphe de tâches (ou de processus), le but est de transposer les différentes tâches sur une architecture donnée (voir figure 1.3).

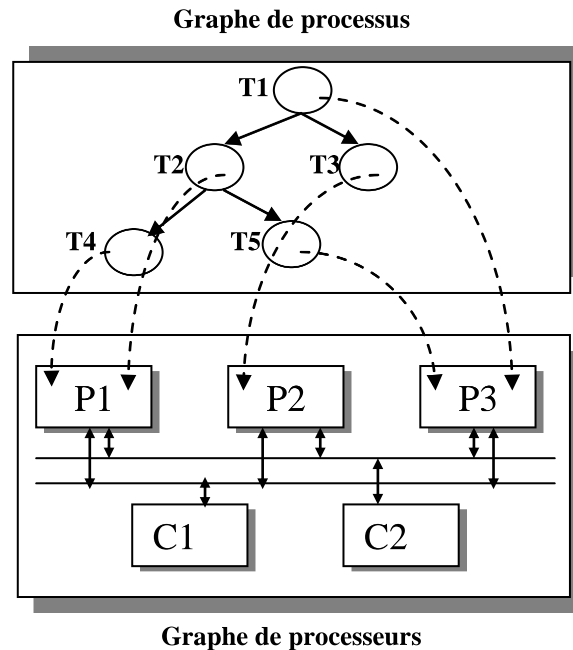


Figure 1.3. Le partitionnement au niveau système

Il s'agit de prendre les décisions qui permettent de répondre aux questions suivantes :

- Quelle fonction coût à optimiser ?
- Dans quelle technologie (logicielle ou matérielle) doit être exécutée chaque tâche ?
- Quelle structure d'architecture faut-il pour exécuter une description donnée ?

Dans la figure 1.3, P1, P2, et P3 désignent des processeurs de technologie logicielle ou matérielle ; C1 et C2 désignent des unités de communication, tel qu'un arbitre de bus ou des micro-contrôleurs. Dans cet exemple le partitionnement a affecté la tâche T1 au processeur P3, T4 au processeur P1 etc.

Le problème de partitionnement est un problème d'ordonnancement et d'allocation à plusieurs paramètres donc NP-complet.

Actuellement, il existe trois cultures différentes des chercheurs qui travaillent sur le partitionnement :

1. Les premiers sont de culture logicielle [20]. Ils commencent donc à partir d'une spécification entièrement logicielle et cherchent à faire de la migration de code vers le

matériel. Les parties critiques d'un système sont identifiées puis affectées à une réalisation matérielle.

2. Les deuxièmes sont de culture matérielle [12]. Ils partent d'une spécification initiale entièrement en matériel. Par la suite, les parties non critiques sont identifiées afin de les affecter à une réalisation logicielle, ce qui permet de réduire le coût de réalisation.
3. Les troisièmes sont de culture système (concepteurs de systèmes). Ils ne se limitent pas à un type particulier de spécification en entrée. Dans ce type d'approche, les différentes parties d'une spécification sont affectées à une réalisation (logicielle ou matérielle) qui satisfait les contraintes de conception (e.g. temps de réponse) [17].

L'affectation d'une tâche vers une réalisation logicielle ou matérielle se base sur une fonction coût à minimiser. Dans la suite nous allons répertorier les approches de partitionnement suivant le type d'architecture cible, la méthode de partitionnement adoptée et les critères d'optimisation retenus.

I.3.1 Architecture cible

Toutes les méthodes de partitionnement ciblent un type d'architecture ou une classe d'architecture. Les méthodes de partitionnement proposées dans [20] et [12] supposent une architecture composée d'un seul processeur, un seul circuit intégré (ASIC), une mémoire, et un bus. D'autres méthodes [8] [21] prennent en considération une architecture composée d'un seul processeur avec plusieurs circuits intégrés (ASICs).

Les approches de partitionnement dans [22] [21] utilisent un banc de test composé par des FPGAs et des interconnexions existantes dans la carte mère d'un ordinateur à usage général.

Pour des raisons de simplification, une architecture de réalisation prédéfinie est utilisée dans [23], [24]. Cette architecture est formée par un noyau RISC (*Reduced Instruction Set Computer*), une mémoire principale, un ensemble de circuits spécifiques et un circuit d'interface pour le contrôle de la communication entre le processeur RISC et les ASICs. Il s'agit dans ce cas d'une architecture du type maître/esclave où le RISC contrôle l'activation de chaque ASIC.

Enfin les approches de partitionnement développées dans [25] et [26] ciblent une architecture constituée d'un processeur connecté à un composant reconfigurable dynamiquement. L'application est décrite en langage C dans [25] ou modélisée suivant un graphe de flots de données dans [26].

I.3.2 Méthode de partitionnement

Face à la complexité du problème de partitionnement logiciel/matériel, plusieurs approches ont adopté des méthodes manuelles pour affecter chaque tâche vers l'entité correspondante sur l'architecture. D'autres préfèrent que le partitionnement soit semi-automatique, il s'agit de combiner à la fois la conception manuelle et la conception automatique. L'approche dans ce cas, suppose que le concepteur commence la synthèse à partir d'une spécification initiale des fonctions d'un système et ayant en tête une solution d'architecture. Toutes les transformations de raffinement sont réalisées de manière automatique mais toutes les décisions de raffinement sont prises par le concepteur qui peut utiliser son savoir faire et son expérience pour converger vers une solution efficace.

La troisième catégorie d'approche est celle qui utilise une méthode automatique ; dans ce cas un algorithme d'optimisation, qui prend en compte tous les paramètres du problème, sera adopté. Souvent des heuristiques seront privilégiées dans ce dernier cas.

Dans la suite, nous citons quelques méthodes de partitionnement qui sont adoptées dans la littérature et qui sont manuelles, semi-automatique ou automatique.

L'approche utilisée à l'Université de Californie/Berkeley sous l'environnement Ptolemy considère un partitionnement manuel. Ce partitionnement est guidé par les contraintes de surface, de vitesse et de flexibilité. Le concepteur essaie d'optimiser des fonctions coûts telles que : coût des communications, espaces mémoires locales et globales, etc. [5] [6] [7] [8] [10] [9] [11].

Dans [12] [13] [14] [16] l'approche de partitionnement commence avec une partition initiale où toutes les opérations, excepté celles à délai non limité, sont affectées au matériel. La partition est raffinée par la migration d'opérations du matériel vers le logiciel afin d'obtenir une partition à moindre coût.

Dans la méthodologie de l'Université de Tubingen [23] [24] la phase de partitionnement vise essentiellement à maximiser le parallélisme entre les composants, supporter la réutilisation des structures matérielles, et minimiser les coûts de communication et l'utilisation des ressources. La méthode de vérification est basée sur la simulation. Les composants matériels sont spécifiés en VHDL, alors que les composants logiciels sont spécifiés en langage C.

Un exemple de partitionnement manuel est celui adopté à l'Université Cincinnati dans le cadre du système RAPID [27]. Dans cette approche, le concepteur essaye différentes alternatives à partir d'une réalisation purement logicielle jusqu'à une réalisation purement matérielle. Les résultats de temps (timing) obtenus après simulation aident le concepteur à choisir les composantes logicielles qui doivent être réalisées en matériel.

Une autre approche de partitionnement est adoptée à l'Université de Linkoping [28], elle est formulée comme un problème de partitionnement de graphes. Elle réalise la transposition d'un graphe de flux de contrôle et de données sur un graphe unique de partitionnement. Différents types d'arcs sont utilisés entre les nœuds pour refléter les dépendances. Le partitionnement divise la fonctionnalité d'un système en un ensemble de modules, chacun correspondant à une unité physique (puce) ou à un paquetage logiciel. Une fonction coût guide le processus de partitionnement. Elle tient compte des coûts de communication et de synchronisation. Cette approche utilise les résultats de simulation pour identifier les parties critiques du système. Ces parties critiques sont alors réalisées en matériel.

La méthode adoptée à l'Université de Manchester [29] consiste à utiliser un outil d'analyse de performances à posteriori (*Profiling*). Cet outil d'analyse permet d'identifier les parties critiques d'un programme en C. Le partitionnement matériel/logiciel est effectué sur la base de l'analyse des performances. Les composants logiciels sont formés par du code C et des appels au coprocesseur alors que les parties matérielles sont traduites en HardwareC.

Nous citons aussi la méthodologie de l'Université de Braunschweig dans le système COSYMA [20] [30] [31], qui a le même principe d'analyse de performances que celui de l'Université de Manchester. La spécification est traduite, dans ce cas, en C^x dans une représentation interne appelée graphe syntaxique étendu. Ensuite, une simulation préliminaire et un profilage sont réalisés.

L'outil de partitionnement logiciel/matériel, dans ce cas, est automatique et basé sur un algorithme de recuit simulé. La stratégie adoptée pour le partitionnement logiciel/matériel consiste à démarrer d'une solution purement logicielle ensuite à extraire les parties critiques pour les réaliser en matériel en exploitant les informations obtenues par analyse de performances. Les parties à réaliser en logiciel sont traduites en langage C et les parties à réaliser en matériel sont traduites en langage HardwareC. Le système de partitionnement vérifie si les contraintes sur le temps sont satisfaites et localise les fautes.

Enfin nous terminons notre étude bibliographique sur les méthodes de partitionnement logiciel/matériel, par des approches récentes ; celles qui utilisent une théorie de clustering par exemple [32][33][25] pour privilégier le regroupement sur une même unité des fonctions ou tâches qui communiquent. Cette technique permet de minimiser le temps d'exécution total en annulant les temps de communication entre les tâches regroupées dans un même cluster.

I.3.3 Les critères d'optimisation

Avant d'appliquer l'algorithme de partitionnement, il faut tout d'abord spécifier les paramètres de la fonction coût. Ceci joue un rôle très important lors du choix de l'architecture cible, parce que selon les critères à optimiser, nous affectons les entités de calcul correspondantes aux entités de l'architecture.

Dans la littérature, les critères d'optimisation sont assez variés. La plupart des approches avec une contrainte de temps réel [20] [12] [34] avait pour objectif de rechercher une accélération des traitements en profitant des temps d'exécution plus faibles sur la partie matérielle de manière à respecter un objectif de performances tout en minimisant la surface de silicium induit par la partie matérielle, ce qui revenait à considérer un compromis temps d'exécution /(surface de silicium ~ consommation) avec une échéance sur le temps d'exécution final.

Dans le système Vulcan [12] [13] [14] [16], les critères à optimiser sont le temps d'exécution avec des contraintes de délai Minimum/Maximum. A l'Université Californie/Ivrine et dans le cadre du système SpecSyn [17] [18] [19], les algorithmes de partitionnement cherchent à minimiser le temps d'exécution ainsi que la taille de l'exécutable et de celle des données.

I.4 Limites des approches classiques

Toutes les approches citées dans le paragraphe précédent (**I.3**), considèrent que les fonctions de l'application possèdent des temps de calcul constants pour un nombre des ressources fixé. Or, de nombreuses applications en particulier en traitement des images, font apparaître des charges de calcul variables en fonction des images à traiter. C'est le cas d'un problème d'étiquetage d'image dont la répartition des volumes de traitements entre les fonctions de l'algorithme varie suivant le nombre et la taille des objets contenus dans l'image. Autre exemple typique de ce problème est l'extraction des contours dont le temps d'exécution est fortement corrélé au nombre et la taille des objets existants dans l'image.

Si nous nous situons sur l'organigramme de dépendances du projet EPICURE (figure 1), nous devons recevoir les résultats des estimations faites par le laboratoire LESTER ; ces estimations ne sont plus valables pour les applications dont les tâches ont des temps d'exécution fortement corrélés aux jeux de données.

Il sera alors nécessaire, et avant d'entamer la phase de partitionnement, de chercher d'autres estimations qui prennent en compte la variation de charge de calcul d'une image à une autre.

Nous allons proposer une méthode de partitionnement qui soit adaptative vis à vis des variations de la charge de calcul variant d'une donnée à une autre.

I.5 Conclusion

L’état de l’art du partitionnement logiciel/matériel présenté dans ce chapitre n’a pas vocation à être exhaustif mais s’efforce de montrer la diversité des approches proposées et les difficultés liées au problème du partitionnement. Nous avons pu remarquer que ce problème est approché de nombreuses façons suivant les méthodes de partitionnement et les architectures considérées. Il fallait donc identifier tout d’abord l’application et son modèle et spécifier un type d’architecture bien déterminé. Ceci sera l’objet de notre prochain chapitre qui présentera en détail l’application sur laquelle nous avons travaillé dans le cadre de ce mémoire puis l’architecture que nous avons ciblée dans notre approche de partitionnement.

Chapitre 2

Applications de traitement d’images et choix d’architecture correspondante

II.1 Introduction

Les applications de traitement du signal et de traitement d'images font apparaître des traitements intensifs sur des structures des données régulières au niveau de données brutes et plus irrégulières lorsque les informations deviennent plus abstraites. Classiquement ces applications sont représentées par des graphes de flots de données, formalisme bien adapté pour décrire leur comportement indépendamment d'une implémentation logicielle ou matérielle. Dans le cadre de ce mémoire nous avons appliqué une nouvelle approche de partitionnement sur une application de traitement d'image. Cette application qui réalise la détection du mouvement sur un fond d'image fixe, nous a été envoyée par le CEA dans le cadre de projet EPICURE. La version envoyée est purement logicielle, elle tourne à une cadence de 130 ms par image. Notre travail consiste à trouver le partitionnement convenable de cette application, tout en ciblant l'architecture adéquate, pour la faire tourner en temps réel (25 images par secondes).

Dans une première partie de ce chapitre, nous allons donner une idée générale sur la modélisation des applications de traitement d'image en particulier l'application de détection de mouvement.

La deuxième partie est consacrée à étudier le choix de l'architecture cible dans un processus de partitionnement.

II.2 Modélisation des applications de traitement d'images

Les applications de traitement des images peuvent aisément se modéliser par des graphes de flots de données (*DFG : Data Flow Graph*) au moins dans les niveaux bas et moyen de traitement. La description initiale du système est un graphe dirigé sans cycle. Les nœuds du graphe représentent les traitements (calculs) et les arcs du graphe représentent les dépendances de données (voir figure 2.1).

Pour ce type de modélisation, le partitionnement consiste à déterminer pour chaque nœud son type d'implantation (processeur , ASIC, FPGA …), son allocation (numéro de la ressource du type choisi, s'il y en a plusieurs) et son ordonnancement, avec des contraintes sur les temps de calcul entre nœuds initiaux et terminaux du graphe. Toutes les communications entre les nœuds sont calculées et prises en compte dans le calcul du temps d'exécution totale.

Il est à noter ici que lors du choix du modèle pour l'application, il faut préciser le niveau de granularité, il s'agit de choisir la taille des données traitées par les opérations. Le niveau de granularité intervient comme paramètre décisif lors du choix de l'architecture cible. Il décide aussi les résultats du partitionnement : en effet une granularité assez fine

permet d'avoir plus de parallélisme et plus de tâches, par contre une granularité assez élevée diminue la complexité du système.

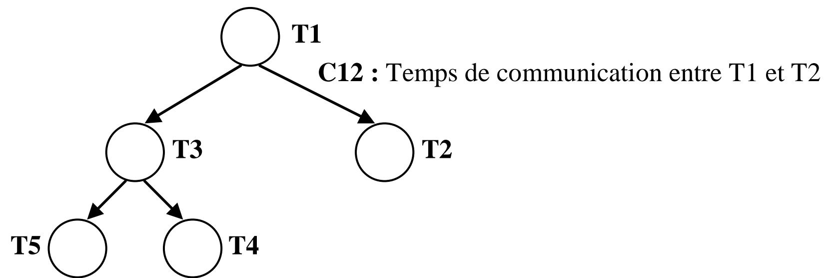


Figure 2.1 graphe de flots de données (DFG)

II.3 Détection du mouvement sur un fond d'image fixe

L'analyse de mouvement dans les séquences d'images est un domaine de recherche active à l'heure actuelle en raison de son importance dans de nombreuses applications : télésurveillance, compression pour les télécommunications ou l'archivage, diagnostic médical, météorologie, contrôle non destructif, robotique mobile, etc.

Le principe de la caméra intelligente, développée par le CEA et ses partenaires, se base sur le concept de la détection de mouvement dans une séquence d'images. Les principaux objectifs de cette caméra sont : la vidéo-surveillance à usage industriel et pour la sécurité des biens et des personnes.

Pour le cas de la vidéo-surveillance à usage industriel, nous pouvons citer des exemples d'application tels que :

- Le contrôle dimensionnel (largeur, position/guidage, épaisseur) sur produits plats ou ronds, fixes ou en défilement.
- La détection de non conformité sur matériaux en défilement continu.
- Comptage de pièces, contrôle de présence/absence
- Identification (code à barres…)
- Tri des pièces par largeur, diamètre, état de surface…
- Endoscopie médicale.

La caméra utilisée dans ces types d'applications, devra posséder des résolutions et des cadences élevées, qui permettent de transcrire fidèlement les variations de l'information vidéo, même petites ou rapides, au système informatique.

Pour la sécurité des personnes et des biens, les applications sont multiples : surveillance d'entrepôts, de grande surface, de sites sensibles éloignés, de parking , de sites privés, etc.

II.3.1 Schéma fonctionnel

La caméra intelligente, étudiée par le CEA, est composée comme le montre la figure 2.2 de :

- ✓ Un capteur CMOS.
- ✓ Un processeur à logique reconfigurable.
- ✓ Une interface pour des communications avec l'extérieur.

Le CEA s'occupe du processeur à logique reconfigurable et des échanges de données avec le capteur CMOS. Le processeur est composé d'un DSP (ou processeur RISC) et d'une logique reconfigurable (FPGA par exemple). La logique reconfigurable sera programmée dynamiquement par le processeur pendant le traitement. Le processeur à logique

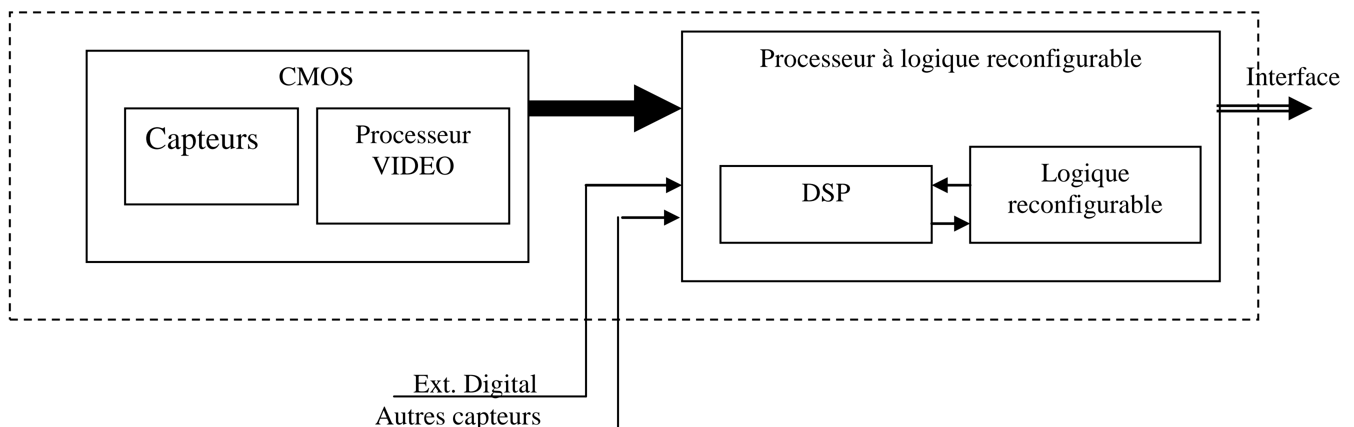


Figure 2.2 Architecture d'une caméra intelligente

reconfigurable possédera une entrée dédiée à d'autres capteurs tel qu'un CCD et une entrée pour des données numériques pour un signal vidéo compressé ou non, un GPS…

II.3.2 Principe de fonctionnement

Le but du traitement est de détecter des éléments en mouvement sur un fond d'image fixe (qui sert comme image de référence) et de les suivre (figure 2.3).

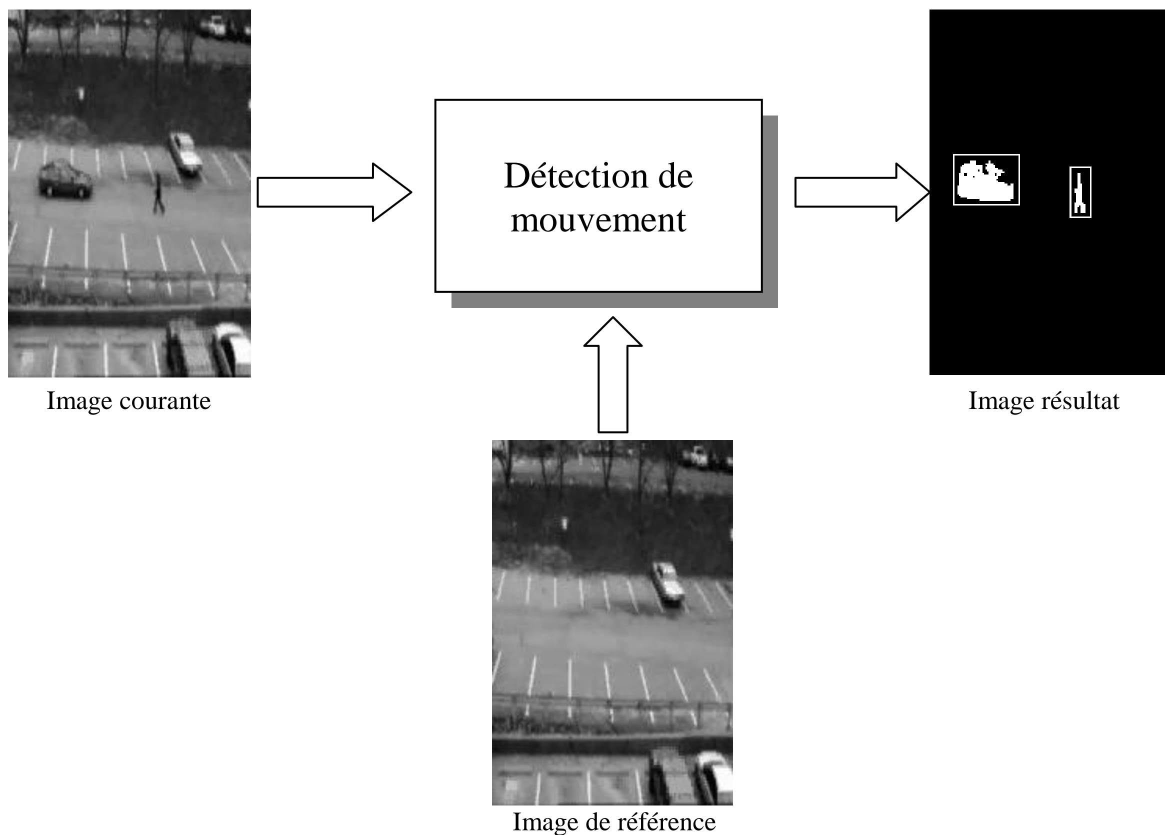


Figure 2.3 principe de la détection de mouvement

II.3.3 Organigramme du processus de détection de mouvement

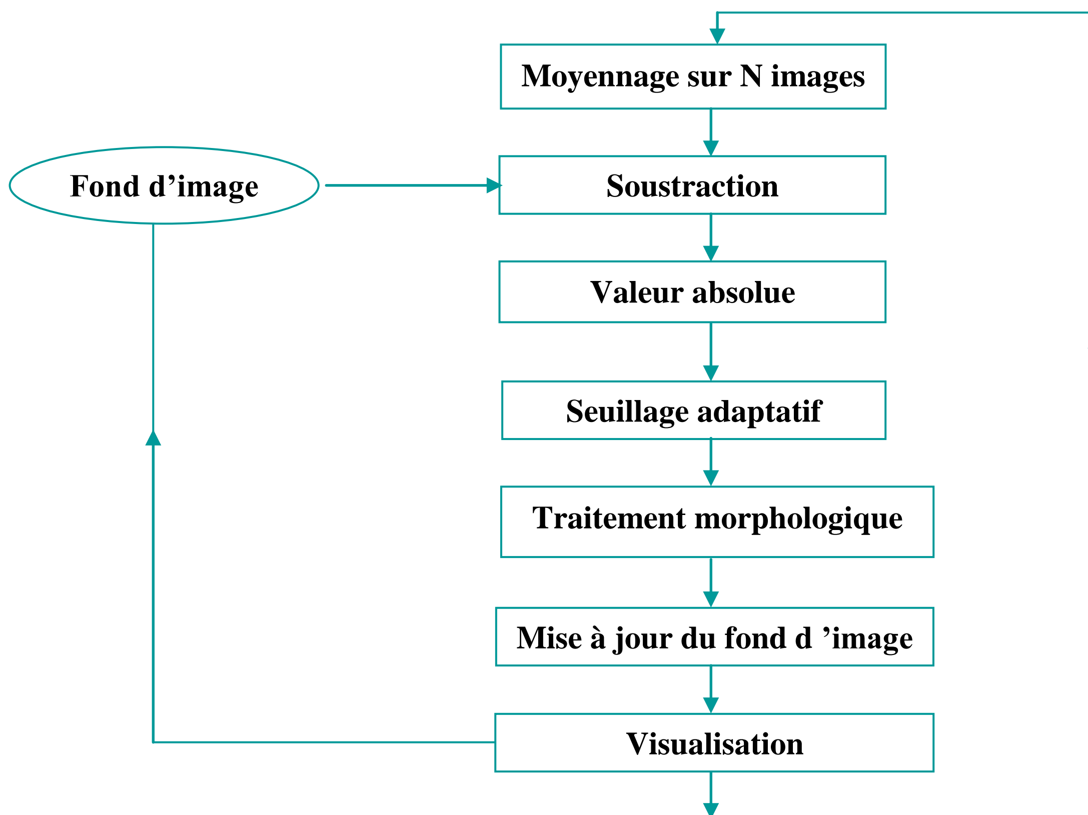


Figure 2.4 Organigramme du processus de détection de mouvement

La programmation du processus de détection de mouvement (figure 2.4) est structurée comme suit :

Une interface codée en C++ a été développée avec Visual C++ 6.0. Cette interface (figure2.5) fait appel à des algorithmes de traitement d'images programmés en C, appliqués à une détection et un suivi de mouvement.

Les images sont alors chargées avec une fonction programmée en C++, elles subissent des traitements en C et enfin la visualisation est assurée par un code C++.

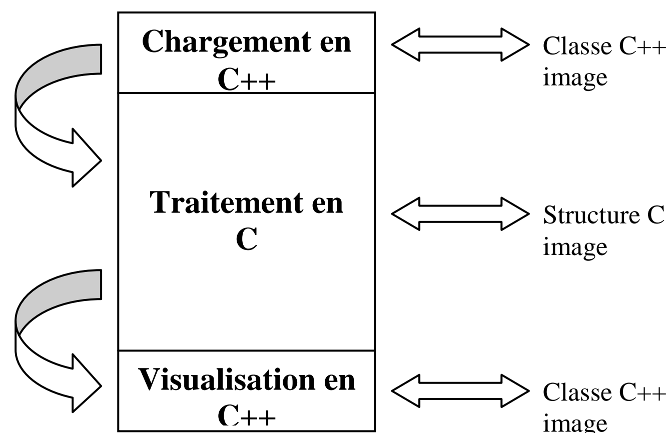


Figure 2.5 Structure du programme

II.3.4 Algorithmes de la détection de mouvement

Les codes de toutes les tâches, constituant le processus de détection de mouvement, sont écrits en langage C. Le travail effectué dans le cadre de ce mémoire, consiste à étudier une nouvelle approche de partitionnement pour ce type d'applications c'est à dire les applications qui contiennent des tâches à temps d'exécution qui changent avec la nature des données traitées. Il fallait donc, commencer par étudier les algorithmes de l'application du point de vue charge de calcul de chaque fonction et le temps d'exécution correspondant (le temps d'exécution est calculé sur un processeur). Une étude approfondie de toutes les fonctionnalités des algorithmes est faite, notre objectif était toujours d'analyser le problème de variation du temps d'exécution des fonctions d'une image à une autre (i.e d'une donnée à une autre). Dans la suite, nous présentons les fonctions qui constituent le processus de détection de mouvement sur un fond d'image fixe, en s'arrêtant sur les aspects de dépendance du charge de calcul pour chaque tâche.

a) Moyennage paramétrable

Afin que le traitement soit moins sensible d'une image à une autre, le Moyennage

consiste à additionner N images successives et les diviser par N (voir figure 2.6). Plus le bruit est grand, plus N est grand. Le paramètre N est défini aussi en fonction de la vitesse des objets ; si l'objet dans l'image a une vitesse de déplacement assez lente, N sera grand et inversement si l'objet se déplace rapidement.

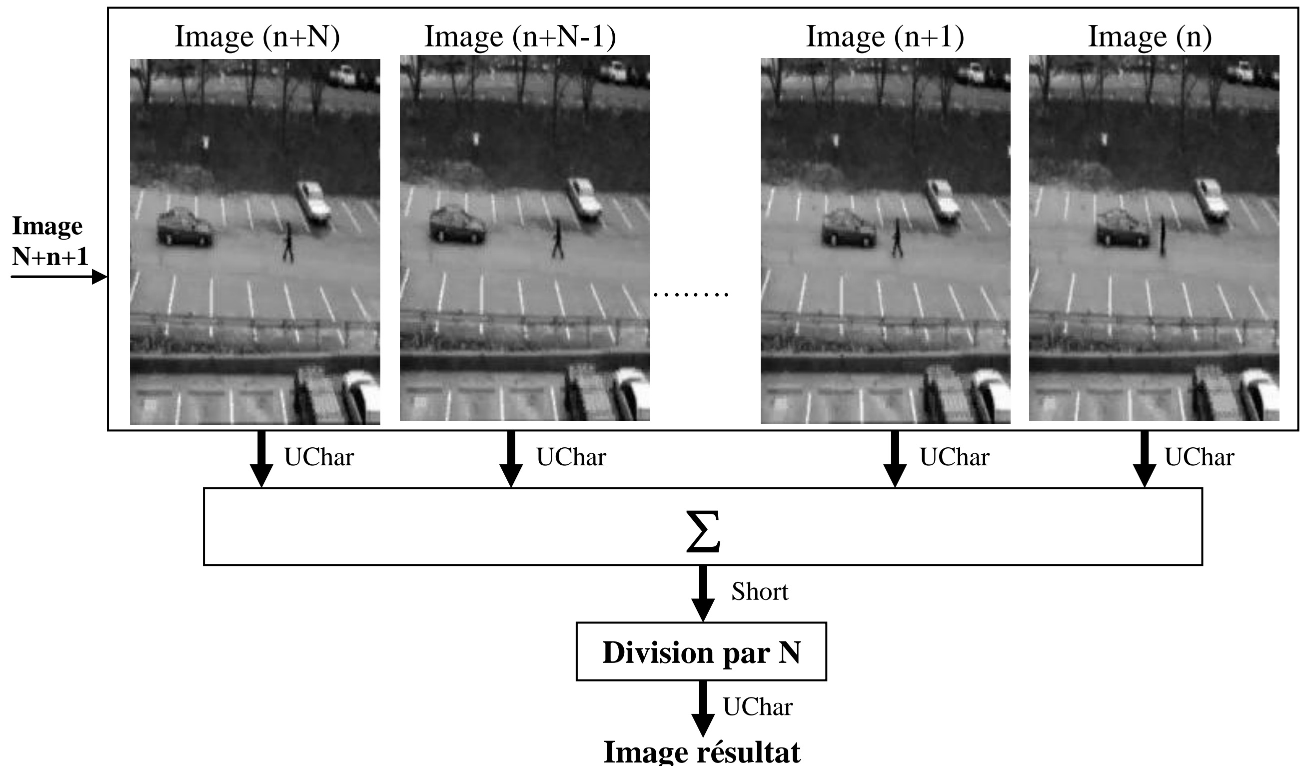


Figure 2.6 Schéma de principe du moyennage

En analysant le code C de la fonction moyennage paramétrable sur N image, nous remarquons que la charge de calcul ne dépend a priori que du nombre N des images à moyenner et de la taille de ces images. Nous rappelons ici, que les pixels des images sont codés sur un octet (une variable de type Unsigned Char : Uchar) et les résultats de l'addition ou soustraction sont codés sur deux octets (variable de type Short).

b) Soustraction

Il s'agit d'une différence entre une image de référence et l'image résultante du moyennage de N images. Elle permet de voir les zones en mouvement et les zones fixes.
L'image de référence est définie lorsqu'au moins un objet n'est plus en mouvement. Celui-ci sera, alors, ajouté à l'image.
La fonction effectue une différence pixel à pixel entre les deux images. Les zones fixes apparaîtront en noir car si l'on soustrait deux mêmes zones, on obtient une zone de niveau de gris égale à 0. Les zones en mouvement apparaîtront, quand à elles, avec un niveau de gris

différent de 0. La soustraction donne des niveaux de gris négatifs, d'où la nécessité de calculer la valeur absolue (figure 2.7).

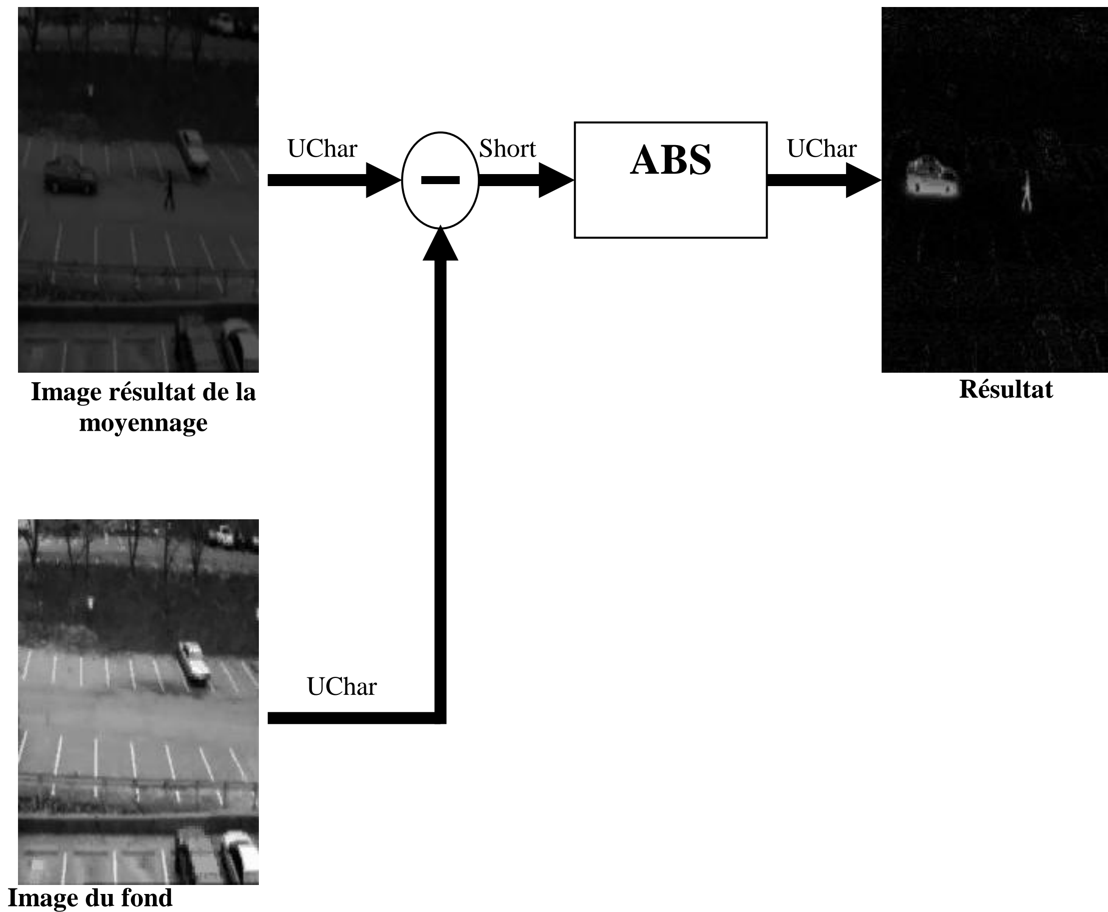


Figure2.7 Schéma de principe de la soustraction et de la valeur absolue

Les traitements des fonctions soustraction et valeur absolue se font au niveau pixel, donc la charge de calcul et le temps d'exécution induit dépendent du nombre de pixels existant dans l'image (c'est à dire de la taille de l'image traitée).

c) La fonction de seuillage

L'objectif est de binariser l'image afin d'isoler les objets se déplaçant. Le seuil de binarisation est calculé à partir du gradient de l'histogramme. Une valeur nulle du gradient de l'histogramme indique une région stable formée par les zones en mouvement d'où la valeur du seuil (voir figure 2.8).

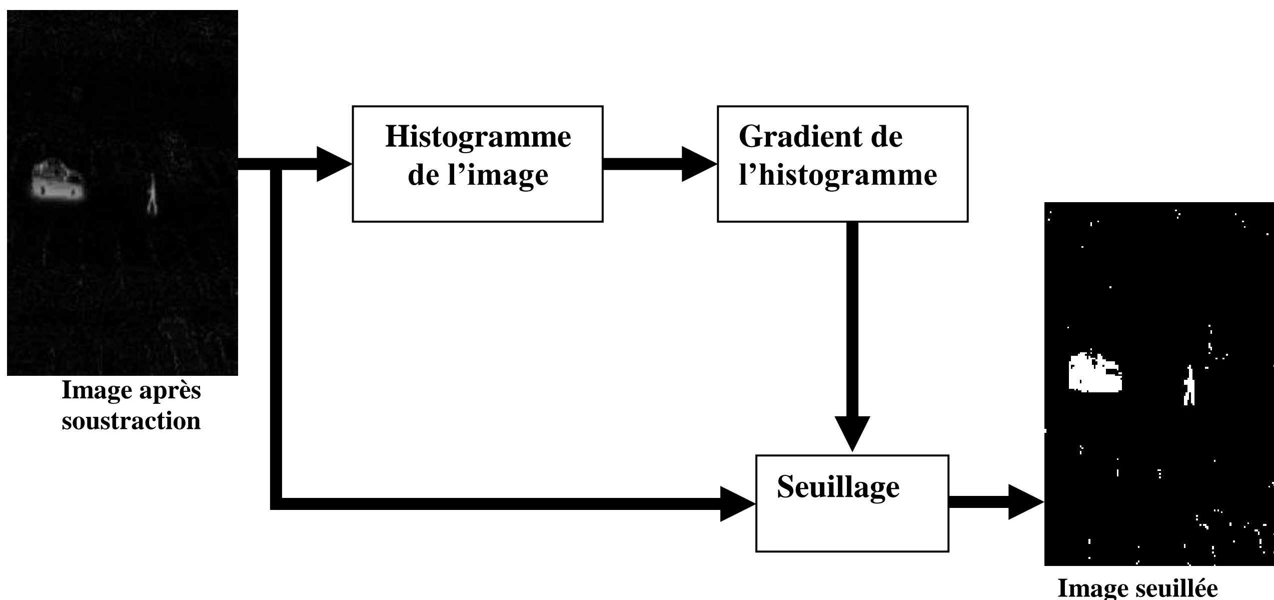


Figure 2.8 Schéma de principe du seuillage

A priori, tous les traitements contenus dans le seuillage adaptatif, ont des temps d'exécution qui ne peuvent dépendre que de la taille de l'image.

d) Traitements morphologiques de l'image

Afin de filtrer l'image, nous allons faire disparaître tous les points isolés. Pour cela, il faut appliquer une ouverture sur l'image qui consiste à pratiquer successivement une érosion sur l'image seuillée et une dilatation sur celle érodée.

Pour chaque pixel de l'image, le traitement sera en fonction des pixels voisins. Les voisins considérés sont ceux en bas, en haut, à droite et à gauche. Un masque de convolution (figure 2.9) sera alors appliqué sur l'image.

0	1	0
1	1	1
0	1	0

Figure 2.9 Masque de convolution

- **L'érosion**

Le principe est le suivant : pour chaque pixel blanc traité, le programme analyse son voisinage. Le pixel traité deviendra alors noir, si et seulement si, au moins un pixel connexe est de niveau 0 (noir).

L'érosion est une étude sur les pixels blancs qui forment les objets en mouvement dans l'image. Donc la charge de calcul induit sera sensible à la taille totale des objets.

- **La dilatation**

Afin de retrouver des formes qui se rapproche des objets en mouvement de l'image seuillée, une dilatation est appliquée à l'image. Il s'agit de balayer l'image érodée. Pour chaque pixel blanc traité, le programme analyse son voisinage. Si le pixel traité a au moins un pixel connexe (voisin) blanc, alors il prendra ce niveau de gris.

Avec cette définition de la dilatation, le traitement va concerner les pixels noirs qui forment la majorité de pixels de l'image. Donc la charge de calcul de cette tâche va dépendre de la taille totale de l'image.

- **L'ouverture par reconstruction**

La reconstruction après ouverture sert à retrouver la forme complète des objets de l'image seuillée qui sont toujours présents après ouverture. Pratiquement, la reconstruction consiste à balayer l'image pixel par pixel et pour chaque pixel blanc, nous recopions les pixels connexes correspondant de l'image seuillée. Comme pour l'érosion, la taille de traitement de la reconstruction dépend de la taille totale des objets en mouvement dans une image.

Le schéma (figure 2.10) représente le principe de tous les traitements morphologiques.

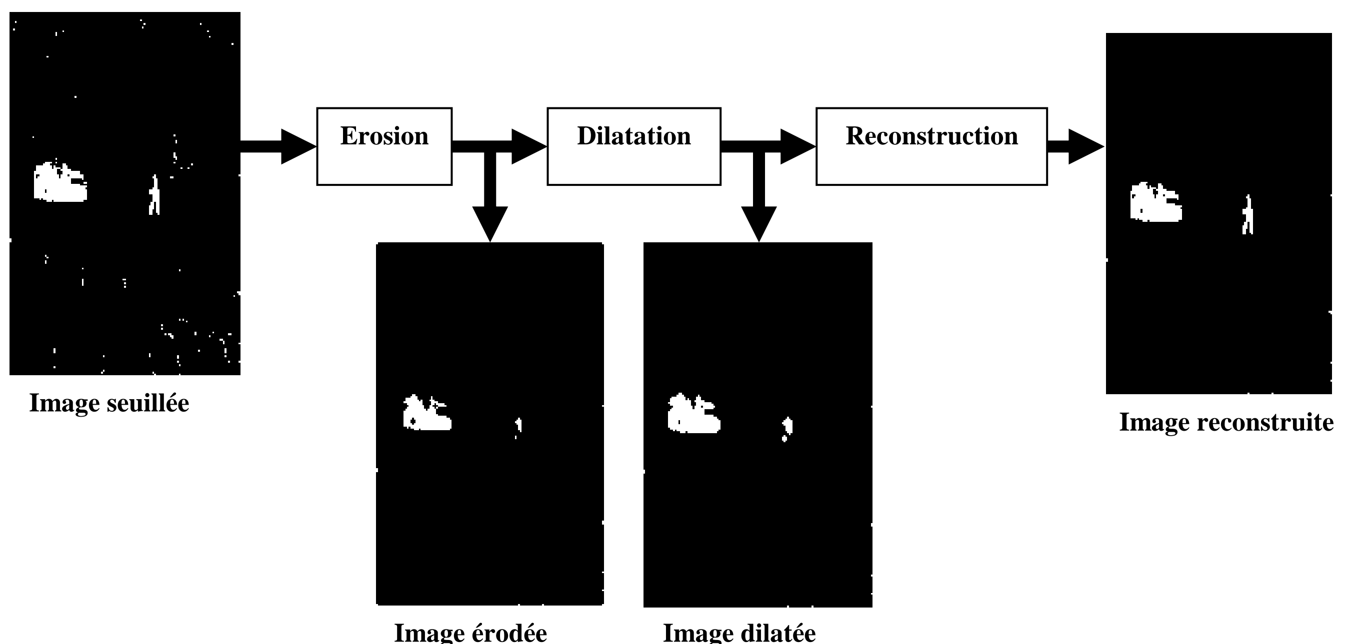


Figure 2.10 Schéma de principe des traitements morphologiques

e) Mise à jour de l'image de référence

Pour mettre à jour l'image de référence, il faut calculer le centre de gravité des objets. Si au bout d'une certaine période de temps (défilement de 10 image dans notre cas), un centre

de gravité garde les mêmes coordonnées, nous remettons à jour l’image de référence. Par contre, si le centre ne possède plus les mêmes coordonnées, alors nous le considérons comme objet en mouvement.

La mise à jour de l’image de référence se fait en trois étapes (figure 2.12):

- ✓ L’étiquetage, Il consiste à attribuer à tous les pixels connexes, qui forment un objet, la même étiquette.
- ✓ Le traçage de l’enveloppe englobante (voir figure2.11), il s’agit de tracer (pour chaque objet) une boite englobante possédant les mêmes coordonnées minimales et maximales que l’objet.

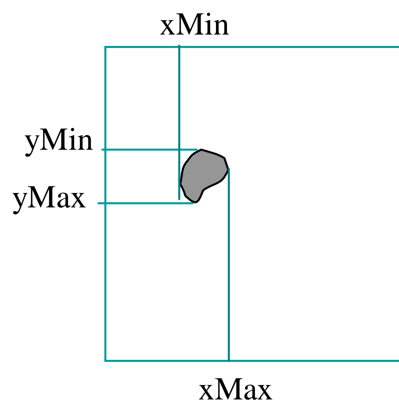


Figure 2.11 Coordonnées d’un objet dans une image

- ✓ Le test sur le déplacement, c’est une sorte de contrôle sur les positions des objets dans une image en comparant les centres de gravité des objets de l’image courante avec ceux de l’image précédente.

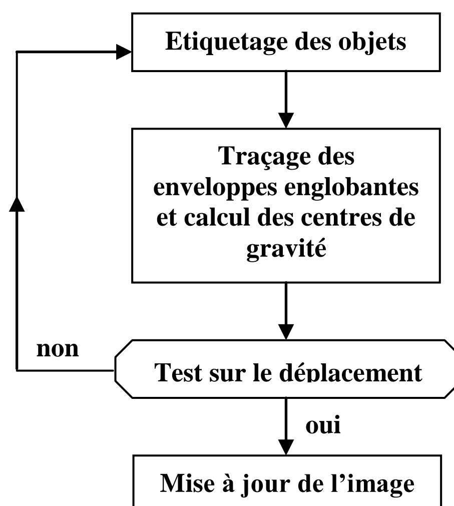


Figure 2.12 Principe de mise à jour de l’image de référence

L'étude du code C des fonctions de mise à jour de l'image de référence montrent que la procédure de l'étiquetage varie au terme de charge de calcul avec la taille totale des objets ; par contre la procédure de traçage des enveloppes englobantes varie avec le nombre des objets contenus dans l'image et enfin, le test sur le déplacement et la mise à jour dépendent de la nature des objets : la vitesse des objets, déplacement…

L'application étudiée (Détection de mouvement dans une séquence d'images) met en jeu plusieurs types d'opérations de traitement des images (sommation, soustraction, seuillage, histogramme, érosion, dilatation, étiquetage….). L'analyse de toutes ces tâches va permettre à l'approche de partitionnement d'être plus souple , plus générale et plus performante.
Vu les caractéristiques de ces types d'applications, à un calcul intensif (plus de 25 millions de pixels traités par seconde pour une image 1024 par 1024), nous avons réfléchi tout d'abord au type de l'architecture cible dans notre approche de partitionnement et ses composants qui doivent manipuler ces calculs. Ceci fera l'objet de la partie suivante de ce chapitre.

II.4 Architecture cible : hétérogène (processeur / FPGA)

Pour les applications nécessitant des phases de calcul intensif sur un grand nombre de données, tel que le traitement d'image, les systèmes reconfigurables (processeur associé à une unité reconfigurable) permettent d'une part, un parallélisme de traitement efficace (par exemple sous la forme de pipelines) et d'autre part, un débit important d'échange avec la mémoire.
Aujourd'hui, l'augmentation des capacités d'intégration a permis de développer des circuits logiques configurables de grande taille (FPGA). La puissance de calcul produite par les circuits FPGAs provient de leur spécialisation par rapport aux besoins des programmes exécutés (traitement sur mesure). Leur programmabilité permet de corriger, modifier, adapter la fonctionnalité de l'ensemble après fabrication de la carte. Aujourd'hui les FPGAs sont employés pour le prototypage rapide de systèmes numériques.
Les FPGAs sont utilisables sur des problèmes nécessitant plusieurs millions de portes logiques. Ils incluent des ressources de mémorisation et disposent de nombreuses broches d'entrées/sorties. La figure 2.13 montre l'évolution rapide de la capacité d'intégration fournie par les circuits FPGA.

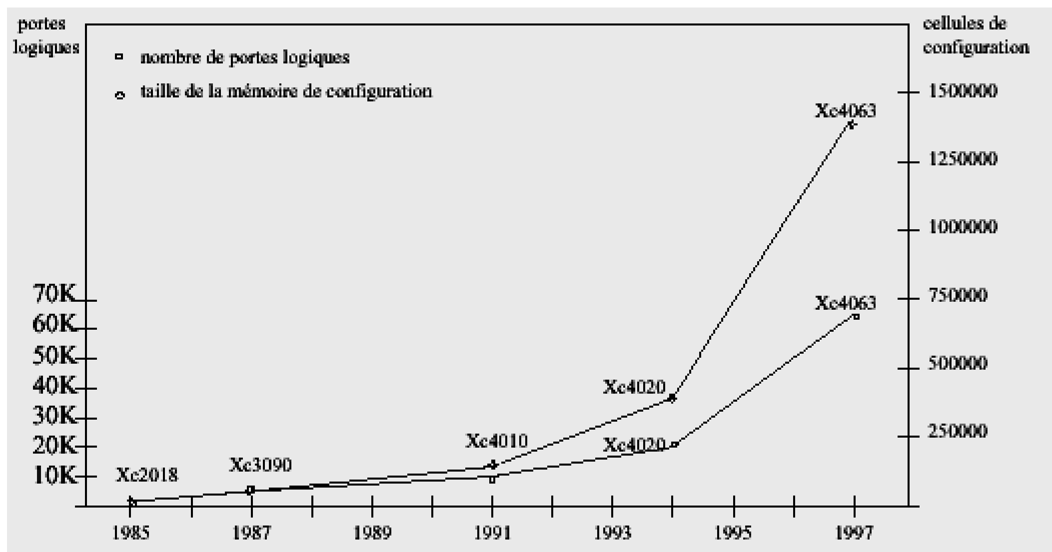


Figure 2.13 Capacité des circuits FPGA et taille de leur mémoire de configuration

Nous pouvons constater d'autres avantages des circuits reconfigurables relatifs à leur exploitation (figure 2.14). En effet avec un FPGA, le test ‘*in-situ’* sera possible sur l’application ; ce qui augmentera la fiabilité du système. D’autre part le cycle de conception,

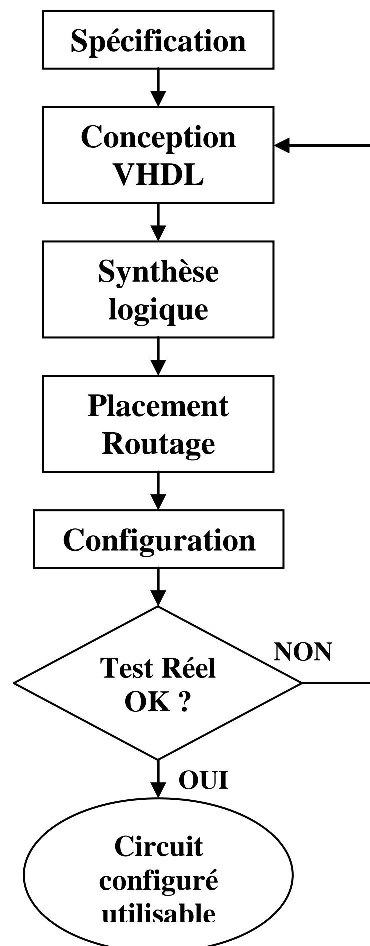


Figure 2.14 Processus de configuration d’un circuit

de test et de correction est réduit ce qui signifie un temps de conception global écourté…
La définition du comportement d'un système reconfigurable est laissée à la tâche de partitionnement, qui détermine l'implantation des fonctions matérielles des parties du traitement à accélérer et l'implantation logicielle pour le reste des fonctions.
Les architectures de systèmes reconfigurables (voir figure 2.14) sont très diverses et peuvent se caractériser par leur couplage avec la partie reconfigurable. Nous trouvons ainsi des processeurs qui possèdent leurs propres unités fonctionnelles reconfigurables (i.e. intégrées dans le chemin de données interne), d'autres qui accèdent à la partie reconfigurable via les bus externes du processeur, ou, enfin, des processeurs qui leur confèrent seulement un statut de périphérique[35].
Dans le projet EPICURE l'architecture considérée est constituée d'un processeur connecté à un circuit reconfigurable dynamiquement à travers une interface générique comme le montre la figure 2.15.

Figure2.15: système reconfigurable à interface externe

Ce type d'architecture est bien adapté pour concevoir des systèmes embarqués de type caméra intelligente. La flexibilité du reconfigurable permet d'adapter les traitements à l'environnement dans lequel est placée la caméra et la reconfiguration dynamique autorise une meilleure exploitation des ressources matérielles et donc de diminuer la surface de silicium. Actuellement, nous disposons au laboratoire le système EXCALIBUR d'Altera (figure 2.16) qui intègre dans un même circuit intégré (APEX 20KE) un processeur et une unité reconfigurable [4].
Des travaux sur cette carte ont été déjà entrepris en commençant par la génération des interfaces de communication entre le Nios (le processeur RISC dans le puce) et le reste de l'APEX (la partie allouée pour le FPGA). L'atout principal du système EXCALIBUR est le faite qu'il comprend un ensemble de solutions et d'outils qui permettent d'intégrer de façon souple sur un même réseau logique un CPU RISC, des mémoires, des périphériques et de la logique utilisateur. Le cœur Risc NIOS est disponible sous forme d'une description RTL, donc facilement adaptable par le concepteur à sa cible. Le processeur ainsi personnalisé sera ensuite fondu dans la logique programmable d'un FPGA au même titre que les autres

éléments du système visé. La souplesse d'une telle solution se paye par une implantation du CPU moins optimisée par rapport aux IP (*Intellectual Properties)* matérielles, mais permet des performances jusqu'à 80 MHz alors que la solution en dur développe jusqu'à 210 Mips.

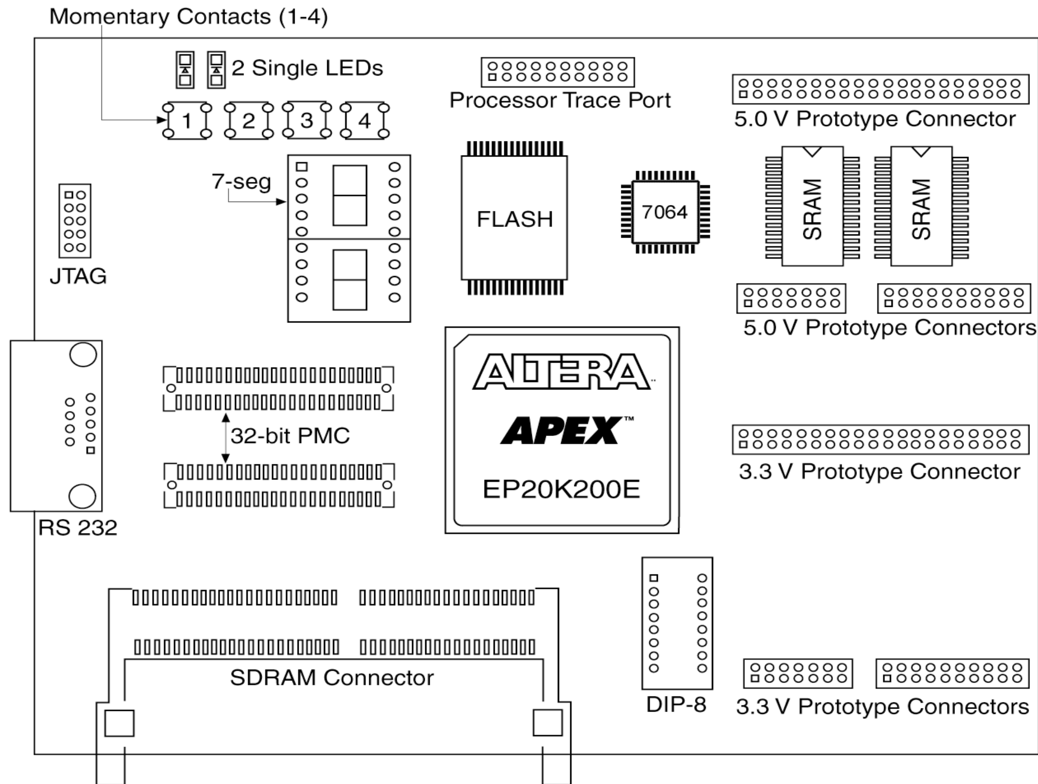


Figure 2.16 Système EXCALIBUR d'Altera

II.5 Conclusion

Pour pouvoir partitionner les applications de traitement d'image, il faut les modéliser sous le format adéquat. Nous avons vu dans ce chapitre que le modèle de DFG s'adapte bien avec ce type d'applications. Dans la deuxième partie de ce chapitre nous avons vu que les systèmes reconfigurables représentent une architecture cible très intéressante pour les applications de traitement de signal en général et particulièrement le traitement des images. La technologie de FPGAs montre des capacités attrayantes pour la résolution des problèmes de grandes puissances de calcul.

Dans le prochain chapitre, nous allons présenter notre nouvelle méthode de partitionnement, l'approche adoptée, les démarches suivies, les possibilités de réalisation,…

Chapitre 3

Nouvelle approche de partitionnement pour les applications à distribution variable de charges de calcul

III.1 Introduction

En traitement d'image de nombreuses opérations ont un temps d'exécution fortement corrélé au contenu de l'image traitée. Nous pouvons donc, considérer deux types de tâches : les tâches qui gardent un temps d'exécution constant pour les différents jeux de données et celles qui ont un temps d'exécution variable.

Pour développer une méthode de partitionnement d'applications contenant des tâches de ce dernier type, il faut tout d'abord définir la nature de la corrélation entre le temps d'exécution et les caractéristiques des données mises en jeu.

Le paramètre de corrélation est identifié par une étude préalable de la tâche concernée, suivi d'une confirmation à l'aide d'essais pratiques. Donc, tout d'abord il faut effectuer une sorte de *profiling* sur l'application considérée.

Dans ce chapitre nous présentons la méthodologie de notre approche de partitionnement logiciel/matériel suivie de quelques idées de l'intégration de l'application, détection de mouvement dans une séquence des images, sur une architecture hétérogène.

III.2 Profiling de l'application considérée

Par définition le profiling d'une application, consiste à donner ses caractéristiques et ses spécificités. Dans notre cas le profiling de l'application à partitionner est la première étape dans l'approche. Elle consiste à analyser les fonctions de l'application afin d'identifier les parties critiques au terme de charge de calcul (respectivement : temps d'exécution).

Dans la suite nous allons revenir sur l'application détection du mouvement sur un fond d'image fixe pour mesurer les temps d'exécution de chacune des tâches qui y forment. Pour chaque tâche nous avons effectuer une étude théorique sur les critères de dépendance de la charge de calcul (que nous l'avons déjà commencé dans le chapitre 2 lors de la présentation de l'application), suivi d'un tableau des mesures, une courbe et des interprétations des résultats.

Les mesures du temps d'exécution sont faites sur différentes images pour pouvoir tirer des interprétations générales. En associant à chaque tâche un compteur qui s'amorce à chaque lancement et donne le temps d'exécution total à la fin, les résultats de mesures seront enregistrés dans un fichier de type texte.

Les codes C de l'application sont tournés sur un processeur qui a les caractéristiques suivantes :

- Processeur x86 Family 6 Model 8 stepping 6
- CPU de vitesse 800 MHz

- Mémoire vive (RAM) : 128 Mo

D'autres part les images, sur lesquelles les mesures sont faites, sont tirées d'une application de vidéo-surveillance d'un parking avec un homme et une voiture le traversant. Ces images ont les caractéristiques suivantes :

- Images en niveau de gris (256 niveaux de gris).
- Nombre de lignes : 235
- Nombre de colonnes : 155

a . Addition de N images

Théoriquement, le temps d'exécution de cette tâche dépend du nombre N des images à additionner, donc il dépend en fin de compte de la nature des images à traiter parce que le nombre N est déjà choisi en tenant compte de deux critères :

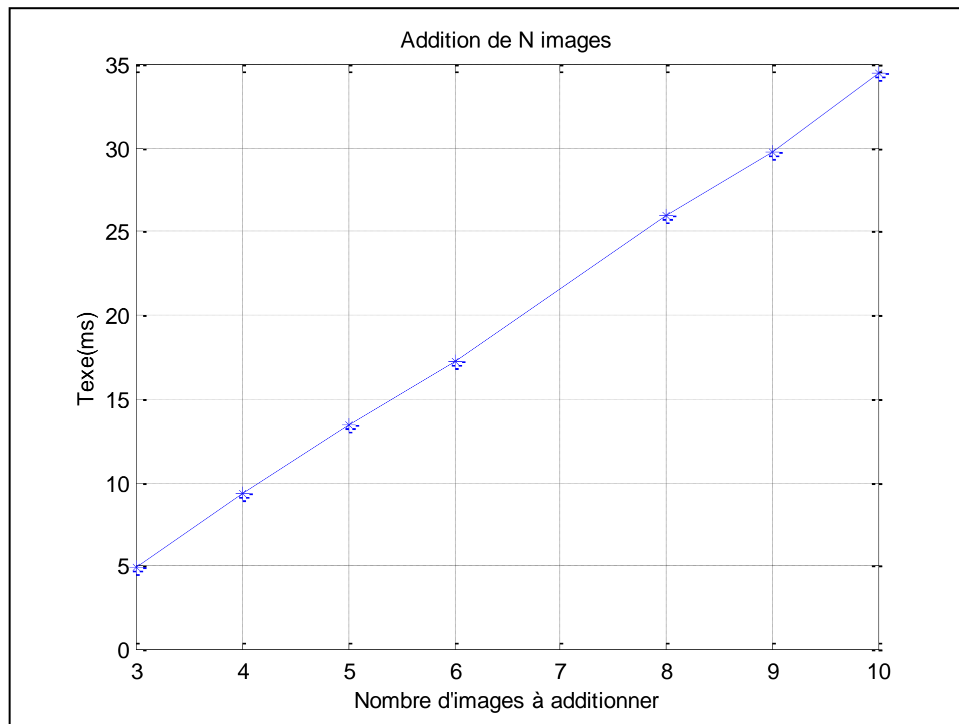
- Le bruit dans les images : Plus le bruit est grand plus N est grand.
- La vitesse des objets : Plus la vitesse est faible plus N est grand .

En conséquence, il sera intéressant d'établir des mesures du temps d'exécution de l'opération d'addition des images en fonction du nombre N :

☞ **Tableau des mesures**

N	*3*		*4*		*5*		*6*		*8*		*9*		*10*	
Nbr de répétition	*20*	*100*	*30*	*50*	*15*	*60*	*20*	*30*	*20*	*30*	*20*	*30*	*20*	*30*
Temps d'exécution (ms)	***99.1***	***480.8***	***281.2***	***470.4***	***200.2***	***810***	***341.3***	***521.5***	***530.5***	***760.7***	***591.6***	***900.9***	***700***	***1022***
T d'exécution / image (ms)	***4.955***	***4.8***	***9.37***	***9.4***	***13.34***	***13.5***	***17.06***	***17.38***	***26.52***	***25.35***	***29.58***	***30.03***	***35***	***34.06***
T d'exécution / image/ N (ms)	*4.8775*		*9.385*		*13.42*		*17.2235*		*25.935*		*29.805*		*34.53*	

☞ **Courbe**



☞ **Interprétation des résultats**

On remarque bien que le temps d'exécution augmente avec le nombre des images à additionner. En principe, la relation entre le temps d'exécution et le nombre des images doit être linéaire car la charge de calcul ne dépend que du nombre des images à additionner ; les essais pratiques ne donnent pas tout à fait une droite car les mesures sont entachés des plusieurs bruits. En effet le temps d'exécution d'une fonction lancée sur un processeur d'un ordinateur dépend d'une part :

- ✓ De l'état de la mémoire cache au moment de lancement de l'application : à l'exécution le processeur cherche les données dans la mémoire cache, s'il les trouves pas il les cherches dans la RAM sinon sur le disque dur (ou autre mémoire de masse), et nous savons que le temps d'accès mémoire diffère d'un type de mémoire à un autre (par ordre croissant : cache, RAM, mémoire de masse).
- ✓ D'autre part de nombre de processus que le processeur les exécutent en parallèle avec la fonction à mesurer tels que les programmes du système d'exploitation (Windows ou autre) qui se lancent automatiquement et périodiquement…

Donc nous ne pouvons pas toujours garantir les mêmes conditions pour tous les essais pour avoir des résultats parfaits.

Nous pouvons conclure , à partir de ces mesures que l'utilisateur de cette application doit prendre une valeur de N (nombre des images à moyenner) le plus petit possible tout en tenant compte bien sûr des deux critères cités ci-dessus.

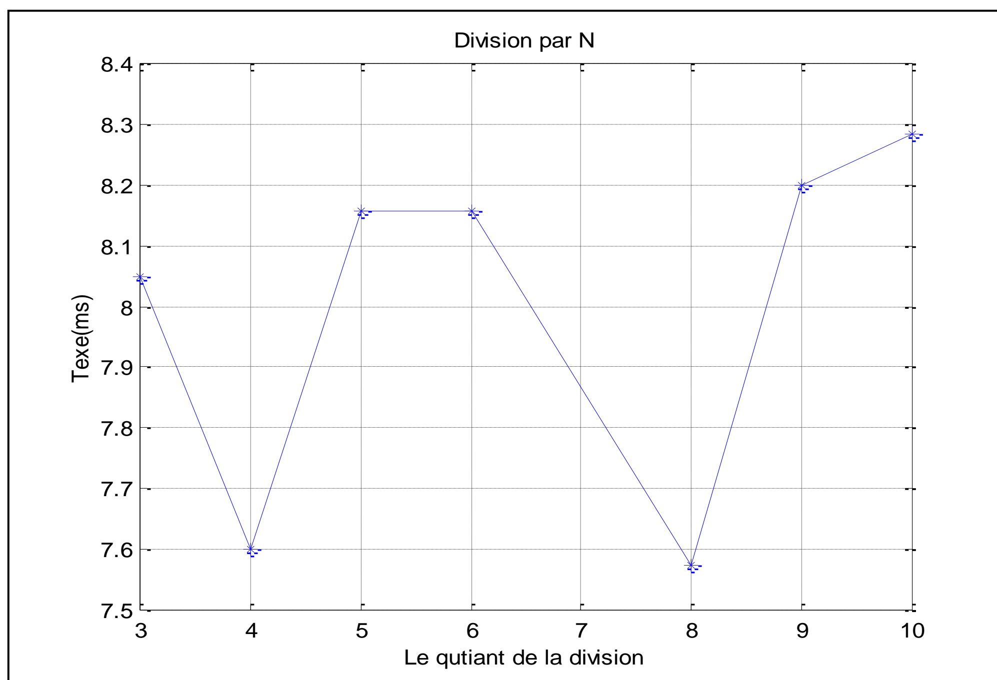
b. Division par N

Nous divisons le niveau de gris de chaque pixel par N, ainsi le nombre des opérations de division est égale au nombre des pixels de l'image qui est : 36425. En conséquence, le temps d'exécution de cette opération doit dépendre de la taille de l'image à traiter, aussi peut être de la parité du nombre N… Vue que nous travaillons avec des images qui ont la même taille (une caméra donne des images de la même taille) il ne reste alors que de vérifier la variation ou non du temps d'exécution en fonction de la nature de N …

☞ **Tableau des mesures**

N	*3*		*4*		*5*		*6*		*8*		*9*		*10*	
Nbr de répétition	*20*	*30*	*20*	*30*	*20*	*30*	*20*	*30*	*20*	*30*	*20*	*30*	*20*	*30*
Temps d'exécution (ms)	***161.1***	***241.2***	***150.5***	***230.2***	***160.1***	***249.2***	***160.1***	***249.2***	***149.5***	***230.2***	***161.1***	***250.2***	***171.1***	***240.2***
T d'exécution / image (ms)	***8.058***	***8.041***	***7.525***	***7.674***	***8.008***	***8.308***	***8.008***	***8.308***	***7.475***	***7.674***	***8.058***	***8.341***	***8.558***	***8.008***
T d'exécution / image/ N (ms)	*8.0495*		*7.5995*		*8.158*		*8.158*		*7.5745*		*8.1995*		*8.283*	

☞ **Courbe**



☞ **Interprétation des résultats**

Nous remarquons bien que pour les nombres pairs nous avons un temps d'exécution plus faible que pour les nombres impairs ; ce qui nous paraît logique parce qu'une division par un nombre puissance de 2 utilise toujours le décalage qui rend le nombre des instructions (assembleur) minime par suite un temps d'exécution plus faible. (critère qu'il faut le tenir en compte lors de choix du nombre N)

c. La soustraction

Cette fonction effectue une différence pixel à pixel entre les deux images. Donc le temps d'exécution de cette fonction ne dépend que de la taille de l'image à traiter. Puisque la taille des images que nous traitons est fixe, alors le temps d'exécution est le même pour tous les images comme le montre le tableau des valeurs suivant qui est issu des mesures effectuées sur différentes images avec :

- Image 1 : image de fond + carreau blanc
- *Image 2 : image de fond + carreau blanc + piéton*
- *Image 3 : image de fond + piéton*
- *Image 4 : image de fond + piéton + voiture.*

Image 1 **Image 2** **Image 3** **Image 4**

☞ **Tableau des mesures**

N° d'image	*Image 1*			*Image 2*			*Image 3*			*Image 4*		
Nbre de répétition	***100***	***200***	***350***	***100***	***200***	***350***	***100***	***200***	***350***	***100***	***200***	***350***
Temps d'exécution (ms)	***1282.2***	***2593.5***	***4536.5***	***1301.3***	***2613.6***	***4797.8***	***1281.2***	***2574.5***	***4606.6***	***1281.2***	***2602.6***	***4745.7***
T d'exécution / répétition (ms)	***12.822***	***12.967***	***12.961***	***13.013***	***13.068***	***13.708***	***12.812***	***12.872***	***13.161***	***12.812***	***13.013***	***13.559***
T d'exécution / répétition / image (ms)	***12.916***			***13.263***			***12.9483***			***13.128***		

☞ **Interprétation des résultats**

Remarque : pour la fonction valeur absolue de l'image nous aurons le même résultat parce que cette fonction traite aussi l'image pixel par pixel donc le temps d'exécution ne dépend que de la taille de l'image traitée. C'est pour cela que dans les mesures effectuées dans le tableau au dessus nous avons mesuré le temps d'exécution de la soustraction avec la valeur absolue.

Comme nous avons déjà expliqué, nous remarquons bien que le temps d'exécution de l'opération (soustraction + valeur absolue) est constant quel que soit le contenue de l'image traitée.

d . La fonction de seuillage

Pour cette fonction nous avons le choix entre un seuillage adaptatif ou un seuillage avec choix du seuil. Dans le cas du seuillage adaptatif, une fonction qui cherche le seuil adéquat (voir paragraphe **II.3.4 c)** dans le chapitre 2) a été réalisée pour lancer automatiquement l'intégralité de l'application « détection de mouvement sur un fond d'image fixe ». Et dans le cas du seuillage avec choix du seuil, c'est l'utilisateur qui choisisse le seuil voulu (c'est deuxième cas ne concerne que les images tests).

Pour le seuillage avec possibilité de choisir le seuil, théoriquement le temps d'exécution ne doit dépendre que de la taille de l'image en question. Il est à noter ici que le temps d'exécution est très faible d'une manière que nous sommes obligés à répéter la même opération plusieurs fois puis diviser la valeur trouvée par le nombre de répétitions.

Nous avons effectué des mesures pour deux images différentes : l'image n°434 qui contient uniquement le piéton et l'image n°278 qui contient le piéton avec la voiture. Dans ces mesures nous avons choisi **128** comme seuil ensuite la valeur **80** (ces valeurs sont prises au hasard : l'essentiel qu'elles soient différentes).

Image N°434

Image N°278

☞ **Tableaux des mesures**

Image n°434

Nbre des fois	*10*	*15*	*20*	*100*	*500*	*1000*
*T,d'exé (ms) seuil **128***	*40.04*	*60.06*	*80.07*	*380.3*	*2052.05*	*4145.15*
Moy/image (ms)	*4.004*	*4.004*	*4.0035*	*3.803*	*4.1041*	*4.14515*
*T,d'exé (ms) seuil **80***	*40.04*	*51.05*	*70.07*	*381.381*	*1961.96*	*4126.13*
Moy/image (ms)	*4.004*	*3.403*	*3.5035*	*3.8138*	*3.9239*	*4.12613*

Le temps d'exécution de l'opération seuillage avec le seuil 128 pour cette image (image n°434) sera alors : **4.010625 ms** et pour le seuil 80 sera **3.79572 ms.**

Image n°278

Nbre des fois	*10*	*15*	*20*	*100*	*500*	*1000*
*T,d'exé (ms) seuil **128***	*41.04*	*50.05*	*80.081*	*390.39*	*1992.99*	*4266.27*
Moy/image (ms)	*4.104*	*3.336*	*4.00405*	*3.9039*	*3.98598*	*4.26627*
*T,d'exé (ms) seuil **80***	*40.04*	*60.06*	*70.07*	*380.38*	*2023.02*	*4105.11*
Moy/image (ms)	*4.004*	*4.004*	*3.5035*	*3.8038*	*4.04604*	*4.10511*

Le temps d'exécution de l'opération seuillage avec le seuil 128 pour cette image (image n°278) sera alors : **3.933366 ms** et pour le seuil 80 sera **3.911075 ms .**

☞ **Interprétation des résultats**

Les résultats de mesures confirment bien que l'opération de seuillage a un temps d'exécution fixe quelque soit l'image en question et quelque soit le seuil choisi.

Pour le seuillage adaptatif, il s'agit de chercher le seuil adéquat par traçage de l'histogramme de l'image à l'aide de la fonction « **ic_getHistogram** », ensuite calculer le gradient de cet histogramme à l'aide de la fonction « **ic_convolveTabHisto** », puis utiliser la fonction « **ic_histoTreshold** » pour trouver le seuil correspondant et enfin seuiller l'image, avec le seuil trouvé, en utilisant la fonction « **ic_thresholdAdapt** ». Le tableau suivant résume les différentes fonctions utilisées dans l'étape seuillage adaptatif et leurs propriétés :

	paramètres	*Type de calcul*	*résultats*
ic_getHistogram	*Image ,* *tableau : pTabHistoNg*	*Balayage du tableau (0 256) + balayage de l'image (0, size)*	*Tableau qui contient les valeurs de l'histogram : pTabHistoNg*
ic_convolveTabHisto	*pTabHistoNg,* *pTabHistoNgDeriv,* **core : noyau du gradient* *sizecore = 5*	▪ *Construction de 3 tableaux qui ont la taille du noyau* ▪ *Calcul du gradient : 256 opérations*	*Tableau : pTabHistoNgDeriv*
ic_histoTreshold	*pTabHistoNgDeriv,* *Uchar : seuilDeriv = 1*	*Balayage du tableau pTabHistoNgDeriv jusqu'à trouver le seuil : seuilDeriv*	*Seuil (Uchar i)*
ic_thresholdAdapt	*Image ,* *Seuil .*	*Suivant le type de l'image , balayage de tous les pixels.(size)*	*Image seuillée.*

D'après le tableau précédent nous pouvons remarquer que le temps d'exécution de toutes les fonctions citées ne dépend que de la taille de l'image (size) ou la taille du noyau sauf pour la fonction « ic_histoTreshold » qui permet de chercher le seuil :
le temps d'exécution de cette fonction dépend de la nature des objets en mouvement qui existent dans l'image ; car nous pouvons trouver un seuil de 30, par exemple, qui correspond bien pour une image mais pas pour une autre qui nécessite un seuil de 50…. Donc la taille du boucle de cette fonction dépend de la valeur du seuil trouvé .
Dans la suite nous allons présenter les mesures effectuées sur différentes images avec :

- ✓ Image 1 : fond noir (pas d'objets)
- ✓ Image 2 : piéton + carreau blanc
- ✓ Image 3 : piéton seul
- ✓ Image 4 : piéton + voiture
- ✓ Image 5 : piéton + voiture (qui forment un seul objet).

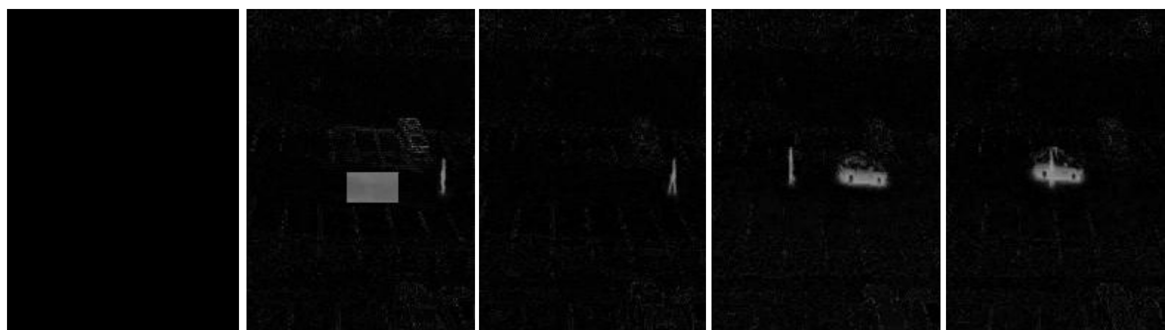


Image 1 Image 2 Image 3 Image 4 Image 5

☞ **Tableau des mesures**

Images	*Image 1*				*Image 2*			
Nbre de répétition	*20*	*30*	*50*	*589*	*20*	*30*	*50*	*589*
Temps d'exécution (ms)	*70.07*	100.1	171.17	2153.1	70.07	110.11	170.17	2173.17
T d'exécution / répétition (ms)	*3.5035*	*3.3366*	*3.4234*	*3.6555*	*3.5035*	*3.6703*	*3.4034*	*3.68959*
T d'exécution / répétition / image (ms)	***3.47975***				***3.56669***			

Image 3				*Image 4*				*Image 5*			
20	*30*	*40*	*589*	*80*	*100*	*140*	*589*	*100*	*589*	*986*	*1000*
70.07	*110.11*	*140.14*	*2153.1*	*311.3*	*350.35*	*490.5*	*2192.1*	*360.36*	*2443.4*	*3734.7*	*3884.8*
3.5035	*3.6703*	*3.5035*	*3.6555*	*3.891*	*3.503*	*3.5035*	*3.7217*	*3.6036*		*3.7877*	*3.8848*
3.5832				***3.6548***				***3.7587***			

☞ **Interprétation des résultats**

D'une part nous remarquons qu'il y a une légère différence entre les différentes temps d'exécution ce qui implique que la longueur du boucle qui cherche le seuil diffère légèrement entre les différentes images.

D'autre part, ces résultats nous semble être logique parce qu'on trouve que la valeur la plus petite est celle pour l'image où il n'y a qu'un fond noir, et la valeur la plus grande est celle pour le piéton avec la voiture (plusieurs objets).

Mais de toute façon , et à partir de ces résultats, nous ne pouvons pas dire que le temps d'exécution de la fonction « seuillage adaptatif » dépend tellement du contenu de l'image traitée, au contraire nous pouvons le considérer constant pour tous les images (qui ont évidemment la même taille).

e . les traitements morphologiques

e.1 L'Erosion :

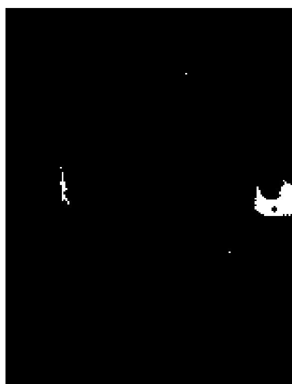
En revenant au code de cette opération, nous remarquons que l'analyse s'effectue sur tous les pixels de l'image :

- Pour les pixels blancs : ces pixels prennent le niveau de gris noir si et seulement si un voisin est noir.
- Pour les pixels noirs : ils restent noirs dans tous les cas.

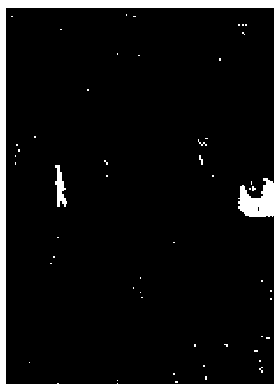
Le traitement se fait alors sur tous les pixels de l'image (noirs et blancs). Ceci veut dire que le temps d'exécution ne dépend pas de la taille des objets (ni du contenu de l'image traitée) mais il dépend plutôt de la taille de l'image uniquement.

Des essais sont effectués sur plusieurs images en essayant de changer le contenu de l'image en proportion des pixels blancs. Pour chaque image nous avons répété les mesures au minimum trois fois pour s'assurer de l'exactitude des résultats.

Les images choisies sont les suivantes :



4.91355
4.90974
pixel B : **314** = 0.862%

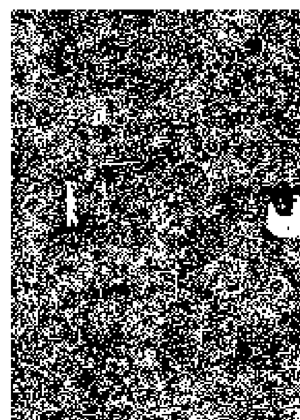


4.9351
4.93952
pixel B : **556** = 1.52%

5.03403
5.04484
5.04324
5.05327
pixel B : **1737** = 4.76%

5.32432
5.28892
5.30746
pixel B : **4449** = 12.21%

5.67149
5.67883
pixel B : **8007** = 21.98%



6.21674
6.23077
6.23698
26.35%
pixel B : **13160** = 36.12%

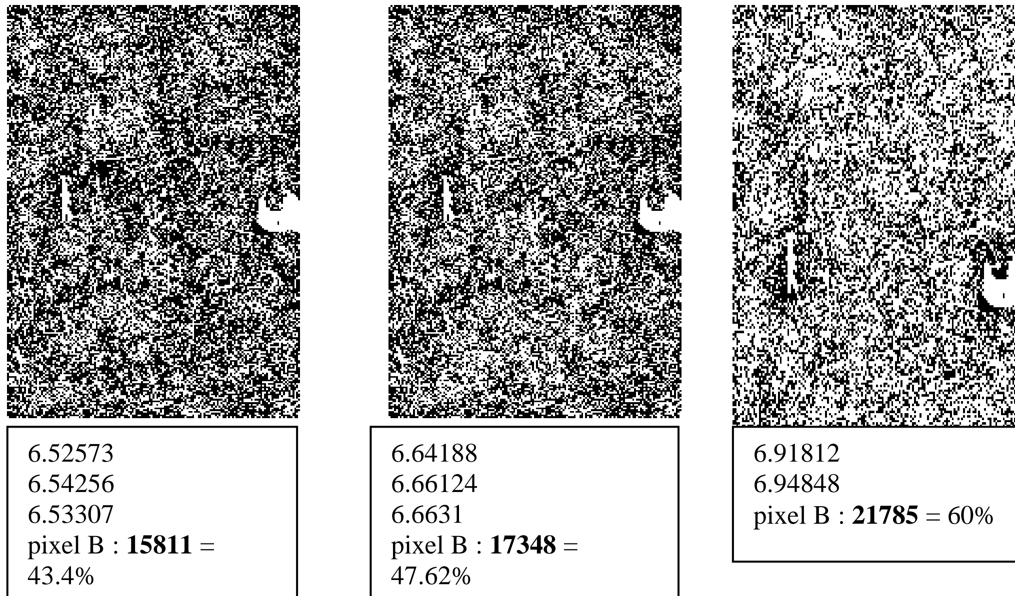
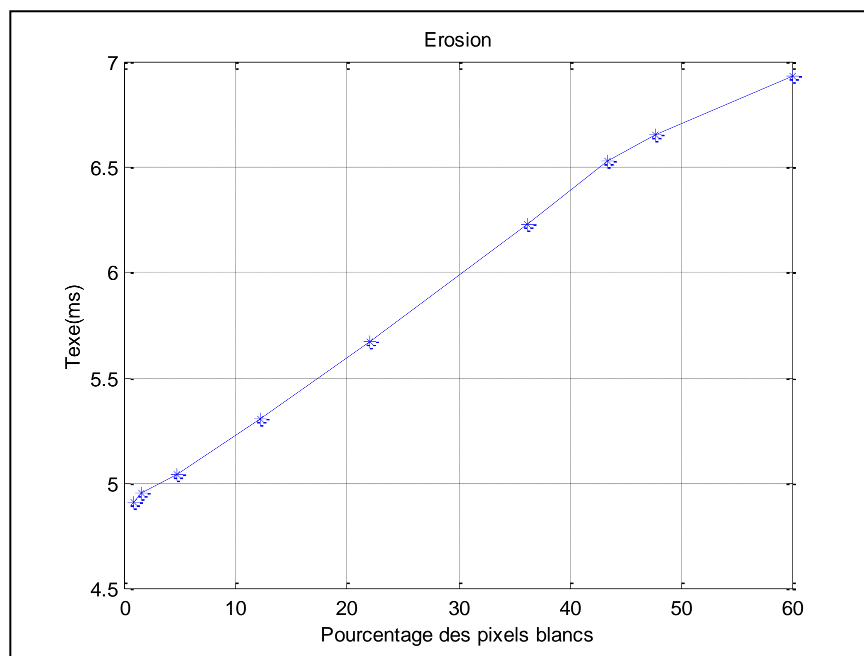


Figure 3.1 Images test de l'érosion

☞ **Tableau de mesures**

Pixel Blanc (%)	0.862	1.52	4.76	12.21	21.98	36.12	43.4	47.62	60
Temps (ms)	4.911645	4.95312	5.043845	5.30662	5.67516	6.228163	6.534145	6.6554	6.9333

☞ **Courbe**



☞ **Interprétation des résultats**

La courbe obtenue montre qu'il y a une légère variation du temps d'exécution en fonction du pourcentage des pixels blancs. Cette variation est due au système de prédiction du branchement du processeur (figure 3.2) qui tombe toujours en erreur à cause de la distribution des pixels blancs qui est presque uniforme sur toute la surface de l'image.

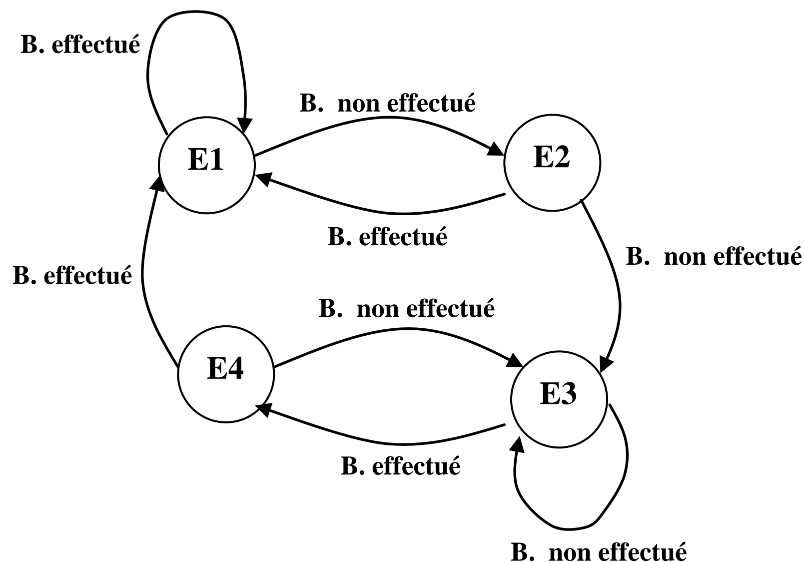


Figure 3.2 Principe de prédiction de branchement à 4 états d'un processeur

En fait, le système de prédiction de branchement à 4 états d'un RISC consiste à prédire à chaque cycle une instruction de branchement (dans le cas où le programme contient des branchements) et si le branchement n'est pas effectué il passe à un autre état : prenons par exemple le cas de la figure 3.2 où le processeur met l'instruction I2 dans le pipeline si le branchement est effectué, et l'instruction I1 dans le cas contraire. Le processeur est à l'état E1, il prédit un branchement et il le vérifie ; si le branchement est effectué (c-à-d c'est bien I2 qui doit le chercher) il reste toujours dans cette état et si le branchement n'est pas effectué (c-à-d qu'il a commis une erreur et il doit chercher I1 et le mettre dans le pipeline), il passe à l'état E2 où il cherche I1. Ensuite, il prédit un branchement en mettant I2 dans le pipeline, si le branchement est effectué il revient à l'état E1, sinon il passe à l'état E3 (c-à-d qu'il a commis encore une erreur : il a cherché I2 au lieu de I1). Dans l'état E3 il cherche I1, si le branchement n'est pas effectué il reste dans cette état ; sinon, il passe à l'état E4 où il va chercher I2.

Dans l'état E4, il prédit un branchement (c-à-d qu'il prévoit I2 dans le prochain cycle du pipeline), si le branchement n'est pas effectué, il revient à l'état E3, sinon il passe à l'état E1 et ainsi de suite…

Maintenant si nous prenons le cas de l'érosion, nous remarquons qu'il y a un branchement pour faire le traitement nécessaire si le pixel est blanc ; et un autre traitement (c'est garder le même niveau de gris) si le pixel est noir. En balayant l'image, le processeur prédit toujours que le pixel suivant est blanc, et quand il s'aperçoit que le pixel traité est noir il doit corriger ce qu'il a mis dans le pipeline ; c'est cette correction qui prend plus du temps, et rend le temps d'exécution total variable en fonction de la distribution des pixels blancs et noirs dans l'image.

Les images tests sont mal choisies de telle façon qu'après chaque pixel blanc nous trouvons un pixel noir ; ce qui rend la prédiction du branchement une perte du temps au lieu qu'elle minimise le temps d'exécution.

Nous trouvons le maximum du temps d'exécution vers la moitié en pourcentage des pixels blancs car c'est là où les erreurs de prédiction augmentent.

Si nous avons séparé l'image en deux parties, une pour les pixels blancs et l'autre pour les pixels noirs et en faisant changer à chaque fois le pourcentage des pixels blancs nous allons minimiser les erreurs de prédiction de branchement de telle façon que nous allons obtenir un temps d'exécution constant pour les différents pourcentages (une expérience test a prouvé cette proposition).

e.2 La dilatation

Le principe de la dilatation ressemble à celle de l'érosion : le traitement se fait sur tous les pixels de l'image quelque soit le contenu. Un pixel noir devient blanc si et seulement si un de ses voisin est blanc ; et les pixels blancs restent blancs pour tous les cas.

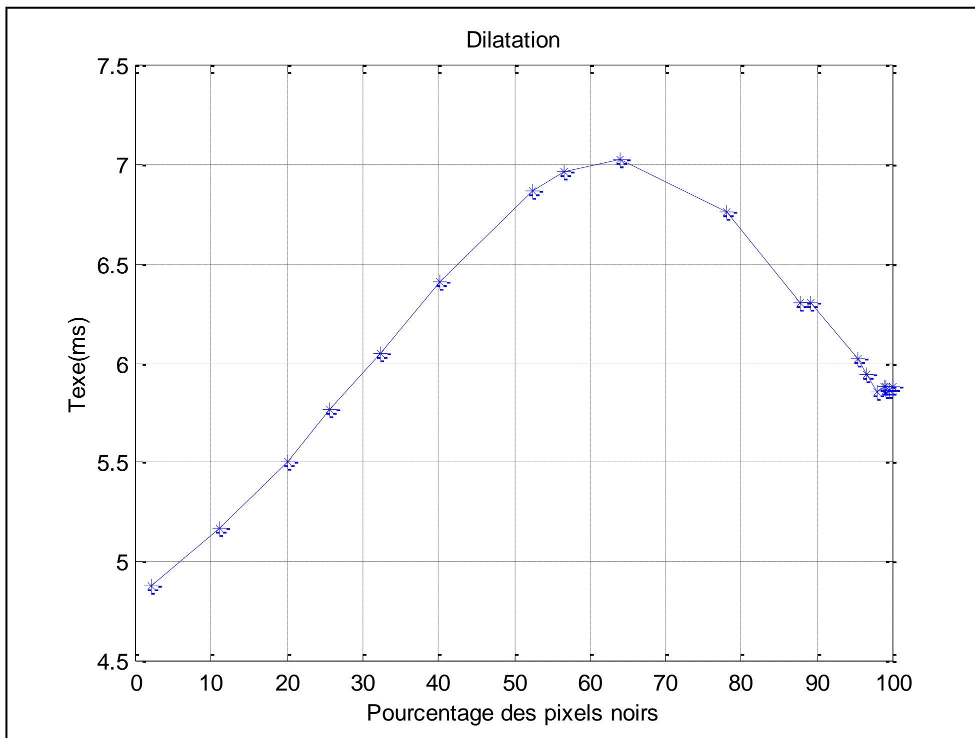
Nous avons effectué les mesures sur les images test de l'érosion et leurs images résultats.

☞ **Tableau des mesures**

pixels noir en %	**2,13**	**11,19**	**20,19**	**25,62**	**32,37**	**40,19**	**52,37**	**56,59**	**63,87**	**78,01**
Texe (ms)	***4,90138***	***5,16727***	**5,50661**	**5,77047**	**6,0482**	**6,40489**	**6,87005**	**6,96084**	**7,01575**	**6,75754**

pixels noir en %	**87,78**	**89,1**	**95,23**	**95,34**	**96,59**	**97,81**	**98,81**	**99,47**	**100**
Texe (ms)	**6,30307**	**6,30709**	**6,02799**	**6,01782**	**5,94304**	**5,8544**	**5,87863**	**5,86378**	**5,87831**

☞ **Courbe**



☞ **Interprétation des résultats**

Pratiquement nous trouvons la même allure que pour l'érosion. Ceci est dû toujours au prédiction du branchement que nous avons déjà expliqué pour le cas de l'érosion.
C'est que nous pouvons conclure à partir de ces deux courbes (érosion et dilatation), c'est que ces deux tâches ont un temps d'exécution fixe et ne dépend d'aucun paramètres sauf la taille de l'image en question.

e.3 L'ouverture par reconstruction

Cette fonction fait le balayage de tous les pixels de l'image, le traitement concerne uniquement les pixels blancs : pour chaque pixel blanc nous recopions ses voisins à partir de l'image qui n'a pas encore subi les traitements morphologiques (érosion et dilatation). Les pixels blancs forment les objets en déplacement qu'il faut les détecter, donc nous avons intérêt à avoir les formes complètes de ces objets, ainsi nous faisons recours à cette ouverture par reconstruction qui consiste à éliminer les effets indésirables de l'érosion ou de la dilatation. Puisque la reconstruction touche uniquement les pixels blancs, le temps d'exécution de cette fonction doit être fortement corrélé au pourcentage des pixels blancs dans l'image. La procédure correspondante à cette fonction, avant de réaliser la reconstruction, passe par l'érosion et la dilatation ; donc le pourcentage de pixels blancs indiqué dans nos mesures correspond bien à l'image après érosion et dilatation. Nous avons effectué des

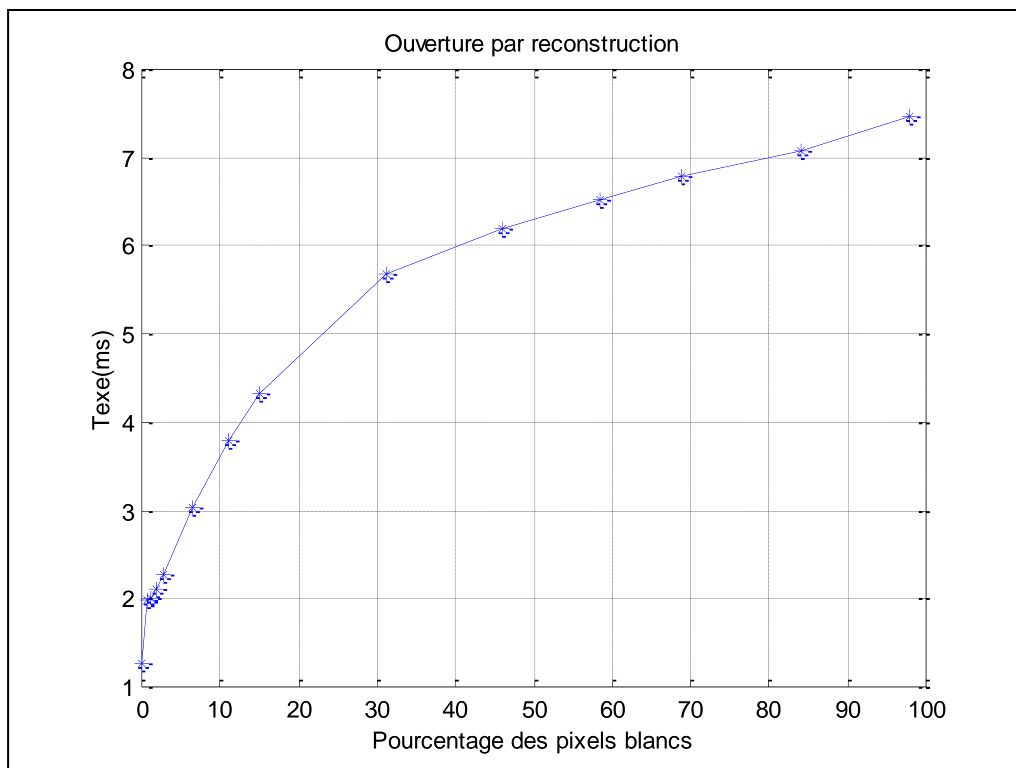
mesures sur plusieurs images pour différents pourcentages des pixels blancs. Le tableau suivant récapitule nos essais :

☞ **Tableau des mesures**

pixels blancs en %	**0**	**0,76**	**1,18**	**1,21**	**1,75**	**2,68**	**6,58**	**11,13**	**15,02**	**31,09**
Texe (ms)	**1,26465**	**1,98567**	**2,00155**	**2,00772**	**2,10587**	**2,26543**	**3,03218**	**3,79628**	**4,31937**	**5,68867**

pixels blancs en %	**45,91**	**58,42**	**68,75**	**84,02**	**97,85**
Texe (ms)	**6,18722**	**6,52855**	**6,79226**	**7,08502**	**7,46023**

☞ **Courbe**



☞ **Interprétation des résultats**

La courbe obtenue montre bien la corrélation entre le temps d'exécution de cette tâche (ouverture par reconstruction) et le pourcentage des pixels blancs.

Les résultats expérimentaux sont en faveur avec l'étude théorique et cette tâche a un temps d'exécution variable en fonction du jeu de donnée en question.

f . L'étiquetage des objets

L'étiquetage des objets dans l'image, est fait selon quatre phases : tout d'abord nous commençons par le premier balayage de l'image qui consiste à affecter des niveaux de gris aux pixels blancs des différents objets tout en construisant un tableau des équivalences entre les pixels. Ensuite nous faisons une actualisation de ce tableau des équivalences pour passer au deuxième balayage de l'image qui consiste à actualiser l'image en fonction des étiquettes. Enfin nous passons à la dernière phase qui est le calcul du nombre d'étiquettes dans l'image.

Théoriquement le temps d'exécution de la première phase dépend du nombre des pixels blancs à traiter (nous utilisons plutôt le pourcentage des pixels blancs). Le temps d'exécution de la phase d'actualisation du tableau d'équivalences ne dépend que de la taille de l'image vue que la taille de ce tableau est celui la taille de l'image. Donc nous pouvons considérer que la deuxième phase a un temps d'exécution fixe.

Pour le deuxième balayage de l'image, son temps de calcul dépend uniquement du pourcentage des pixels blancs (vue que le traitement concerne ce type de pixels uniquement).

Pour la dernière phase, en principe le temps d'exécution va dépendre du nombre d'étiquettes dans l'image, mais puisque ce temps (incrémenter un compteur) est très faible nous allons le négliger.

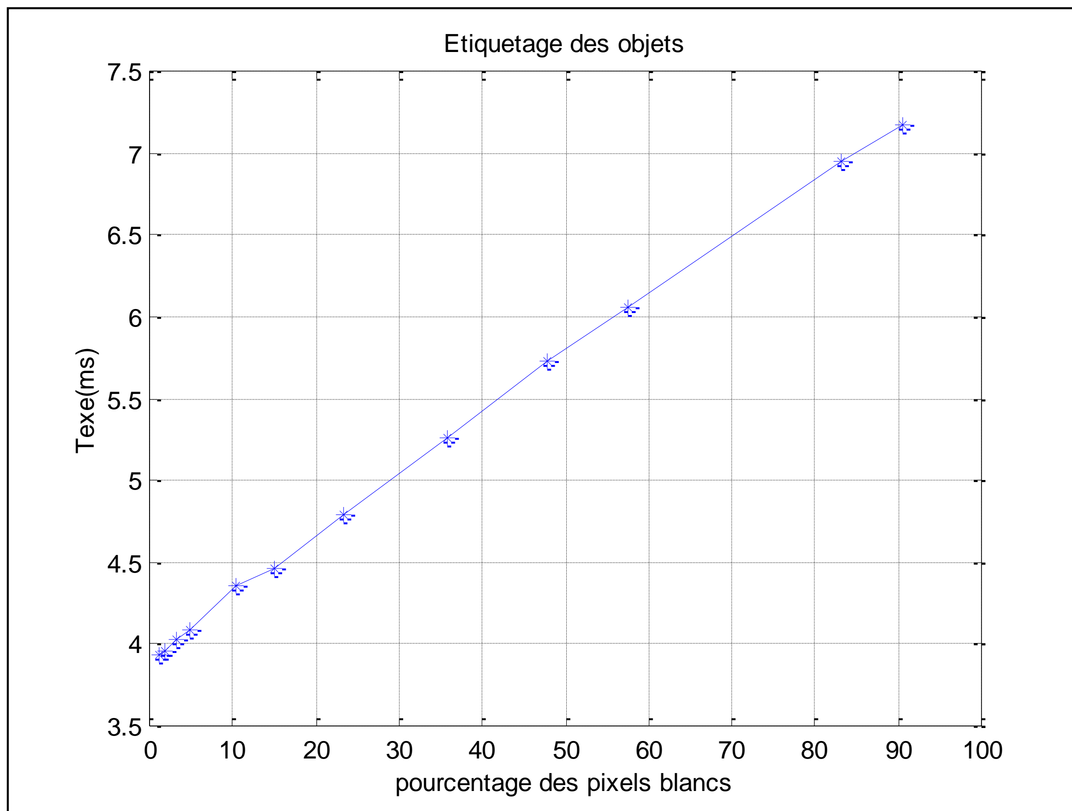
En conclusion, l'étiquetage des objets ne dépend, théoriquement, que du nombre de pixels blancs dans l'image traitée (c-à-d il ne dépend que de la taille totale des objets à étiqueter).

☞ **Tableau des mesures**

pixels blancs en %	**1,18**	**3,28**	**4,79**	**10,42**	**15,01**	**23,33**	**35,7**	**47,85**	**57,54**
Texe (ms)	**3,92953**	**4,03063**	**4,08588**	**4,35926**	**4,46346**	**4,79379**	**5,25435**	**5,73133**	**6,06176**

pixels blancs en %	**83,11**	**90,45**
Texe (ms)	**6,95095**	**7,17427**

☞ **Courbe**



☞ **Interprétation des résultats**

Comme il est montré dans la courbe ci-dessus le temps d'exécution de l'étiquetage des objets est fortement corrélé au pourcentage des pixels blancs dans l'image en d'autres termes il dépend de la taille totale des objets à étiqueter. La courbe obtenue peut être assimilée à une droite donc nous pouvons dire que la relation entre le temps d'exécution et le pourcentage des pixels blancs est quasiment linéaire.

g . Construction de l'enveloppe englobante

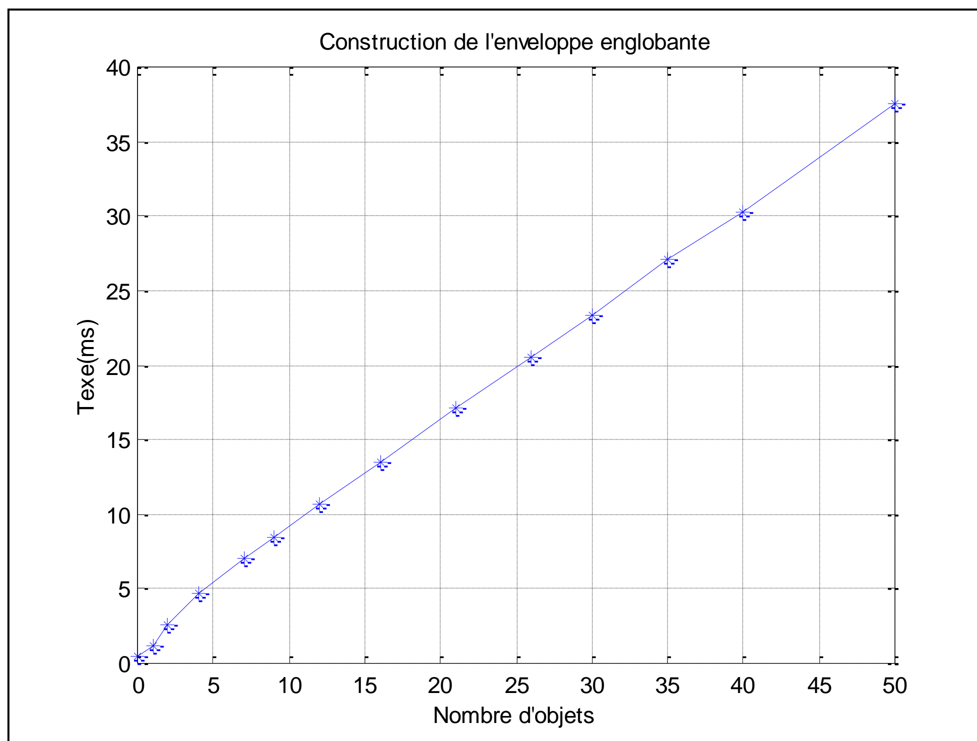
Cette procédure se fait en trois étapes, d'abord la définition des coordonnées des enveloppes, ensuite le calcul du centre de gravité et enfin l'affichage de l'enveloppe englobante. Théoriquement, les temps d'exécution de toutes ces étapes dépendent du nombre des objets (étiquettes) qui existent dans l'image.

Nous allons mesurer le temps d'exécution de cette fonction sur différentes images qui n'ont pas le même nombre des objets.

☞ **Tableau des mesures**

Nombre d'objets	Nombre des pixels blancs	Temps d'exécution (ms)
0	**0**	**0.425526**
1	**350**	**1.12429**
2	**433**	**2.56702**
4	**697**	**4.65169**
7	**1195**	**7.04646**
9	**1747**	**8.41643**
12	**1950**	**10.6992**
16	**2090**	**13.5432**
21	**2370**	**17.1166**
26	**2613**	**20.5806**
30	**2977**	**23.3376**
30	**3416**	**23.4558**
35	**3798**	**27.063**
40	**4107**	**30.2404**
50	**4534**	**37.5743**

☞ **Courbe**



☞ **Interprétation des résultats**

Les résultats expérimentaux sont conformes à l'étude théorique : la tâche construction de l'enveloppe englobante a un temps d'exécution variable, qui dépend du nombre des objets dans l'image en question.

D'après la courbe de temps d'exécution (en millisecondes) en fonction du nombre des objets, nous trouvons une relation qui est linéaire et qui peut être assimilé à une droite.
Remarque : la mesure pour le nombre d'objets égal à 30 est répétée deux fois pour deux nombres de pixels blancs différents. Ceci pour s'assurer que le paramètre de corrélation est bien le nombre d'étiquettes indépendamment de la taille totale des objets.

h . Test sur le déplacement

Théoriquement, cette fonction a un temps d'exécution très faible parce que la nature des instructions employées dans cette tâche fait que le traitement ne se fait pas au niveau pixel ni au niveau objet mais plutôt sous forme des instructions des affectations au niveau structure des coordonnées des objets. Ce type d'instructions ont un temps d'exécution très faible par rapport aux reste des tâches de l'application. Expérimentalement nous trouvons des mesures de l'ordre de micro-secondes. Donc cette fonction est considérée comme une tâche qui a un temps d'exécution fixe.

i . Mise à jour du fond de l'image

Cette fonction a les mêmes caractéristiques que la fonction précédente : expérimentalement nous trouvons des mesures de temps d'exécution de l'ordre de micro-secondes donc cette fonction est aussi considérée comme une tâche à un temps d'exécution fixe.

III.3 Traitement des résultats du *Profiling*

Tout au long de notre étude sur l'application détection de mouvement sur un fond d'image fixe, nous avons cherché toujours à distinguer entre deux types des tâches : celles qui ont un temps d'exécution fixe en fonction de la nature de données mises en jeu et celles qui ont un temps d'exécution variable.
Nous allons nous intéresser plutôt sur ce dernier type de tâches. Parmi les fonctions de l'application étudiée, nous avons pu distinguer trois tâches qui ont des temps d'exécution variables.

En analysant les courbes de ces trois tâches, nous allons diviser chaque courbe sur plusieurs catégories. Pour chaque catégorie nous allons attribuer un temps d'exécution qui sera le même pour toutes les images y appartiennent. Ce temps d'exécution peut être le maximum des temps de l'ensemble des points qui appartiennent à la même catégorie dans le but d'obtenir des exécutions qui satisfassent les contraintes dans le pire des cas. La contrainte sur cette classification est que les seuils définis sur un paramètre de corrélation soient

identiques pour le processeur et le FPGA. Lors de division de chaque courbe nous devons chercher d'une part à avoir le maximum des catégories pour une tâche, à fin de se rapprocher de plus en plus de la valeur exacte du temps d'exécution. D'autre part à respecter les contraintes pour la division qui sont :

- ✓ le temps de conception c'est à dire le temps nécessaire pour faire tourner l'outil de partitionnement (pour une seule combinaison l'algorithme génétique par exemple prend 5 minutes pour trouver une solution qui peut ne pas être pas la meilleure) autant de fois pour toutes les combinaisons trouvées.
- ✓ La limite de l'espace mémoire qui va rassembler tous les contextes correspondants à toutes les combinaisons trouvées. Le choix de catégories dépend aussi de la courbe analysée : nous estimons plus des catégories pour la pente la plus élevée (en comparant toutes les pentes des courbes).

Une solution possible pour diviser ces courbes en plusieurs catégories est celle montrée sur les figures 3.2, 3.3 et 3.4.

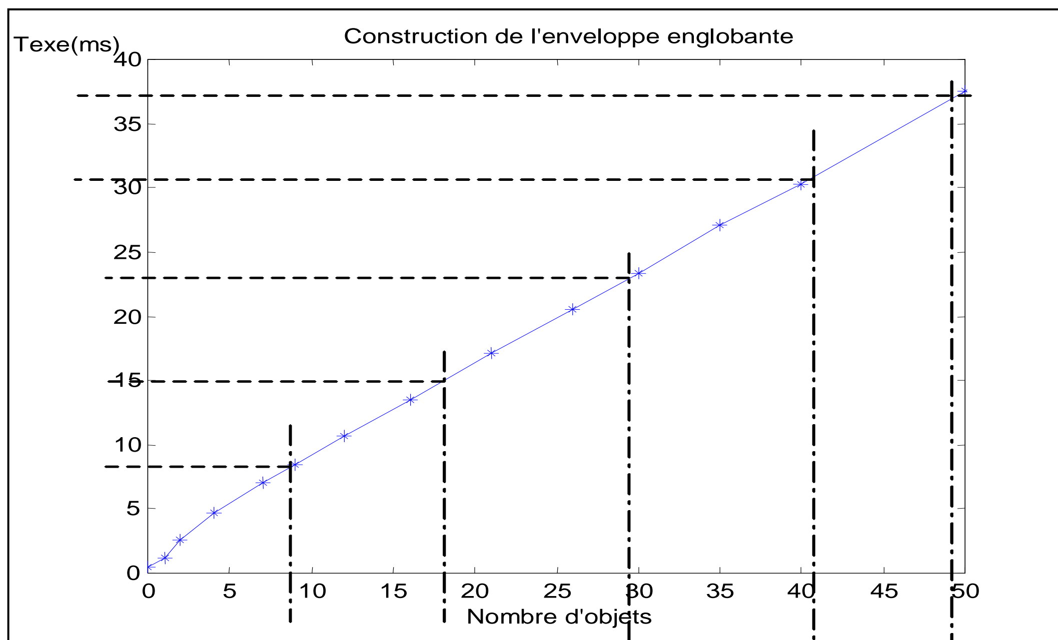


Figure 3.2 traitement de la fonction : construction de l'enveloppe englobante

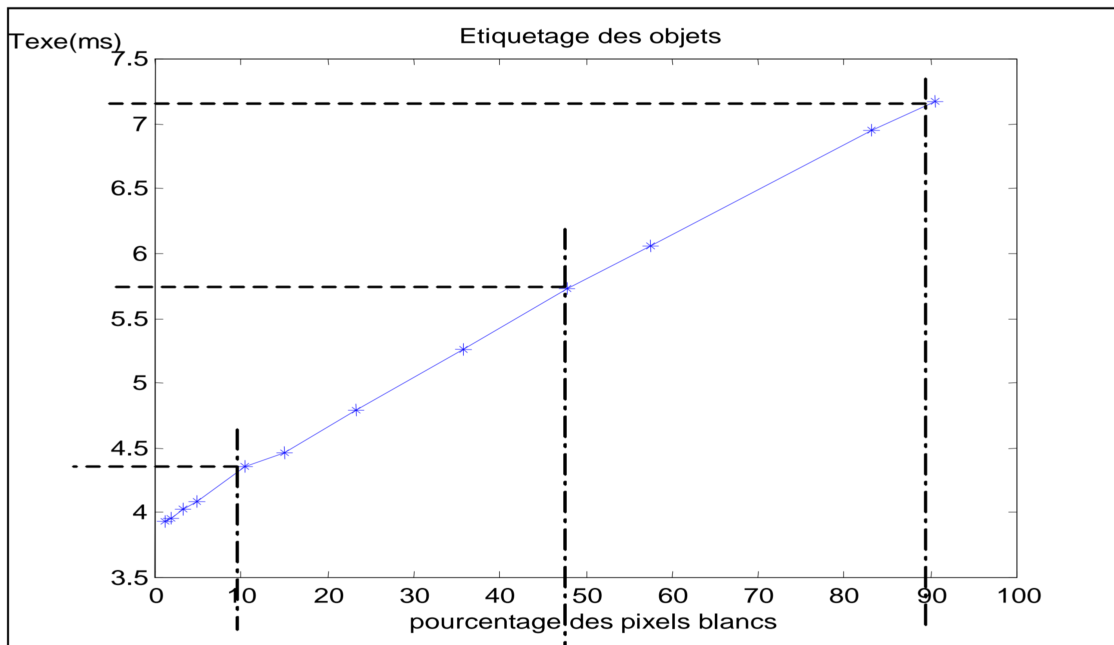


Figure 3.4 Traitement de la fonction : Etiquetage

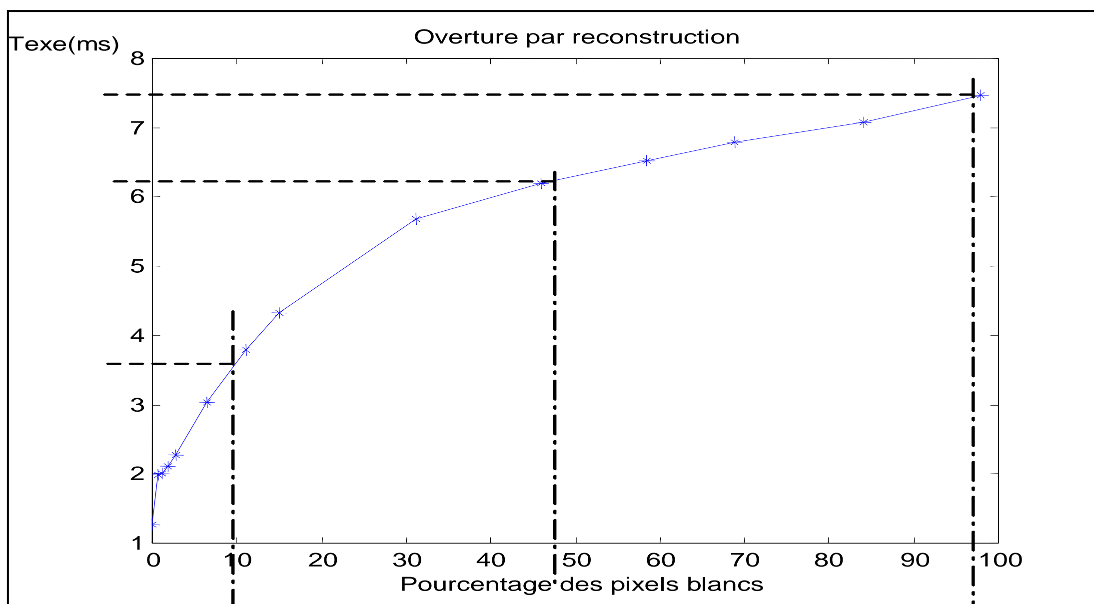


Figure 3.3 Traitement de la fonction : Ouverture par reconstruction

Avec ce choix nous aurons 3 catégories pour la tâche ouverture par reconstruction :

Pour les images qui ont un pourcentage des pixels blancs inférieur à 10% elles auront un temps d'exécution égal à 3.7 ms etc.

Pour la construction de l'enveloppe englobante nous aurons 5 catégories et pour l'étiquetage des objets nous aurons 3 catégories.

Pour les fonctions qui ont le même paramètre de corrélation, nous avons essayé de choisir les mêmes intervalles de catégories pour diminuer le nombre de combinaisons possibles pour l'application entière. Par exemple pour les deux tâches ouverture par reconstruction et étiquetage, nous avons choisi les mêmes intervalles de catégories parce qu'elles dépendent tous les deux du pourcentage de pixels blancs dans l'image.
Avec ce choix de catégories, nous obtenons 15 combinaisons possibles c'est à dire qu'il faut chercher 15 solutions avec un outil de partitionnement (exp : algorithme génétique) pour 15 implantations cibles.

Remarque :
Pour le choix des catégories, nous pouvons affiner notre choix si nous aurons une idée sur la probabilité d'avoir un type d'image à un moment donné. Prenons par exemple le cas où cette application est utilisée dans la surveillance d'un parking, nous pouvons alors opter pour deux modes de fonctionnement : mode pour les heures de pointe (entre 7 :00 et 9 :00) et mode pour les heures où nous n'aurons pas beaucoup des voitures qui bougent. Pour chaque mode nous allons attribuer un choix des catégories.

III.4 Graphe de flots de données conditionné

A ce stade nous avons effectué toutes les mesures du temps d'exécution des tâches tournées sur le processeur. Pour les estimations des implantations sur les parties matérielles nous allons les déduire à partir des résultats des estimations sur le soft (mesures effectuées sur le processeur).
Nous pouvons alors construire un graphe de flots de données conditionné dans lequel chaque tâche, à temps d'exécution variable, sera répliquée autant de fois qu'il y avait de catégories identifiées pour elle (figure 3.4).

A partir de ce graphe de flots de données conditionné, nous pouvons définir l'ensemble des configurations possibles (figure 3.5) qui sont autant de graphes de flots de données non conditionnées [11 art]. Nous rappelons ici que le choix de granularité, lors de spécification d'un modèle pour l'application, a un fort impact sur le nombre de configurations possibles et par la suite il permet d'éviter une explosion combinatoire des combinaisons.

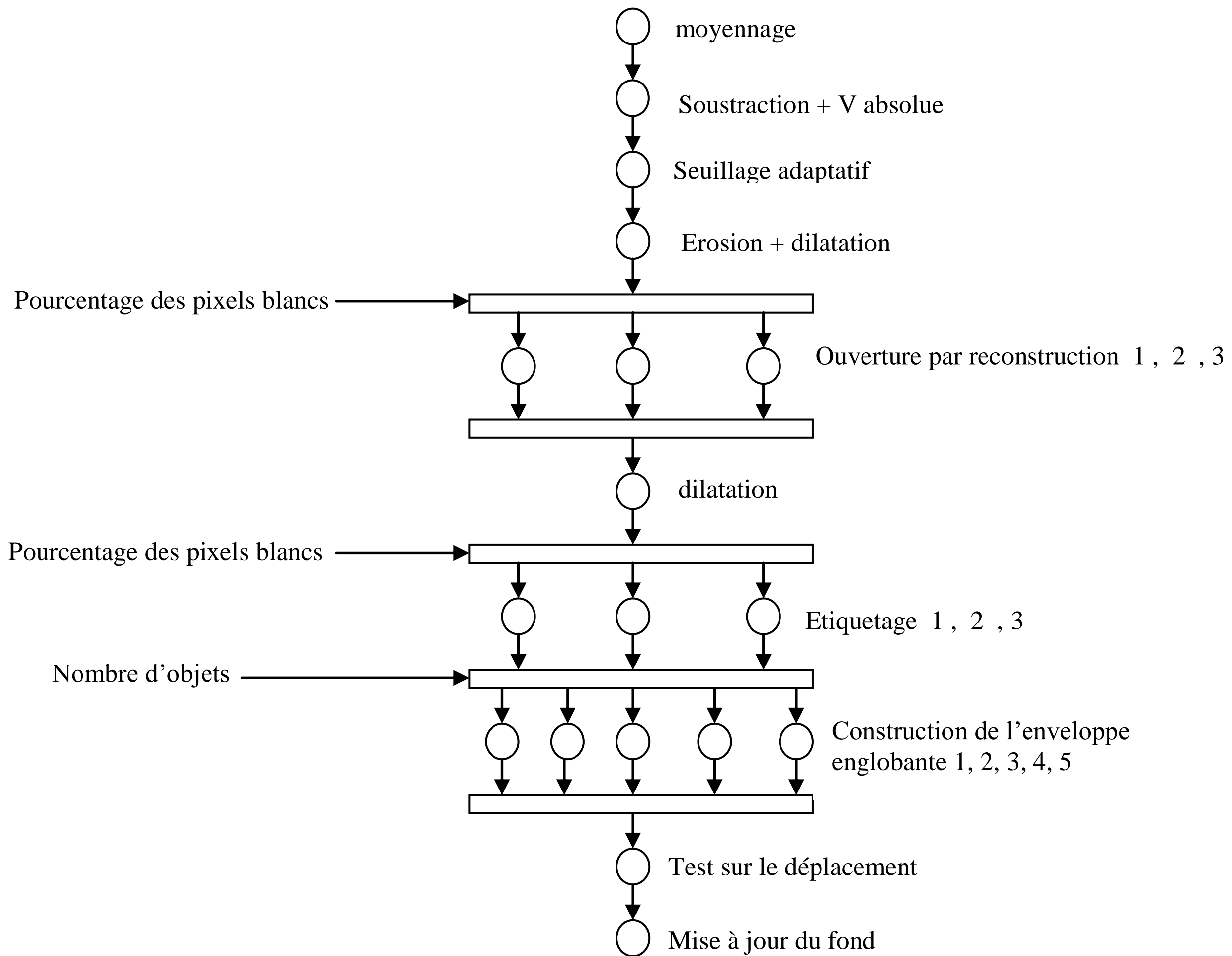


Figure 3.4 DFG avec tâches à temps d'exécution variable

Les mesures effectuées sur le processeur sont rassemblées dans le tableau suivant :

	Texec 1 (ms)	Texec 2 (ms)	Texec 3 (ms)	Texec 4 (ms)	Texec 5 (ms)
Moyennage	12.927				
Soustraction + V absolue	13.0638				
Seuillage adaptatif	3.60862				
Erosion + dilatation	13.96423				
Ouverture par reconstruction	3.625	6.1875	7.678		
Dilatation	41.5587				
Etiquetage	4.34615	5.7538	7.15384		
Construction d'enveloppes	8.21428	15.05	22.8571	30.714285	37.142857
Test sur le déplacement	0.002				
Mise à jour du fond	0.0007				

Une estimation des mesures des temps d'exécution, avec les nombres de CLBs associés, qui seront effectuées sur le FPGA est donnée dans le tableau suivant :

	Texec 1 (ms) Nbr CLBs	Texec 2 (ms) Nbr CLBs	Texec 3 (ms) Nbr CLBs	Texec 4 (ms) Nbr CLBs	Texec 5 (ms) Nbr CLBs
Moyennage	10.3416 4 CLBs				
Soustraction + V absolue	10.45104 3 CLBs				
Seuillage adaptatif	2.886896 1 CLBs				
Erosion + dilatation	11.171384 4 CLBs				
Ouverture par reconstruction	2.9 1 CLBs	4.95 2 CLBs	6.1424 3 CLBs		
Dilatation	33.24696 12 CLBs				
Etiquetage	3.47692 2 CLBs	4.60304 3 CLBs	5.723072 4 CLBs		
Construction d'enveloppes	6.571424 3 CLBs	12.04 6 CLBs	18.28568 9 CLBs	24.571428 12 CLBs	29.7142856 15 CLBs
Test sur le déplacement	0.0016 1 CLBs				
Mise à jour du fond	0.00056 1 CLBs				

Avec ces mesures nous trouvons 15 combinaisons possibles de DFG non conditionnés à faire tourner sur l'algorithme génétique. Un exemple de graphe de flots de données d'une combinaison est donné dans la figure 3.5.

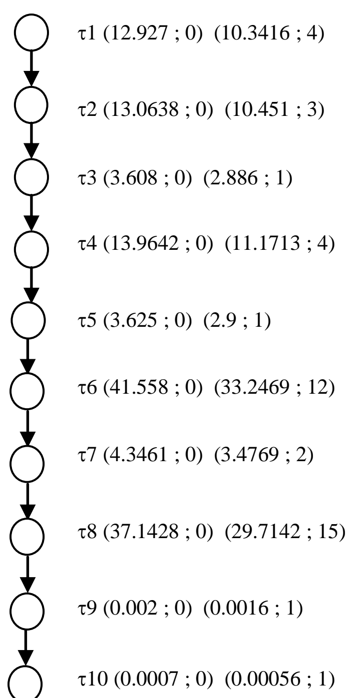


Figure 3.5 DFG non conditionné de la combinaison 13

III.5 Exploration automatique de l'espace de combinaisons

Après avoir transformé le graphe de flots de données d'une application en un graphe de flots de données conditionné, nous avons réfléchi à automatiser la méthode d'extraction des combinaisons possibles. Etant donné un graphe de flots de donnée conditionné quelconque et connaissant les paramètres de corrélation de chaque tâche à temps d'exécution variable, nous devons pouvoir extraire automatiquement toutes les configurations réalisables !

Il nous fallait alors écrire un code en langage évolué (C ,C++….) qui peut lire toutes les données concernant un graphe conditionné quel que soit sa forme et sa taille ; et qui donne en sortie toutes les combinaisons possibles et réalisables.
Une combinaison est dite réalisable si elle ne contient pas des conflits des intervalles du paramètre du contrôle :
Soient les deux courbes suivantes (figure 3.6) qui représentent les temps d'exécution de deux tâches en fonction du même paramètre de corrélation et avec les mêmes intervalles.

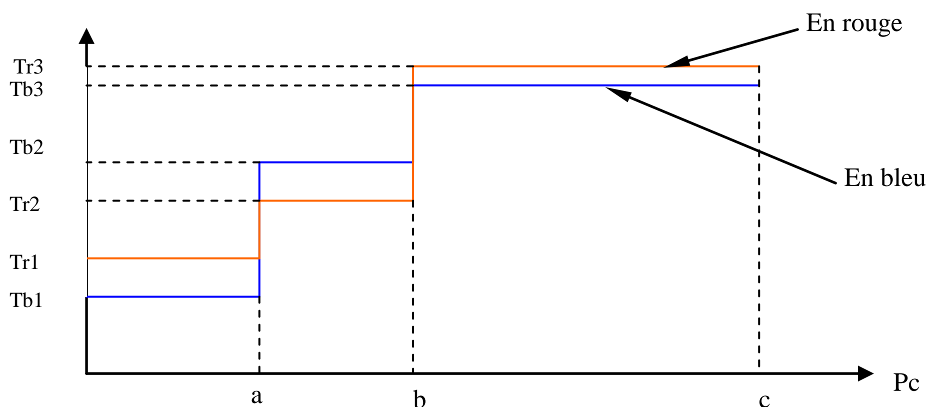


Figure 3.6 temps d'exécution en fonction d'un paramètre de corrélation

Il est clair qu'une solution qui contient Tr3 pour la tâche en rouge et Tb1 pour la tâche en bleu n'est pas réalisable par ce qu'on ne peut pas se situer au même temps, et pour le même paramètre de corrélation, aux intervalles [0 a] et [b c].
Donc il faut chercher les bonnes combinaisons correspondant aux bons intervalles.

Le cas le plus simple est lorsque toutes les tâches à temps d'exécutions variable ont des paramètres de corrélation différentes : dans ce cas toutes les combinaisons possibles sont réalisables et il n'y a aucun risque de conflit.
Dans le cas où il y a deux paramètres de corrélation ou plus qui sont identiques, il faut éliminer les solutions non réalisables.

III.5.1 Structure du programme

Nous avons créé un projet en Visual C++ pour résoudre ce problème. Notre programme est structuré, comme le montre l'organigramme dans la figure 3.7, en trois étapes :

- Lecture d'un graphe conditionné
- Recherche des combinaisons réalisables
- Affichage et sauvegarde des solutions

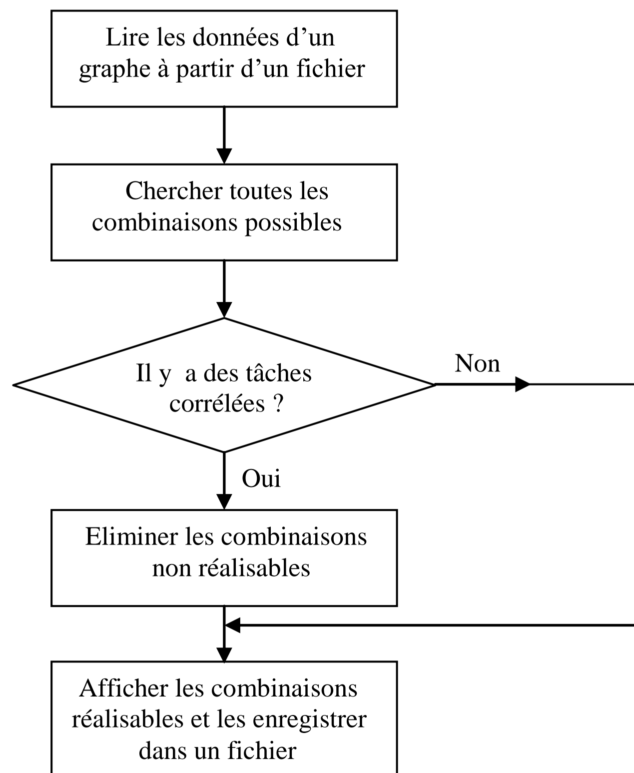


Figure 3.7 Organigramme du processus d'exploration

Pour faire rentrer tous les données concernant un graphe de tâches conditionné nous avons utilisé une classe qui prend en considération le temps d'exécution d'une tâche, son numéro, les numéros de ses successeurs, les valeurs inférieur et supérieur de l'intervalle de travail concerné, le paramètre de corrélation et enfin le niveau de la tâche dans le graphe.

III.5.2 Données d'entrée

Le fichier d'entrée contenant les données d'un graphe conditionné est un fichier texte, il commence par le nombre total de tâches qui se trouvent dans le graphe de flot de données conditionné. Ensuite, pour chaque tâche nous avons 5 lignes qui sont :

- ☞ La première ligne contient trois valeurs : le temps d'exécution de la tâche, le numéro de la tâche et le nombre de successeurs de la tâche.
- ☞ La deuxième ligne cite les numéros de successeurs de la tâche courante.
- ☞ La troisième ligne contient les valeurs inférieures et supérieures de l'intervalle dans lequel le temps d'exécution est valable (c'est pour le cas des tâches à temps d'exécution variable)
- ☞ La quatrième ligne contient un numéro indiquant le type de paramètre de corrélation.
- ☞ La cinquième ligne indique le niveau de la tâche dans le graphe.

A noter que pour les tâches à temps d'exécution fixe ils auront des valeurs nulles pour les limites des intervalles et pour le type de paramètre de corrélation.

<u>Exemples :</u>

Données concernant une tâche à temps d'exécution fixe :

```
13.96423  4  3
51   52   53
 0     0
 0
 4
```

Données concernant une tâche à temps d'exécution variable :

```
5.7538    72    5
81     82    83    84    85
10     48
  1
  7
```

III.5.3 Recherche des combinaisons réalisables

Il faut tout d'abord calculer le nombre maximal de combinaisons possibles pour pouvoir allouer la mémoire au tableau correspondant.
Nous traitons les tâches à temps d'exécution fixe (paramètre de corrélation nulle) : ces tâches ont des temps d'exécution constants pour toutes les combinaisons ; ensuite les tâches à temps d'exécution variable qui se changent d'une combinaison à une autre.
Pour pouvoir éliminer les combinaisons non réalisables, nous faisons un test sur les paramètres de corrélation ; s'il y a deux paramètres ou plus qui sont identiques, nous cherchons les tâches correspondantes en comparant les limites inférieures et supérieures des intervalles.
Pour chaque tâche, nous associons un tableau dynamique qui contient les numéros des tâches avec les quelles elle est corrélée. Nous initialisons les combinaisons non réalisables à zéro et par transfert par un tableau intermédiaire nous remplissons un autre tableau par les combinaisons retenues.

III.5.4 Exemple d'application : La détection de mouvement sur un fond d'image fixe

Notre code est prêt pour traiter n'importe quel graphe de flot de données qui comprend des tâches à temps d'exécution variable ou non. Pour tester ce code nous avons voulu le faire tourner sur le graphe de flot de données de notre application : Détection de mouvement sur un fond d'image fixe.
Le fichier de sortie obtenu était le suivant :

```
Le fichier d'entrée de graphe est: dmouv.txt
les combinaisons possibles sont:

12.900 13.064 3.609 13.964 3.625 41.559 4.346 8.214 0.002 0.001
12.900 13.064 3.609 13.964 6.188 41.559 5.754 8.214 0.002 0.001
12.900 13.064 3.609 13.964 7.678 41.559 7.154 8.214 0.002 0.001
12.900 13.064 3.609 13.964 3.625 41.559 4.346 15.050 0.002 0.001
12.900 13.064 3.609 13.964 6.188 41.559 5.754 15.050 0.002 0.001
12.900 13.064 3.609 13.964 7.678 41.559 7.154 15.050 0.002 0.001
   .
   .
```

III.6 Idée de réalisation

Une fois nous avons fait tourné le programme, expliqué dans le paragraphe précédent, sur un graphe de flots de données conditionné ; nous obtenons un ensemble de graphes de flots de données non conditionnés.

Sur chaque graphe ainsi obtenu nous appliquons un algorithme de partitionnement et les résultats générés seront sous forme des contextes qui seront mémorisés dans l'architecture pour programmer le processeur et le FPGA en fonction des données traitées.

Dans notre approche nous utilisons un algorithme génétique pour effectuer le partitionnement. Une fois les différents contextes obtenus à partir des configurations identifiées, il est possible de construire le flot de contrôle de l'application sur l'architecture. Ce flot de contrôle doit activer les traitements qui correspondent aux contextes définis par le partitionnement.

Au moment de l'exécution de l'application, nous utilisons donc comme valeur de contrôle les paramètres de corrélation trouvés pendant le *Profiling* pour pouvoir identifier le graphe de tâches à considérer et par la suite le contexte qu'il faut appliquer au FPGA et au processeur (figure 3.8).

Ce critère de contrôle peut être le paramètre de corrélation lui même comme il peut être une combinaison de plusieurs paramètres. En régime permanent nous pouvons calculer le critère de contrôle pour l'image (i) à partir de l'image (i-1) déjà traitée, ou bien par pondération sur les paramètres des images qui les précèdent.

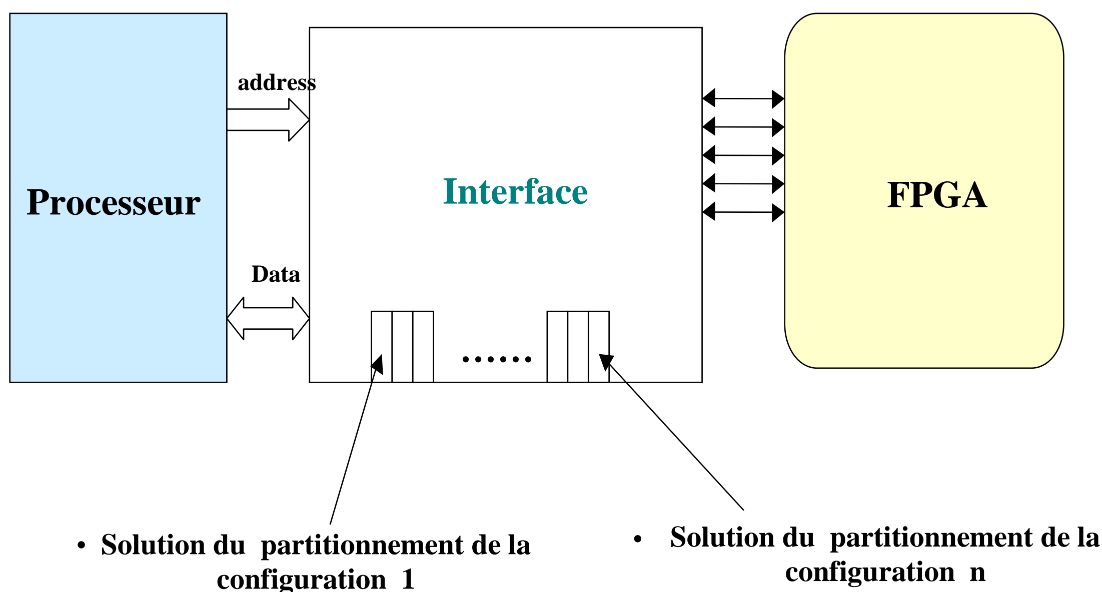


Figure 3.8 Idée de réalisation de l'approche de partitionnement

III.7 Conclusion

Dans ce chapitre, nous avons présenter notre nouvelle approche de partitionnement des applications à distribution variable de charge de calcul, que nous pouvons la résumer comme suit : il s'agit d'effectuer une sorte de *Profiling* sur l'application pour distinguer les tâches dont le temps d'exécution est variable avec le jeu de donnée en entrée, et déterminer par la suite leurs paramètres de corrélation. Ainsi nous pouvons transformer le graphe de flots de données de l'application en un graphe de flots de données conditionné. Avec un programme qui permet d'explorer l'espace de combinaisons réalisables nous aurons un ensemble de graphe de flots données non conditionnés. Nous faisons tourner l'algorithme génétique sur chacune de ces graphes pour chercher toutes les solutions optimales de partitionnement logiciel/matériel. Nous mémorisons ces solutions dans une interface dans l'architecture ; et avec une logique de contrôle, qui se sert des paramètres de corrélations identifiés précédemment, nous utilisons la solution de partitionnement adéquate pour chaque jeu de donnée qui arrive.

Notre approche ainsi décrite nous semble très efficace et donne des solutions au problème de partitionnement des applications à distribution variable de charge de calcul. Cependant la deuxième étape de cette approche et qui consiste à traiter les courbes résultats de *Profiling* pour construire le graphe de flots de données conditionné, est encore manuelle. Le concepteur doit utiliser tout son savoir faire pour guider à bien cette étape.
Concevoir une approche algorithmique qui permet d'automatiser cette étape nous semble une perspective pour ce travail.

Dans le prochain chapitre nous allons revenir sur l'outil de partitionnement qui est un algorithme génétique associée à une approche de Clustering.

Chapitre 4

Technique de partitionnement basée sur un algorithme génétique

IV.1 Introduction

Dans les chapitres précédents, nous avons élaboré une approche de partitionnement logiciel/matériel adaptatif à la distribution de charge de calcul de l'application partitionnée. La méthode de partitionnement utilisée dans cette approche est un algorithme génétique associé à une méthode de Clustering. Nous avons travaillé tout au long de ce projet sur l'application détection de mouvement sur un fond d'image fixe.

Dans ce chapitre nous allons reprendre notre approche de partitionnement, en introduisant l'outil de partitionnement utilisé et les différentes démarches pour l'expérimenter sur notre application.

IV.2 Modèle d'application retenu

Comme nous l'avons déjà expliqué dans le chapitre précèdent, le partitionnement opère sur des graphes de flots de données non conditionnés. L'outil de partitionnement récupère les résultats de l'estimation faite par le laboratoire LESTER (voir organigramme de dépendance du projet EPICURE sur figure 1), ces résultats sont sous la forme suivante :

- Un ensemble de tâches *SW* allouées sur le processeur hôte. Ces types de tâches ne présentent pas des contraintes temps réel, et une implantation logicielle leur permet une flexibilité assez importante (pour toute mise à jour ou extension).
- Un ensemble de tâches *HW* allouées sur le FPGA. Ces tâches doivent respecter des contraintes temps réel.
- Un ensemble de tâches dites FLOUES que l'estimateur n'arrive pas à les affecter au SW ou au HW. Les deux types des implantations sont possibles et envisageables.
- Une courbe de points d'implantation dans le plan temps d'exécution / Ressources pour chacune des tâches floues avec un point représentant l'implantation *SW* (0 ressource) et plusieurs points représentant différentes versions *HW* (par des techniques de déroulage de boucles : chercher tout parallélisme possible des traitements).

L'estimation donne aussi la taille des données transférées entre les différentes tâches (*DATA* : en nombre de mots/paquets) ainsi que leurs formats (*FORMAT* : 16 bits, 32 bits…) ce qui permet de calculer les temps de communication [46] :

- Nous considérons que les temps de communication entre deux tâches *SW* sont nuls :

$$\mathbf{T_{com}\,(\mathit{SW}\,,\,\mathit{SW}) = 0}$$

- On considère que les temps de communication entre une tâche *SW* et une tâche *HW* ou l'inverse (entre *HW* et *SW*) sont donnés par (voir figure 4.1) :

$$T_{com}\,(SW \longleftrightarrow HW) = (DATA * FORMAT) * \left(\frac{1_VB + T_A}{1_TB} + \frac{2_VB + T_A}{2_TB} \right) \quad \text{(Eq 4.1)}$$

Avec :
1_VB : Temps du passage d'un paquet de donnée par le Bus1 (ns)
2_VB : Temps du passage d'un paquet de donnée par le Bus2 (ns)

1_TB : Taille des paquets du Bus1 (en octets)
2_TB : Taille des paquets du Bus2 (en octets)

T_A : Temps d'accès mémoire (en lecture ou en écriture) par paquet de données. (en ns) [32].

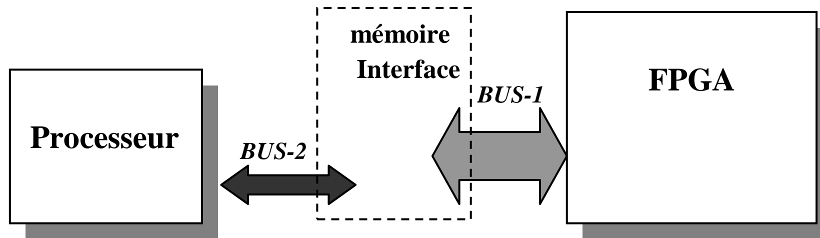


Figure 4.1 Transit des données par la mémoire d'interface

- Nous considérons que les temps de communication entre deux tâches *HW* sont donnés par l'équation Eq 4.2.

$$T_{com}\,(HW, HW) = 2 * (DATA * FORMAT) * \left(\frac{1_VB + T_A}{1_TB} \right) \quad \text{(Eq 4.2)}$$

Les temps de communication ainsi calculés sont attribués aux arcs correspondants dans le *DFG*. Il reste donc à déterminer la meilleure implantation pour chaque tâche (temps d'exécution, nombre de CLBs ('0' pour l'implantation *SW*) de manière à optimiser le temps d'exécution total.

Le partitionnement va principalement cibler les tâches floues. Pour cela, l'algorithme génétique est utilisé pour chercher la meilleure combinaison de ces tâches, entre la version *SW* et les différentes versions *HW*, qui minimisent à la fin une fonction de coût global. Cette fonction donne le temps d'exécution total de l'application. Pour évaluer l'algorithme génétique, un algorithme de 'Clustering' est utilisé pour définir les contextes de chaque individu (solution) et l'ordonnancement des tâches sur le processeur et le FPGA.

IV.3 Présentation des algorithmes génétiques

Les algorithmes génétiques sont des algorithmes d'optimisation s'appuyant sur des techniques dérivées de la génétique et de l'évolution naturelle : croisements, mutations, sélection, etc. Les algorithmes génétiques sont relativement anciens puisque les premiers travaux de John Holland sur les systèmes adaptatifs remontent à 1962 [38]. L'ouvrage de David Goldberg [39] a largement contribué à les vulgariser.

IV.3.1 Principes généraux

Un algorithme génétique recherche le ou les extrema d'une fonction définie sur un espace de données. Pour l'utiliser, nous devons disposer des cinq éléments suivants :

1. Un principe de codage de l'élément de population. Cette étape associe à chacun des points de l'espace d'état une structure de données. Elle se place généralement après une phase de modélisation mathématique du problème traité. La qualité du codage des données conditionne le succès des algorithmes génétiques. Les codages binaires ont été très utilisés à l'origine. Les codages réels sont désormais largement utilisés, notamment dans les domaines applicatifs pour l'optimisation de problèmes à variables réelles.
2. Un mécanisme de génération de la population initiale. Ce mécanisme doit être capable de produire une population d'individus non homogène qui servira de base pour les générations futures. Le choix de la population initiale est important car il peut rendre plus ou moins rapide la convergence vers l'optimum global. Dans le cas où l'on ne connaît rien du problème à résoudre, il est essentiel que la population initiale soit répartie sur tout le domaine de recherche.
3. Une fonction à optimiser. Celle-ci retourne une valeur appelée fitness ou fonction d'évaluation de l'individu.
4. Des opérateurs permettant de diversifier la population au cours des générations et d'explorer l'espace d'état. L'opérateur de croisement recompose les gènes d'individus existant dans la population, l'opérateur de mutation a pour but de garantir l'exploration de l'espace d'états.
5. Des paramètres de dimensionnement : taille de la population, nombre total de générations ou critère d'arrêt, probabilités d'application des opérateurs de croisement et de mutation.

Le principe général du fonctionnement d'un algorithme génétique est représenté sur la figure 4.2 :

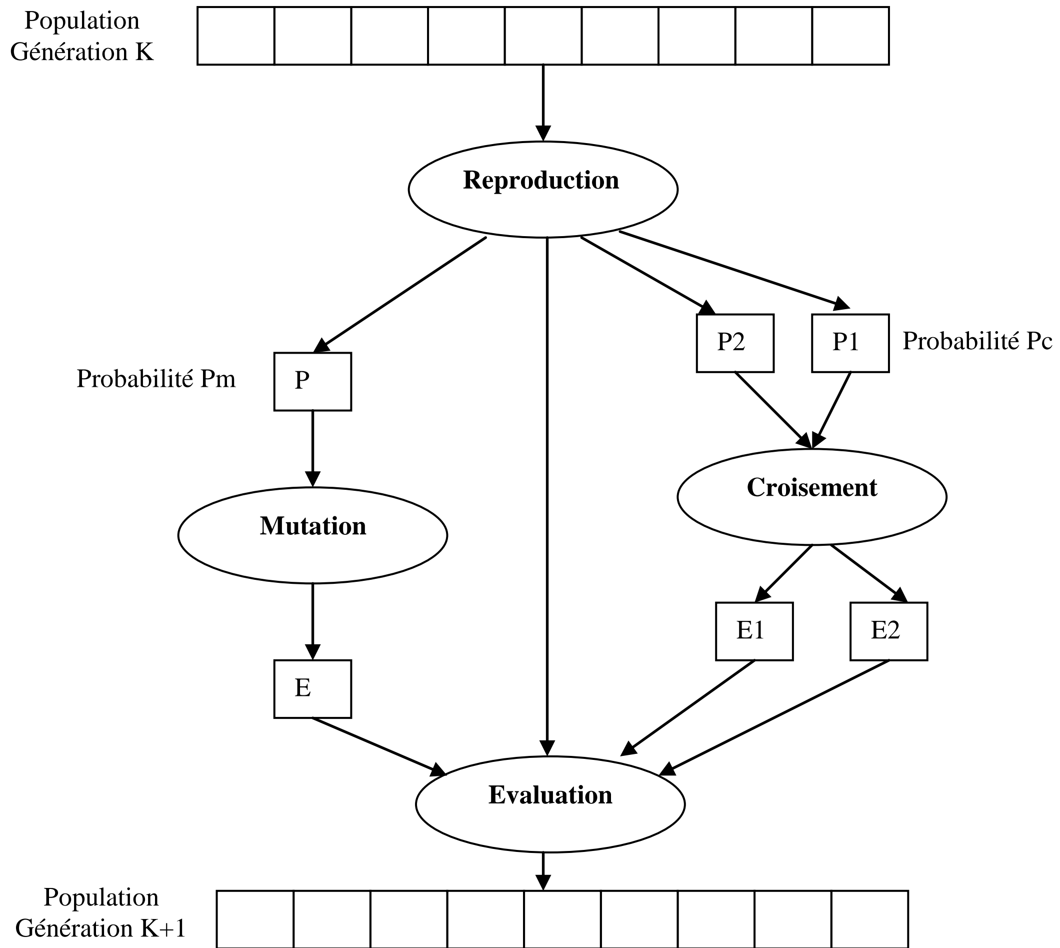


Figure 4.2 Principe général des algorithmes génétiques

Nous commençons par générer une population d'individus de façon aléatoire. Pour passer d'une génération k à la génération k+1, les trois opérations suivantes sont répétées pour tous les éléments de la population k. Des couples de parents P1 et P2 sont sélectionnés en fonction de leurs adaptations. L'opérateur de croisement leur est appliqué avec une probabilité Pc (généralement autour de 0.6) et génère des couples d'enfants C1 et C2. D'autres éléments P sont sélectionnés en fonction de leur adaptation. L'opérateur de mutation leur est appliqué avec la probabilité Pm (Pm est généralement très inférieure à Pc) et génère des individus mutés P'. Le niveau d'adaptation des enfants (C1, C2) et des individus mutés P' sont ensuite évalués avant insertion dans la nouvelle population. Différents critères d'arrêt de l'algorithme peuvent être choisis :

- ✓ Le nombre de générations que l'on souhaite exécuter peut être fixé a priori. C'est ce que l'on est tenté de faire lorsque l'on doit trouver une solution dans un temps limité.
- ✓ L'algorithme peut être arrêté lorsque la population n'évolue plus ou plus suffisamment rapidement.

Nous allons maintenant détailler chacun de ces points.

IV.3.2 Description détaillée

a) codage des données

Historiquement le codage utilisé par les algorithmes génétiques était représenté sous forme de chaînes de bits contenant toute l'information nécessaire à la description d'un point dans l'espace d'état. Ce type de codage a pour intérêt de permettre de créer des opérateurs de croisement et de mutation simples. C'est également en utilisant ce type de codage que les premiers résultats de convergence théorique ont été obtenus.

Cependant, ce type de codage n'est pas toujours bon comme le montrent les deux exemples suivants :

- ✓ Deux éléments voisins en terme de distance de Hamming ne codent pas nécessairement deux éléments proches dans l'espace de recherche. Cet inconvénient peut être évité en utilisant un codage de Gray.
- ✓ Pour des problèmes d'optimisation dans des espaces de grande dimension, le codage binaire peut rapidement devenir mauvais. Généralement, chaque variable est représentée par une partie de la chaîne de bits et la structure du problème n'est pas bien reflétée, l'ordre des variables ayant une importance dans la structure du chromosome alors qu'il n'en a pas forcément dans la structure du problème.

Les algorithmes génétiques utilisant des vecteurs réels [40] [41], évitent ce problème en conservant les variables du problème dans le codage de l'élément de population sans passer par le codage binaire intermédiaire. La structure du problème est conservée dans le codage.

b) Génération aléatoire de la population initiale

Le choix de la population initiale d'individus conditionne fortement la rapidité de l'algorithme. Si la position de l'optimum dans l'espace d'état est totalement inconnue, il est naturel de générer aléatoirement des individus en faisant des tirages uniformes dans chacun des domaines associés aux composantes de l'espace d'état en veillant à ce que les individus produits respectent les contraintes [42]. Si par contre, des informations a priori sur le problème sont disponibles, il parait bien évidemment naturel de générer les individus dans un sous-domaine particulier afin d'accélérer la convergence. Dans l'hypothèse où la gestion des contraintes ne peut se faire directement, les contraintes sont généralement incluses dans le critère à optimiser sous forme de pénalités. Il est clair qu'il vaut mieux, lorsque c'est possible ne générer que des éléments de population respectant les contraintes.

c) Gestion des contraintes

Un élément de population qui viole une contrainte se verra attribuer une mauvaise fonction fitness et aura une probabilité forte d'être éliminé par le processus de sélection. Il peut cependant être intéressant de conserver, tout en les pénalisant, les éléments non admissibles car ils peuvent permettre de générer des éléments admissibles de bonne qualité. Pour de nombreux problèmes, l'optimum est atteint lorsque l'une au moins des contraintes de séparation est saturée, c'est à dire sur la frontière de l'espace admissible.

Gérer les contraintes en pénalisant la fonction fitness est difficile, un ``dosage'' s'impose pour ne pas favoriser la recherche de solutions admissibles au détriment de la recherche de l'optimum ou inversement.

Disposant d'une population d'individus non homogène, la diversité de la population doit être entretenue au cours des générations afin de parcourir le plus largement possible l'espace d'état. C'est le rôle des opérateurs de croisement et de mutation.

d) Opérateur de Croisement

Le croisement a pour but d'enrichir la diversité de la population en manipulant la structure des chromosomes. Classiquement, les croisements sont envisagés avec deux parents et génèrent deux enfants. Initialement, le croisement associé au codage par chaînes de bits est le croisement à découpage de chromosomes (slicing crossover). Pour effectuer ce type de croisement sur des chromosomes constitués de M gènes, on tire aléatoirement une position dans chacun des parents. On échange ensuite les deux sous-chaînes terminales de chacun des deux chromosomes, ce qui produit deux enfants C1 et C2 (voir figure 4.3).

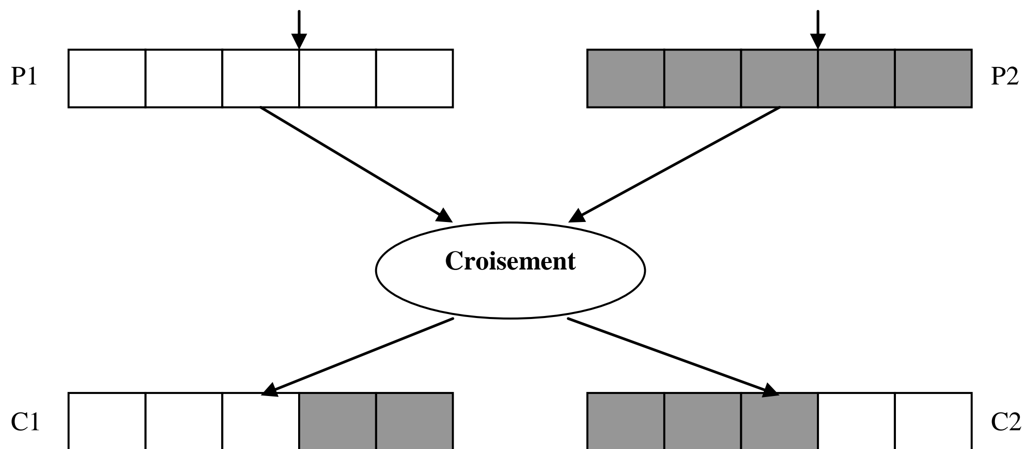


Figure 4.3 Slicing Crossover

Nous pouvons étendre ce principe en découpant le chromosome non pas en 2 sous-chaînes mais en 3, 4, etc [43].

e) Opérateur de Mutation

L'opérateur de mutation apporte aux algorithmes génétiques la propriété d'ergodicité de parcours d'espace. Cette propriété indique que l'algorithme génétique sera susceptible d'atteindre tous les points de l'espace d'état, sans pour autant les parcourir tous dans le processus de résolution. Ainsi en toute rigueur, l'algorithme génétique peut converger sans croisement, et certaines implantations fonctionnent de cette manière [44]. Les propriétés de convergence des algorithmes génétiques sont donc fortement dépendantes de cet opérateur sur le plan théorique.

Pour les problèmes discrets, l'opérateur de mutation consiste généralement à tirer aléatoirement un gène dans le chromosome et à le remplacer par une valeur aléatoire (voir figure 4.4).

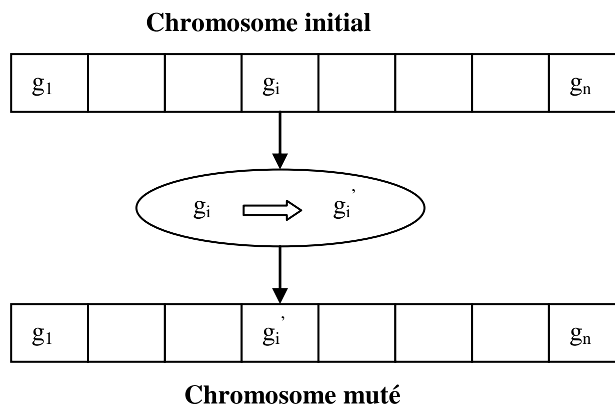


Figure 4.4 Principe de l'opérateur de mutation

f) Principes de sélection

A l'inverse d'autres techniques d'optimisation, les algorithmes génétiques ne requièrent pas d'hypothèse particulière sur la régularité de la fonction objective. L'algorithme génétique n'utilise notamment pas ses dérivées successives, ce qui rend très vaste son domaine d'application. Aucune hypothèse sur la continuité n'est non plus requise. Néanmoins, dans la pratique, les algorithmes génétiques sont sensibles à la régularité des fonctions qu'ils optimisent. Le peu d'hypothèses requises permet de traiter des problèmes très complexes. La fonction à optimiser peut ainsi être le résultat d'une simulation. La sélection permet d'identifier statistiquement les meilleurs individus d'une population et d'éliminer les mauvais. On trouve dans la littérature un nombre important de principes de sélection plus ou moins adaptés aux problèmes qu'ils traitent. Dans le cadre de notre travail, le principe de sélection suivant a été utilisé :

✓ Roulette wheel selection [39]

Le principe de Roulette wheel selection consiste à associer à chaque individu un segment dont la longueur est proportionnelle à sa fitness. Nous reproduisons ici le principe de tirage aléatoire utilisé dans les roulettes de casinos avec une structure linéaire. Ces segments sont ensuite concaténés sur un axe que nous le normalisons entre 0 et 1. Nous tirons alors un nombre aléatoire de distribution uniforme entre 0 et 1, puis nous identifions le segment sélectionné. Avec ce système, les grands segments, c'est-à-dire les bons individus, seront plus souvent adressés que les petits. Lorsque la dimension de la population est réduite, il est difficile d'obtenir en pratique l'espérance mathématique de sélection en raison du peu de tirages effectués. Un biais de sélection plus ou moins fort existe suivant la dimension de la population.

IV.4 Application de l'algorithme génétique au problème de partitionnement

L'utilisation d'un algorithme génétique pour résoudre le problème de partitionnement doit donc respecter les contraintes précitées.

Pour qu'un individu soit une solution de notre problème de partitionnement, il faut que le chromosome correspondant contienne l'information sur l'implantation (la version *SW* ou une des versions *HW*) de chacune des tâches constituant le graphe. Il faut que cette solution soit viable, c'est à dire qu'elle n'enfreint aucune des contraintes que nous nous sommes fixées.

La seule contrainte que nous ayons est celle de la surface maximale au niveau du FPGA. Une configuration (un contexte) constituée de plusieurs tâches ne peut pas être mappée sur le FPGA si la somme des ressources nécessaires à toutes les tâches dépasse la ressource maximale.

Un codage qui choisit pour chaque tâche une implantation et l'alloue en même temps à un contexte peut générer des individus non-viables du fait du caractère aléatoire des opérateurs de croisement et de mutation. De ce fait, nous avons pris la décision de séparer la fonction choisissant l'implantation et de l'attribuer à l'algorithme génétique, de celle choisissant les contextes et de l'attribuer à une heuristique de *clustering*.

IV. 4.1 Le Codage choisi

L'approche utilisée consiste à coder une solution par un vecteur formé de N indices (N est le nombre de tâches constituant le graphe) pointants sur N points des courbes d'implantations.

Pour illustrer ce type de codage, un exemple avec N = 7, le *DFG* et les courbes d'implantations sont donnés par la figure 4.5.

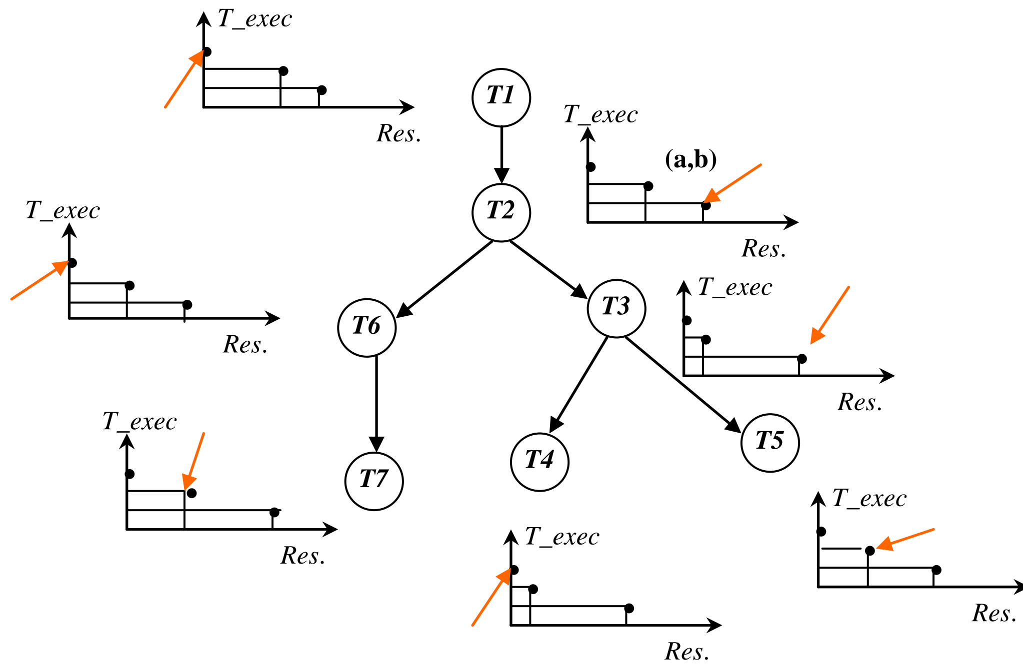


Figure 4.5 Exemple de DFG avec les implantations des tâches

La solution du partitionnement, indiquée par les flèches, est déduite du chromosome **(0,2,2,0,1,0,1)** pour donner comme résultat le temps d'exécution total et l'ordonancement des

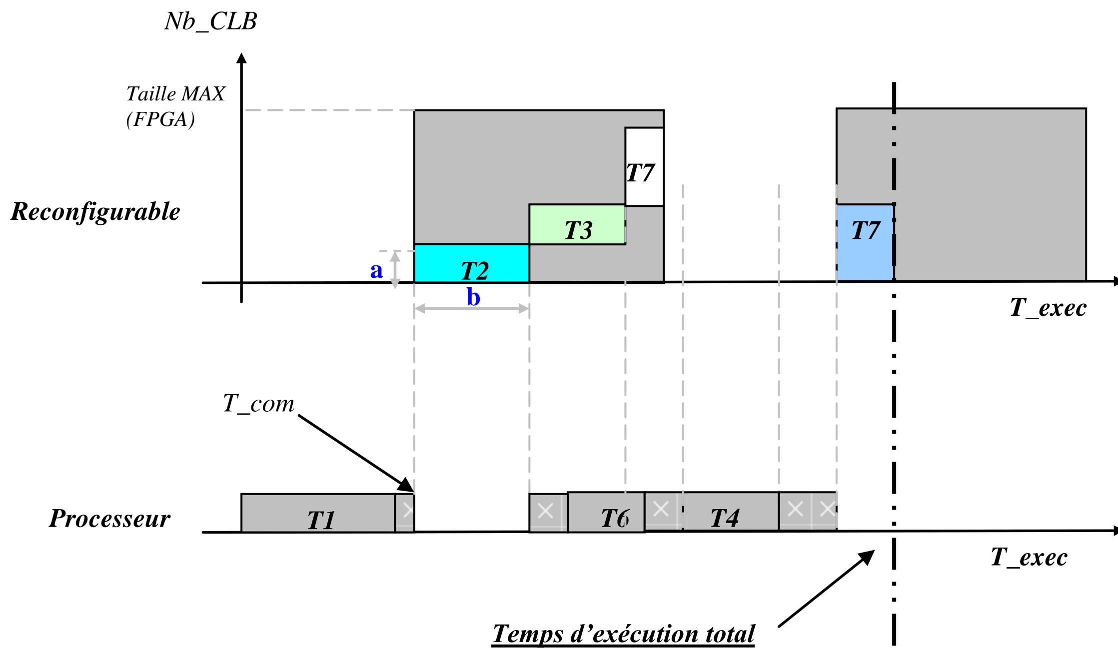


Figure 4.6 : exemple d'ordonnancement d'une solution sur l'architecture

tâches sur l'architecture (figure 4.6).

Toutes les solutions issues de ce codage sont viables. Reste maintenant à regrouper les tâches *HW* au sein de contextes (ou de *clusters*) de manière à retourner à l'algorithme génétique la meilleure évaluation de l'individu (de la solution) qu'il a proposé. C'est le rôle de l'étape de *clustering*.

IV. 4.2 L'évaluation

L'évaluation retourne pour chaque individu proposé par l'algorithme génétique, le temps d'exécution final. Elle se base sur un algorithme de *clustering* semblable à celui développé dans *COSYN* [45].

IV.4.2.1 L'algorithme de Clustering

L'approche de Clustering que nous avons utilisé dans le cadre de ce mémoire a été développée, dans le cadre d'un stage de DEA, dans [46]. L'objectif principal de cette approche est de chercher la meilleure évaluation d'une solution proposée par l'algorithme génétique. L'idée était de regrouper plusieurs tâches dans un même contexte (Cluster) et affecter des temps de communication nuls entre les tâches qui appartiennent au même cluster. Ainsi nous minimisons le temps d'exécution total qui sera recalculé après le Clustering.

L'algorithme commence par attribuer des priorités à chaque tâche en partant des tâches terminales (les feuilles du graphe) et de proche en proche jusqu'aux racines. Ensuite, il rassemble les tâches HW entre elles au sein du même *cluster* en partant des tâches les plus prioritaires.

L'attribution de priorités se fait selon la formule suivante :

$$Priority(\tau i) = T_exec(\tau i) + SUP_{(j)}(Priority(\tau j) + T_com(\tau i, \tau j)) \qquad \text{(Eq 4.3)}$$

Pour tout tâche τj où il y a communication de τi vers τj.

Si nous prenons l'exemple du DFG de la figure 4.7, annoté des temps de communications pour chaque arc, et d'un couple (temps d'exécution, nombre de CLB) pour chaque nœud, les calculs donne les priorités encadrées dans la figure 4.7.

La priorité de la tâche 4 par exemple, prend en compte le maximum des priorités des tâches 6 et 5 y compris leur temps de communication correspondant. Ceci donne un total de 12 au quel on ajoute le temps d'exécution de cette tâche (tâche 4), pour donner finalement une priorité de 24.

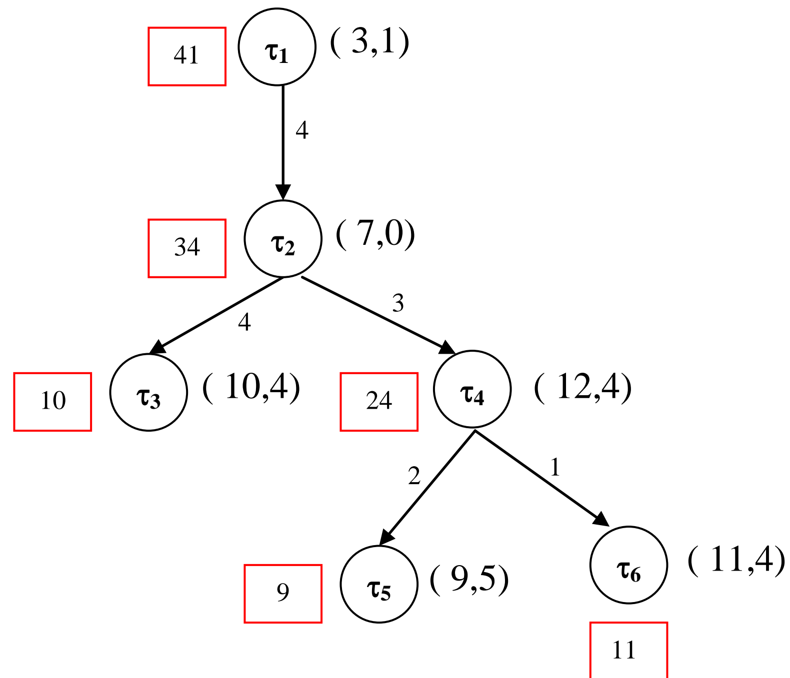


Figure 4.7 Exemple d'affectation des priorités dans un Clustering d'un DFG

Une calculées les priorités nous passons à la définition des contextes HW. En tenant compte d'un nombre maximal de ressources du FPGA (CLB), nous essayons d'allouer le maximum de tâches possibles à un contexte donné en partant des tâches les plus prioritaires et en procédant selon une priorité décroissante.

Si à un moment donné, la tâche à allouer au contexte en cours ne trouve pas assez de ressources, le contexte courant est clos et une reconfiguration est lancée. Nous passons au contexte suivant et ainsi de suite jusqu'à arriver à 'clusteriser' la dernière tâche du graphe.
La figure 4.8 reprend l'exemple du DFG précédent et présente deux contextes C1 et C2 déterminés sous la contrainte d'une surface maximale CLB_max = 10 CLB. Notons que les tâches SW n'interviennent pas dans l'étape de Clustering.

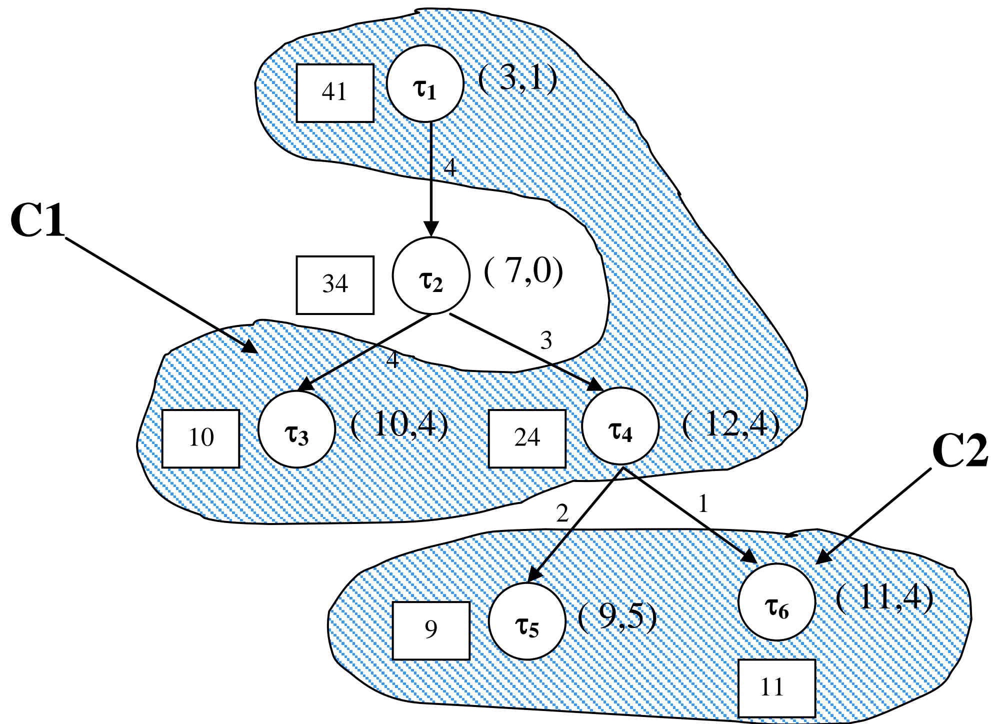


Figure 4.8 Définition des contextes d'un DFG

IV.4.2.2 L'évaluation d'un individu

Pour calculer le temps d'exécution total, il faut mettre à jour les temps de communication pour qu'ils tiennent compte des compositions des différents contextes.

- ✓ Mise à jour des temps de communication entre tâches d'un même contexte :

D'abord, nous commençons par mettre à jour les temps de communication des tâches se trouvant au sein du même contexte. On considère que le temps de communication entre deux tâches d'un même contexte est nul.

$$T_{com}\ (HW\ ,\ HW)_{CTXi} = 0 \qquad \text{(Eq 4.4)}$$

- ✓ Mise à jour des temps de communication entre tâches de contextes différents :

On considère que le temps de communication entre deux tâches *HW* appartenant à des contextes présents dans le cache est le temps de communication normal (correspondant à deux temps d'accès mémoire + deux temps de transit par le bus) présenté au paragraphe IV.2 :

$$T_{com}\ (HW_{CTXi}\ ,\ HW_{CTXj}) = T_{com}\ (HW,\ HW) \qquad \text{(Eq 4.5)}$$

Si les CTX i et j sont présents à la fois dans le cache.

Le temps de reconfiguration qui dépend, comme nous l'avons déjà vu, de la taille du contexte est calculé de la manière suivante :

$$T_reconf = Nb_CLB * T_reconf\ (1_CLB) \qquad \text{(Eq 4.6)}$$

Avec :
Nb_CLB : Le nombre de *CLB* utilisé pour le contexte.
T_reconf (*1_CLB*) : Le temps de reconfiguration d'un *CLB* (dépend de la technologie et du type d'FPGA).

Le temps de communication final entre deux tâches HW de contextes différents est donné par :

$T_{com} = SUP\ (T_{com}, T_reconf) \ \sim= T_reconf$ (Eq 4.7) dans la plupart des cas.

Nous devons donc, mettre à jour tous les temps de communication entre les tâches et calculer de nouveau l'évaluation de l'individu solution de l'algorithme génétique (temps d'exécution total).

IV.4.2.3 La fonction coût

La fonction coût peut tenir compte des plusieurs critères d'optimisation (temps d'exécution, consommation, surface, coût de réalisation, flexibilité, complexité, limites des ressources…). Cependant, il est très difficile de trouver un modèle mathématique qui décrit exactement la fonction coût vue la non-conformité des paramètres de cette dernière. Dans notre cas la fonction coût calcule le temps d'exécution totale de l'application :

Pour chaque tâche l'algorithme affecte une variable qu'il l'appelle ASAP donné par l'équation 4.8 pour les tâches allouées sur le SW et par l'équation 4.9 pour certaines tâches allouées sur le HW.

Pour les tâches allouées sur les unités SW :

ASAP (i) = Max (ASAP (j)) + Tcom (i,k) + Texe (i). (Eq 4.8)

Avec

- ✓ j : concerne toute tâche prédécesseur de la tâche i
- ✓ Le temps Tcom (i,k) est le temps de communication de la tâche i avec le prédécesseur qui a la plus grande ASAP.
- ✓ Le temps Texe (i) est le temps d'exécution de la tâche i.

Pour les tâches allouées sur les unités HW, nous distinguons deux cas :

- ➢ Si la tâche est précédée directement par une reconfiguration alors l'ASAP est donné par l'équation 4.9

ASAP (i) = Max (ASAP (j)) + Tcom (i,k) + Treconf + Texe (i). (Eq 4.9)

Avec Treconf est le temps de reconfiguration.

- ➢ Si la tâche n'est pas précédée directement par une reconfiguration, l'ASAP est donné par l'équation 4.8

A noter ici que l'outil de partitionnement permet de choisir entre une reconfiguration totale où la reconfiguration concerne tous les CLBs (ressources) de l'FPGA ou une reconfiguration partielle qui ne concerne que les CLBs qui vont être utilisés dans un contexte.
L'utilité de ce choix entre les types de reconfiguration est expliquée par la restriction de quelques technologies d'FPGA sur la reconfiguration totale et ne permettent pas une reconfiguration partielle.

La fonction coût considérée par l'outil du partitionnement dans notre cas sera donnée par l'équation 4.10

$$Fco\hat{u}t = Max\ (ASAP(i)) \qquad \text{(Eq 4.10)}$$

Avec i : variable qui parcourt toutes les tâches de l'application

IV. 4.3 Paramètres de réglage de l'algorithme génétique

Nous partons d'une population de départ de taille fixe (*pop_size*=200 par exemple), générée aléatoirement, ce qui correspond à 200 vecteurs de taille *N* (*N* est le nombre de tâches constituant le graphe).
Ces individus sont évalués selon la procédure de Clustering, et nous faisons correspondre à chacun d'eux un temps d'exécution. Ces individus sont ensuite classés par ordre de temps d'exécution croissant (des plus performants aux moins performants).

L'étape suivante est la sélection des individus appelés à se reproduire. La sélection se fait par tournois sans fossé des générations : c'est à dire que si nous désirons sélectionner 30 individus, il faut organiser 30 tournois de k individus (généralement entre 2 et 10) choisis uniformément dans la population entière et sélectionner les 30 plus performants.

Ces 30 individus sélectionnés vont générer une population d'enfants de taille fixe (*children_size*=50 par exemple) par des reproductions hétérozygotes (cas des opérateurs de croisement) ou homozygotes (cas des opérateurs de mutation).

Pour l'opérateur de mutation, un nombre seuil *S* assez petit est fixé, et nous parcourons le chromosome en tirant pour chaque gène un nombre aléatoire. Si ce nombre est inférieur à *S*, le gène mute sinon, nous passons au gène suivant.
Nous utilisons une évolution dynamique du nombre d'individus produits par chaque opérateur. Si un opérateur donne naissance à des individus performants, le nombre d'individus à produire par cet opérateur lors de la prochaine génération va augmenter.

Une fois la population d'enfant générée, il se pose le problème de son intégration dans la population courante. Pour cela, un renouvellement élitiste est utilisé : Les enfants sont eux-

mêmes à évaluer, ensuite les mauvais individus de la population sont substitués par les meilleurs de la nouvelle génération, tout en gardant une taille de population (*pop_size*) fixe.

La politique de renouvellement considérée ne permet pas d'avoir des clones pour lutter contre les éventuelles dérives génétiques. Une fois la population renouvelée, nous répétons le processus. Il reste à spécifier le critère d'arrêt pour cet algorithme. Nous utilisons dans notre approche, un arrêt s'il n'y a pas d'amélioration du meilleur individu après un nombre de générations donné (100 générations par exemple).

IV. 5 Conclusion

L'algorithme génétique associé à une approche de Clustering forment ensemble un outil de partitionnement efficace. Cependant plusieurs améliorations de cet outil sont possibles surtout au niveau du choix de la population initiale pour mieux guider le processus d'optimisation.

Le prochain chapitre est assigné pour présenter les résultats de nos travaux et les perspectives possibles dans le futur.

Chapitre 5

Résultats et analyses

V.1 Introduction

Dans les chapitres précédents, nous avons mis l'accent sur les caractéristiques des applications de traitement d'images et les problèmes liés au partitionnement de telles applications. Ensuite, nous avons proposé une nouvelle approche de partitionnement qui prend en compte la distribution variable de la charge de calcul de tâches.

Dans ce chapitre, nous allons présenter et interpréter les résultats de partitionnement de l'application détection de mouvement sur un fond d'image fixe.

V.2 Résultats de l'outil du partitionnement

Après avoir extrait toutes les combinaisons réalisables à partir d'un DFG conditionné de l'application «détection du mouvement sur un fond d'image fixe », nous avons appliqué l'algorithme génétique, associé à l'approche de Clustering, sur toutes ces configurations. Dans la suite nous présentons les résultats du partitionnement pour deux exemples de configurations : les configurations 3 et 6 (figure 5.1 et 5.2 respectivement).

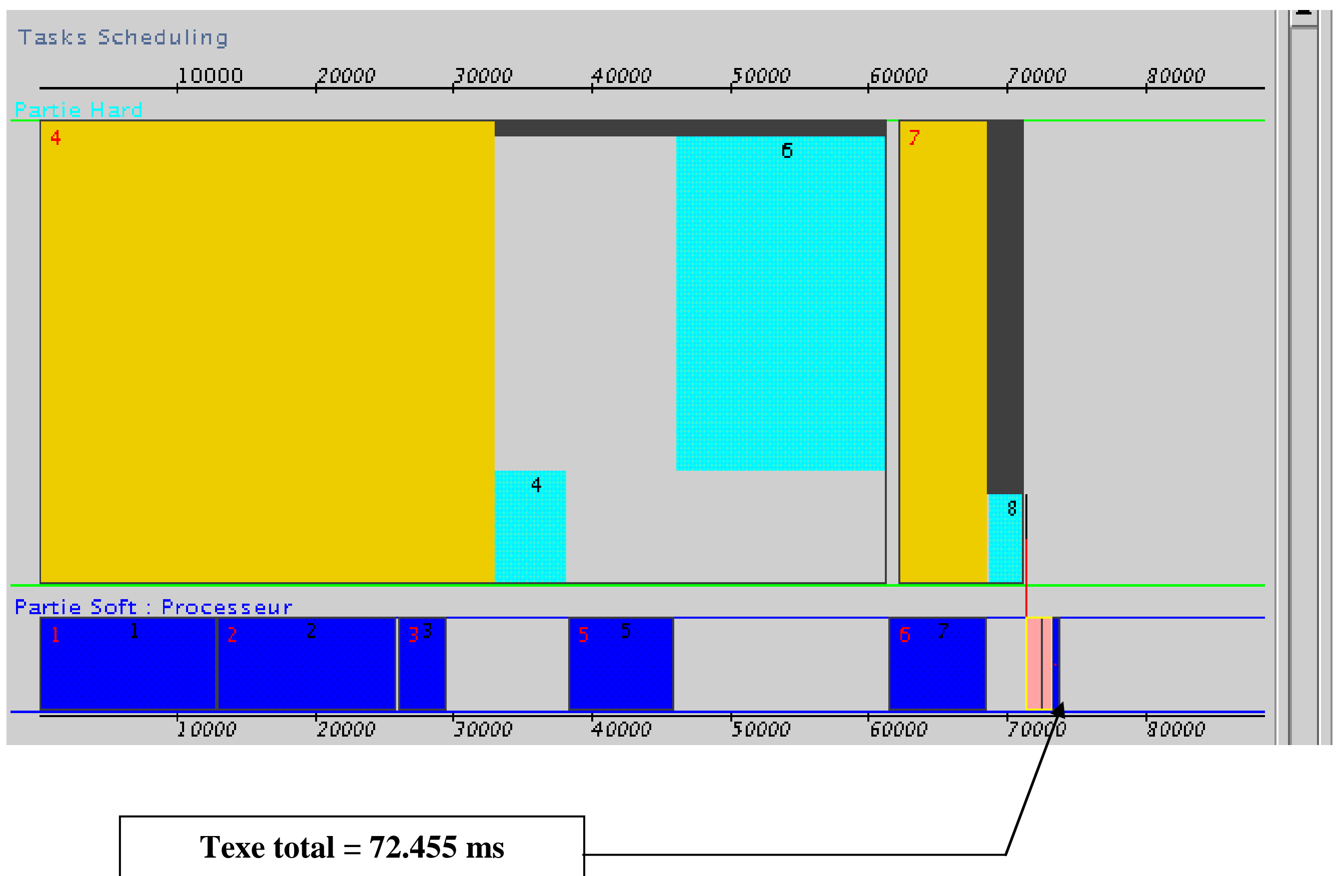


Figure 5.1 : Résultats du partitionnement de la configuration 3

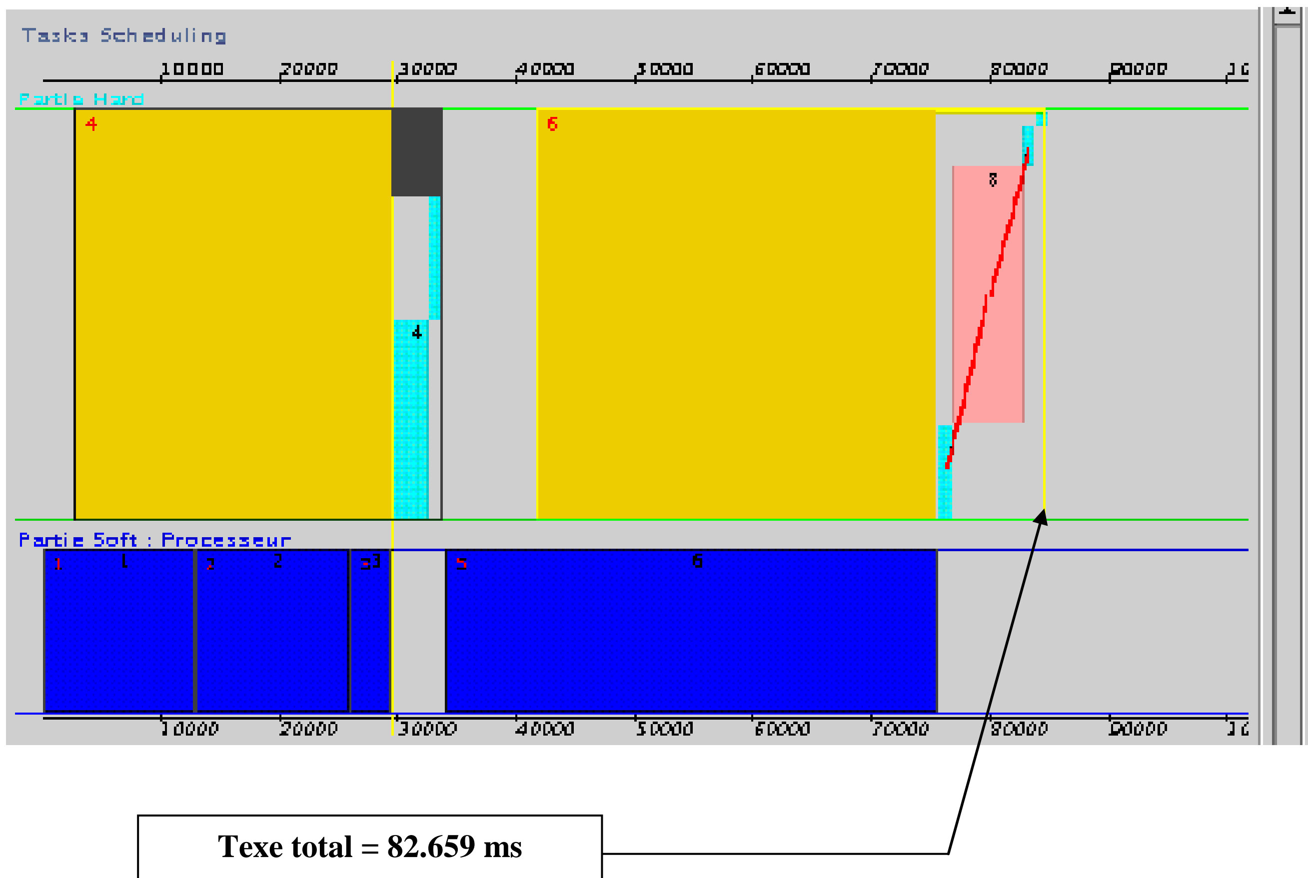


Figure 5.2 : Résultats du partitionnement de la configuration 6

Les figures 5.1 et 5.2 représentent les solutions de partitionnement de l'application détection de mouvement sur un fond d'image avec deux combinaisons possibles des tâches. Chaque solution est destinée à être appliquée sur un type particulier d'image : suivant ses caractéristiques (les paramètres de corrélation : nombre des objets dans l'image, nombre des pixels blancs …) dans le flot des images. Suivant les caractéristiques mesurées, nous appliquons tel ou tel partitionnement.

La solution de la configuration 3 représentée sur la figure 5.1 donne un temps d'exécution total de **72.455 ms**. Ce temps ne correspond pas à la cadence vidéo (Objectif cherché) parce que les valeurs du temps d'exécution sur l'FPGA ne sont que des estimations théoriques et ne reflètent aucune réalité. Cette solution de partitionnement (figure 5.1) utilise un total de ressources matérielles de **850** LABs (Logic Array Block) en implantant les tâches 4, 6 et 8 (en cyan sur la figure 5.1) sur le FPGA. La somme des temps de reconfiguration (en jaune sur les figures) est de **35.11 ms.**

Avec ce partitionnement (de la configuration 3) nous aurons **6** transferts entre l'unité SW et l'unité HW donc un temps de communication total de **0.6774.**

En comparant cette solution de partitionnement avec celle de la configuration 6 représentée sur la figure 5.2, nous remarquons qu'il y a des différences sur trois points :

- Le temps d'exécution total
- Les temps de communications
- L'utilisation des ressources du FPGA

Dans la suite nous allons analyser ces différences pour les solutions de toutes les configurations.

V.3 Temps d'exécution total

La figure 5.3 représente la distribution du temps d'exécution total de l'application pour toutes les configurations.

Nous remarquons qu'il y a des différences (parfois nettes parfois légères) entre les différents temps d'exécution. Dans le cas pratique de la caméra intelligente, nous allons opter pour la solution de temps d'exécution dans le pire de cas dans le but de fixer un débit de travail (qui

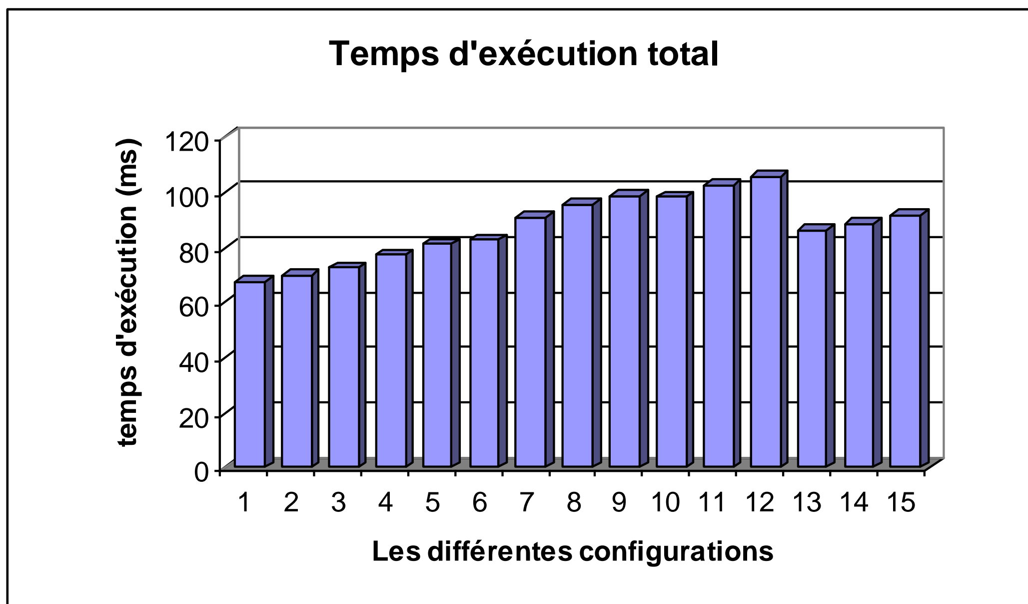


Figure 5.3 : Temps d'exécution total de toutes les configurations

correspond au débit de défilement des images) qui doit être déterminé une fois pour toute pour tous les types des images (selon leur complexité). Dans notre cas, le débit de travail doit correspondre à la configuration 12. Il en découle un temps libre (voir figure 5.4) dans le cas de l'exécution des autres partitionnements qui peut être utile pour d'autres types de traitement comme par exemple : l'amélioration de la qualité de l'image en ajoutant une fonction correspondante dans l'intervalle du temps libre.

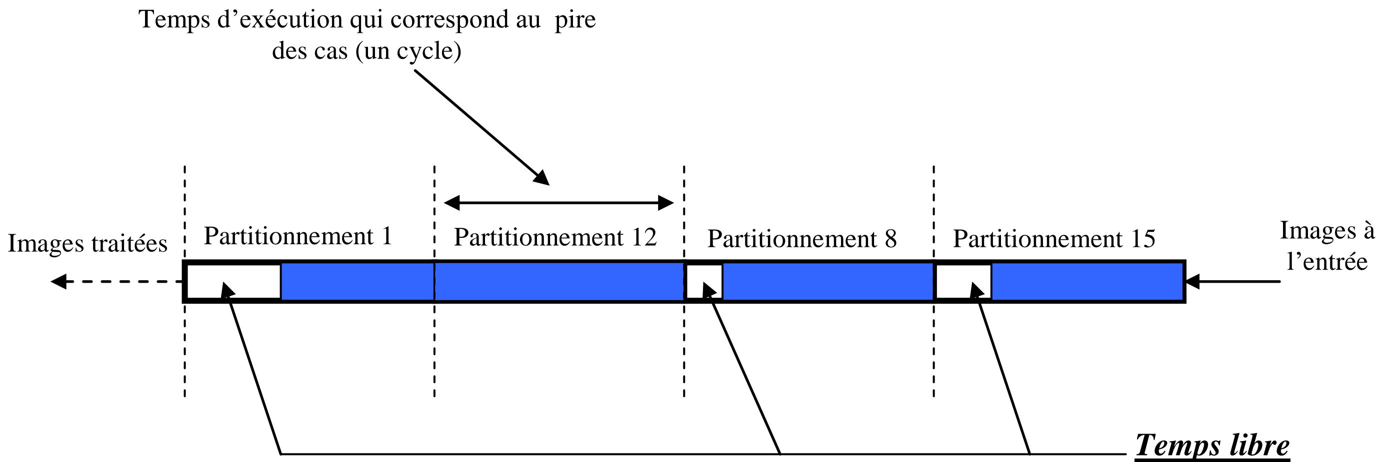


Figure 5.4 : Principe du temps libre pendant l'exécution de l'application

V.4 Temps de communication

Le temps de communication, dans le cas de notre application, est le même pour toutes les tâches parce que la taille des données échangées entre deux tâches est toujours constante : il correspond à la taille d'une image.

La figure 5.5 représente la somme des temps de transfert de toutes les communications (SW-HW ou HW-HW) pour chaque solution de partitionnement.

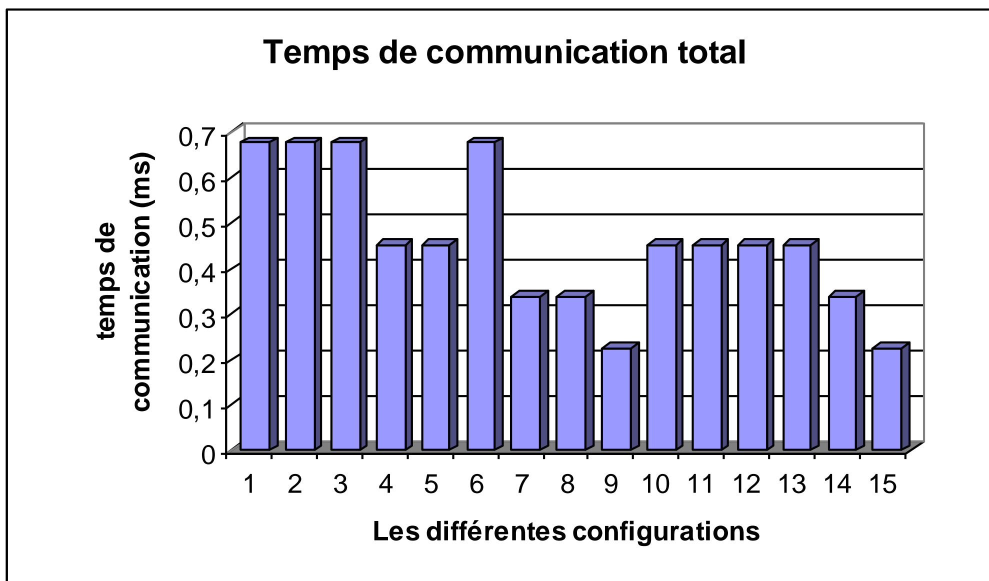


Figure 5.5 Temps de communication total pour toutes les configurations

Les différentes solutions de partitionnement ne possèdent pas toutes les mêmes temps de communication. Ces résultats illustrent l'avantage de notre approche par rapport aux approches classiques qui travaillent avec la solution de partitionnement de la configuration de pire de cas.

En effet, quand nous adaptons le partitionnement aux caractéristiques de l'image traitée, nous arrivons à diminuer le temps de communication (pour une catégorie des images).

Diminuer le temps de communication entre les unités d'une architecture, engendre une diminution de la consommation du système car les opérations d'accès mémoire et de transfert de données à travers les bus sont assez "gourmands" en terme de consommation.

V.5 Utilisation des ressources matérielles

La dernière différence observée sur les résultats du partitionnement concerne l'allocation des ressources matérielles. Les différentes solutions n'ont pas les mêmes allocations des CLBs (ou LABs). D'une solution de partitionnement à une autre, nous pouvons avoir une diminution de nombre des ressources utilisées et ces dernières peuvent être utilisées pour réaliser d'autres fonctions (telle qu'une amélioration de la qualité de l'image).

V.6 Travaux futurs

Actuellement, notre approche permet de chercher toutes les combinaisons réalisables d'un graphe de flots de données conditionné. Nous cherchons par la suite les solutions optimales de partitionnement de ces combinaisons avec l'outil de partitionnement adopté.

Pour optimiser d'avantage les résultats de partitionnement, nous avons pensé à exploiter les résultats de partitionnement d'une configuration pour partitionner la configuration suivante. Cette exploitation peut être faite sous deux formes :

- ☞ Nous intervenons au niveau de la population initiale de l'algorithme génétique en utilisant les résultats du partitionnement de la configuration précédente. Ainsi nous guidons l'évolution de l'algorithme génétique vers une solution plus optimale puisque toutes les populations initiales sont des solutions optimales sauf pour la première configuration où la population initiale est choisie au hasard.
- ☞ La deuxième forme s'inspire de la première en ajoutant des probabilités aux combinaisons. Nous commençons par le partitionnement de la configuration qui correspond au pire de cas ensuite nous affectons des probabilités aux restes des configurations afin de partitionner ensuite la configuration la plus probable et ainsi de suite jusqu'au partitionnement de la dernière configuration.

Les probabilités sont affectées selon plusieurs paramètres :

- ✓ En comparant les paramètres de corrélations pour les tâches à temps d'exécution variable. En effet la combinaison ayant les paramètres de corrélations les plus proches de celles de la combinaison déjà partitionnée est la plus probable si l'image suivante conduit à un changement de configuration.
- ✓ Il faut tenir en compte aussi du dernier état du reconfigurable (FPGA) lors du partitionnement précédent. Pour une tâche à temps d'exécution variable, si lors du dernier partitionnement, la tâche est allouée sur le reconfigurable, nous allons chercher à ce qu'elle reste sur le reconfigurable dans la prochaine configuration. Ainsi nous exploitons le temps de reconfiguration d'une configuration pour l'autre…

Des recherches sur cet axe sont envisageables dans le laboratoires I3S de sophia-Antipolis dans le cadre d'une continuation de nos travaux sur le partitionnement des applications à distribution variable de charges de calcul.

V.7 Conclusion

Les résultats du partitionnement présentés dans ce chapitre sont très utiles pour améliorer le processus de conception conjointe (Codesign) des systèmes mixtes logiciel/matériel. En effet notre approche aide le concepteur à partitionner les tâches dont le temps d'exécution est corrélé à la nature de données.
Le critère actuel de l'algorithme de partitionnement est le temps d'exécution global minimum, cependant l'objectif de l'application est de vérifier une cadence d'exécution minimale ; donc nous pouvons évoluer les critères de partitionnement pour tenir en compte de la contrainte du temps d'exécution maximum et/ou de minimiser la consommation…

Actuellement les mesures du temps d'exécution sur les unités matérielles ne sont que des estimations. Avoir des mesures exactes nécessite de la programmation de l'application en langage VHDL ce qui sort du cadre de ce DEA mais permettrait d'obtenir des résultats plus précis qui pourrait être comparés à une expérimentation sur un système réel.

Conclusion générale

L'évolution progressive de la technologie a permis aujourd'hui aux systèmes reconfigurables d'intégrer un ou plusieurs cœurs de processeurs, des mémoires et des unités matérielles spécifiques configurées sur la matrice programmable. Cependant, ces évolutions technologiques sont récentes et n'ont pas été suivies par la définition de méthodologies de conception et par des outils efficaces associés.
Aussi le concepteur des systèmes électroniques mixtes (logiciel/matériel) se trouve fréquemment confronté à un espace de conception très vaste au sein duquel il est difficile de converger vers les solutions les plus adaptées. Il est nécessaire donc d'améliorer les approches de conception actuelles.

Face à ces problèmes, nous avons proposé une nouvelle approche de partitionnement logiciel/matériel qui permet de partitionner les fonctions d'une application sur les éléments d'une architecture cible. Les applications principalement visées par notre approche sont celles qui montrent une distribution variable de charge de calcul telles que les applications de traitement d'image, où le temps d'exécution d'une fonction est fortement corrélé au contenu de l'image traitée.
L'approche ainsi développée dans ce mémoire, apporte des améliorations nettes au niveau adaptation du partitionnement au jeu de donnée. Elle permet d'avoir aussi une meilleure exploitation des ressources de la composante matérielle de l'architecture. D'autre part les résultats du Profiling, adopté dans notre approche, permettent de donner des configurations qui réduisent le pessimisme du cas pire.

Cependant la deuxième étape de cette approche et qui consiste à traiter les résultats de *Profiling* pour construire le graphe de flots de données conditionné, est encore manuelle. Le concepteur doit utiliser son savoir-faire pour guider cette étape.
Concevoir une approche algorithmique qui permet d'automatiser cette étape nous semble une perspective pour ce travail.

Un autre problème posé lors de validation de notre approche, est le choix des séquences de test pour caractériser les comportements de l'application qui induisent les configurations retenues pour le partitionnement. Ces configurations doivent être représentatives dans le but de garantir que les exécutions du système, sur les images réelles vérifient les contraintes. Si nous nous limitons à un cas particulier de traitement tel que la

surveillance d'un parking de voitures, le choix d'une séquence de test représentative peut être aisé à déterminer. Par contre dans un environnement plus général où il est plus difficile de caractériser les scènes analysées, il faut penser à un partitionnement dynamique en fonction de la nature des données. Une telle approche sera considérée dans nos futurs travaux.

Références

[1] **F.GHAFFARI**, M.AUGUIN, M.BEN JEMAA, "Etude du partitionnement logiciel/matériel d'applications à distribution variable de charge de calcul". Renpar'14 /ASF/SYMPA, Hamamet, TUNISIE, pp. 334–338, 10 – 13 Avril 2002.

[2] M, Abid, "Contribution à la conception des systèmes mixtes logiciel/matériel", thèse d'état présentée à l'ENIT (TUNISIE), Mai 2000.

[3] S. Bilavarn, "Exploration Architecturale au Niveau Comportementale- Applications aux FPGAs". Thèse de doctorat de l'université de Bretagne Sud, Février 2002.

[4] Nios Embedded Processor Getting Started Version 1.1 user Guide, ALTERA March 2001.

[5] A. KALAVADE, E.A. LEE, "A Hardware-Software Codesign Methodology for DSP Applications", IEEE Design & Test of Computers, Vol. 10, N° 3, pp. 16-28, September 1993.

[6] A. Kalavade, and E.A. Lee, "Manifestations of Heterogeneity in Hardware/Software Codesign", Proc. 31 st Design Automation Coference (DAC), IEEE CS Press, pp. 437-438, June 1994.

[7] A. KALAVADE, E. LEE, "Global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem, International Workshop on Hardware/Software Codesign, pp 42-48, September 1994.

[8] M.B. Srivastava, and R.B. Brodersen "Rapid-prototyping of Hardware and Software in a Unified Framework", Proc. Int'l Conf. On Computer-Aided Design (ICCAD), IEEE CS Press, pp. 152-155, 1991.

[9] M.B. Srivastava, and R.B. Brodersen, "Using VHDL for High-Level, Mixed-Mode Simulation", IEEE Design & Test of Computers, pp. 31-40, September 1993.

[10] M.B. Srivastava, Rapid-prototyping of Hardware and Software in a Unified Framework, Ph.D. thesis, University of Calif. Berkeley, June 1992.

[11] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, A. Sangiovanni-Vincentelli, "A Formal Specification Model for Hardware/Software Codesign", Wshp Handouts of Int'l Wshp on Hardware-Software Co-Design, Cambridg, Massachusetts, IEEE CS Press, p. 53, October 1993.

[12] R. GUPTA, G. DE MICHELI, "Hardware-Software Cosynthesis for Digital Systems", IEEE Journal Design and Test of Computers, pp 29-41, september, 1993.

[13] R.K. GUPTA, G. DE MICHELI, "System-level Synthesis using Re-programmable Components", Proc. Third European Conference on Design Automation, IEEE CS Press, pp. 2-7, 1992.

[14] R.K. GUPTA, C.N. Coelho Jr., and G. DeMicheli, "Program Implementation Schemes for Hardware-Software Systems", Wshp Handouts of Int'l Wshp on Hardware-Software Co-Design, IEEE CS Press, October 1992.

[15] R.K. Gupta, C.N. Coelho Jr., and G. DeMicheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components", Proc. 29th Design Automation Conference (DAC), IEEE CS Press, pp. 225-230, 1992.

[16] R.K. GUPTA, C.N. Coelho Jr., and G. DeMicheli, "Program Implementation Schemes

for Hardware-Software Systems", IEEE Computer, Vol. 27, N° 1, pp. 48-55, January 1994.

[17] D. Gajski, F. Vahid, "Specification and Design of Embedded System", IEEE Design & Test of Computers, pp. 53-67, Spring 1995.

[18] D. Gajski, F. Vahid, and S. Narayan, "A system-Design Methodology : Executable-Specification Refinement", Proc. European Design & Test Conference (EDAC-ETC-EuroASIC), Paris, France, IEEE CS Press, pp. 458-463, February 1994.

[19] J. Gong, D. Gajski, and S.Narayan, "Software Estimation from Executable Specifications", Proc. European Design Automation Conf. (EuroDAC), IEEE CS Press, Grenoble, France, pp. 47-57, September 1994.

[20] R. ERNST, J. HENKEL, T. BENNER, "Hardware-Software Cosynthesis for Microcontrollers", IEEE Journal Design and Test of Computers,Vol. 10, N° 4, pp. 64-75, December 1993.

[21] D.E. Thomas, J.K Adams, and H. Schmit, "A Model and Methodology for Hardware-Software Codesign", IEEE Design & Test of Computers, Vol. 10, N° 3, pp. 6-15, September 1993.

[22] J.K. Adams, H. Schmitt, and D.E.Thomas, "A Model and Methodology for Hardware-Software Codesign", Handouts of Int'l Wshp on Hardware-Software Co-design, Cambridge, Massachusetts, IEEE CS Press, October 1993 p. 6.

[23] E.Barros, W.Rosentiel, and X. Xiong, "A Method for Partitioning UNITY Language in Hardware and Software", Proc. European Design Automation Conference (EuroDAC), IEEE CS Press, Grenoble, France, September 1994, pp. 580-585.

[24] E.Barros, and W .Rosentiel, "A Method for Hardware Software Partitioning" , IEEE Comp. Euro,1992, p. 5.

[25] Y. LI, T. CALLAHAN, E. DARNELL, R. HARR, U. KURKURE, J. STOCKWOOD, "Hardware-software co-design of embedded reconfigurable architectures", Design Automation Conference, Los Angeles, pp. 253-260, June 2000.

[26] R. MAESTRE, F. KURDAHI, N. BAGHERZADEH, H. SINGH, R. HERMIDA, M.FERNANDEZ, "Kernel Scheduling in reconfigurable computing", DATE, Munich, p. 90, march, 1999.

[27] N.L. Rethman, and P.A. Wilsey, "RAPID : A Tool for Hardware/Software Tradeoff Analysis", Proc. IFIP Conf. Hardware Description Languages (CHDL), Publ. Elsevier Science, Ottawa, Canada, April 1993.

[28] Z. Peng, and K. Kuchcinki, "An Algorithm for Partitioning of Application Specific Systems", Proc. European Design & Test Conference (EDAC-ETC-EuroASIC), IEEE CS Press, pp. 316-321, February 1993.

[29] M. Edwards, and J. Forrest, A Development Environnement for the Cosynthesis of Embedded Software/Hardware Systems, Proc. European Design & Test Conference (EDAC-ETC EuroASIC), Paris, France, IEEE CS Press, pp. 469-473, February 1994.

[30] J. Henkel, R. Ernst, U. Holtman, and T. Benner, Adaptation of Partioning and High-Level Synthesis in Hardware/Software Co-Synthesis, Proc. Int'l Conf. On Computer-Aided Design (ICCAD), IEEE CS Press, pp. 96-100, 1994.

[31] D.Herman, J. Henkel, and R. Ernst, An Approach to the Adaptation of Estimated Cost Parameters in the Cosyma System, Proc. Third Int'l Wshp on Hardware/Software

Codesign (CODES/CASHE), Grenoble, France, IEEE CS Press, pp. 100-107, September 1994.

[32] B. DAVE, G. LAKSHMINARAYANA, N. LHA, "COSYN: hardware/software co-synthesis of embedded systems", Design Automation Conference, pp 703-708, Anaheim, June 9-13, 1997.

[33] B. DAVE, "CRUSADE: hardware/software co-synthesis of dynamically reconfigurable heterogeneous real time distributed embedded systems", DATE, Munich, p. 97, march, 1999.

[34] M. AUGUIN, L. BIANCO, L. CAPELLA, E. GRESSET, "Partitioning conditional data flow graphs for embedded system design", International Conference on Application Specific Systems, Architectures and Processors, ASAP, Boston, p.337, July 10-12, 2000.

[35] Les architectures reconfigurables, Stéphane Rubini (laboratoire d'informatique de Brest) et Dominique Lavenier (IRISA), Technique et science informatiques, vol 18,no 10, 1999.

[36] A. Seznec, S. Jourdan, P. Sainrat, P. Michaud, ``Multiple-Block Ahead Branch Prediction'', International Symposium on Computer Architecture, pp. 116-127, May 1996.

[37] M.Auguin, L.Bianco, L.Capella, E.Gresset. Conception de systèmes embarqués par partitionnement de spécifications flots de données conditionnel, Conférence Architectures Nouvelles de Machines, Sympa'6, Besançon, pp. 139-148, 19-21 juin, 2000.

[38] John Holland. "Outline for a logical theory of adaptive systems". Journal of the Association of Computing Machinery, vol. 9, no. 3, pp. 297-314, July 1962.

[39] D.E Goldberg. "Genetic Algorithms in Search, Optimization and Machine Learning", Reading MA Addison Wesley, 1989.

[40] D.E Goldberg. "Real-coded genetic algorithms", virtual alphabets and blocking. Complex Systems, 139-167, 1991.

[41] A.H Wright. "Genetic algorithms for real parameter optimization". In Proceeding of the Foundation Of Genetic Algorithms. FOGA, 1991.

[42] Z Michalewicz and C.Z Janikov. "Handling constraints in genetic algorithms". In Proceedings of the Fourth International Conference on Genetic Algorithm. ICGA, pp. 151-157, San Mateo, California, 1991.

[43] C.L Bridges and D.E Goldberg. "An analysis of multipoint crossover". In Proceedings of the Foundation Of Genetic Algorithms. FOGA, 1991.

[44] L.J Fogel, A.J Owens, and M.J Walsh. Artificial Intelligence Through Simulated

Evolution. Wiley and sons. NY, 1966.

[45] B. P. Dave, G. Lakshminarayana and N. K. Jha, COSYN : Hardware-Software Co-Synthesis Heterogeneous Distributed Embedded Systems, IEEE Transactions on VLSI Systems, Vol. 7, No. 1, pp. 243-258 March 1999

[46] K.Ben Chehida, Partitionnement Logiciel/Matériel pour des architectures reconfigurables utilisant une approche génétique, Rapport de stage de DEA Université de Nice Sophia - Antipolis, Juillet 2001.

EPICURE
Environnement de partitionnement et de Co-développement pour Utilisation sur architectures REconfigurables

Objectifs :

Aujourd'hui, l'évolution de la demande des marchés pour des applications plus performantes et donnant accès à plus de services conduit à des systèmes plus complexes, malléables et miniaturisés.
Ces nouveaux dispositifs intègrent ainsi, bien souvent sur une même puce de silicium (System-On-a-Chip), des fonctionnalités relevant de domaines de plus en plus variés (traitement de signal, traitement de données, contrôle, …). Celles-ci sont programmées sur les processeurs les mieux adaptés (µ-contrôleur, processeurs RISC/CISC, DSP), et depuis peu, grâce aux progrès des technologies submicroniques, seront programmées sur des structures reconfigurables.
Ce projet apportera des solutions au développement d'applications basées sur du matériel de plus en plus malléables qui impliquent une forte coopération entre les différentes unités de traitement intégrées dans le produit.
L'un des défis majeurs à relever pour la maîtrise de ces dispositifs consiste en particulier à obtenir, dès les phases de spécification et de conception préliminaire des applications, une bonne adéquation entre l'implantation du logiciel sur les unités de calcul traditionnelles (µ-contrôleurs, processeurs, DSP) et sur les structures reconfigurables .
Dans le projet EPICURE, nous nous proposons d'étudier un environnement qui intégrera la méthode et les outils logiciels afin de réaliser ce partitionnement de manière automatique ou semi-automatique.
En entrée de la méthode, les spécifications seront développées à l'aide de formalismes basés en langage synchrone de type Esterel, Signal. A ce stade, UML servira de vecteur pour la spécification à haut niveau des applications pour aller, après partitionnement, jusqu'à la génération des codes implantés sur les processeurs (C, Java) et sur les structures reconfigurables.
L'originalité de notre proposition réside d'abord dans l'intégration de la méthode dans des flots de conception du marché mais surtout dans son optimisation aux nouvelles architectures de processeurs reconfigurables.
Le projet EPICURE répond aux priorités énoncées par le groupe de travail B1 sur les systèmes embarqués. En particulier et parmi les technologies à développer, il adresse plus particulièrement le domaine des méthodes de CoDesign mentionné au paragraphe 5.10 de l'appel. Les opportunités de marchés, sans être limitatifs, liés à ces développements concernent les applications grand public, les applications mobiles et/ou communiquantes (set top box, multimédia, agenda personnel, téléphonie). Ce projet apportera des solutions au développement d'applications basées sur du matériel de plus en plus malléables qui impliquent une forte coopération entre les différentes unités de traitement intégrées dans le produit.

Mise en œuvre et état de l'art :

De nouvelles technologies émergent actuellement (structures reconfigurables) qui apportent un degré de malléabilité élevé au matériel. Pour en exploiter toutes les performances de nouveaux outils et méthodes sont à concevoir.
L'offre présente est essentiellement orientée vers des cycles de conception de circuits intégrés. Les méthodes de partitionnement se basent en particulier sur une connaissance a priori de

l'architecture cible. Celle-ci peut être constituée d'un ensemble d'IP interconnectées, mais la granularité de l'architecture reste "gros grain" et relativement figée.
Au niveau des méthodes et des formalismes de spécifications, les offres actuelles sont soit à très bas niveau (outils propriétaires pour le reconfigurable) et demandent une approche encore très manuelle du Co-développement, soit à des niveaux plus abstraits (exemple l'environnement VCC de Cadence) offrant des possibilités de partitionnement "gros grain". Quelques acteurs du domaines proposent des approches plus globales (Environnement ArchiMate d'Arexsys) mais leurs performances en partitionnement sont encore peu satisfaisantes et restent ciblés vers la conception de circuits intégrés ASIC par intégration d'IP.
Les atouts de notre approche résident dans la forte interaction entre le modèle de l'implémentation matérielle et l'outil d'estimation et d'optimisation, contrairement aux approches actuelles, dans lesquelles la méthodologie de partitionnement et les outils y afférents sont dédiés pour une implémentation matérielle figée. De plus les outils existant n'apportent pas de réponses concernant l'insertion des technologies reconfigurables dans les systèmes embarqués hétérogènes (processeurs + reconfigurable).

Ruptures technologiques :

La maîtrise des architectures reconfigurables est stratégique dans la perspective des applications à venir. Les verrous technologiques et méthodologiques à lever sont nombreux. Dans le cadre du projet EPICURE nous nous intéressons à l'un de ceux qui est parmi les plus importants. En l'occurrence, le développement d'une méthode et d'outils aptes à rendre le partitionnement des tâches plus automatique.

Organisation du projet :

Les cinq sous-projets composant le projet EPICURE permettent d'appréhender la problématique du partitionnement logiciel/matériel selon tous ses aspects.

Un premier sous-projet dirigé par le CEA-LETI s'attachera à définir précisément toute les données du problème tant au niveau des méthodes et du logiciel qu'au niveau de la technologie et des applications. En particulier un modèle d'architecture reconfigurable sera proposé par le CEA-LETI. Les travaux de développements au cœur même de ce projet seront menés dans deux sous-projets qui considérerons chacun un aspect important dans la méthode de partitionnement :

Dans le sous projet 2, le LESTER définira les méthodes et développera les outils permettant de réaliser les étapes de modélisation et d'estimation du flot de co-design pour le modèle d'architecture reconfigurable qui aura été défini par le CEA-LETI.

Dans le sous projet 3 l'I3S développera une méthode et un outil de partitionnement qui s'interfacera avec les estimateurs. Ces deux sous-projet seront évidemment menés en étroite collaboration.

Le sous-projet 4 d'intégration et de validation coordonné par THOMSON-CSF s'attachera à vérifier la pertinence de la méthode et des outils qui auront été développés au cours des sous-projets précédents sur des cas d'applications réalistes. En préalable ce sous projet étudiera aussi les passerelles nécessaires à une bonne intégration dans les flots de conception existants.

Enfin un sous-projet 5 clôturera le projet EPICURE et sous la responsabilité de la société SIMULOG évaluera les perspectives industrielles et commerciales des méthodes et outils qui auront été développés.

Retombées du projet :
Les retombées d'un tel projet peuvent être nombreuses. En terme de poursuites des travaux : d'un côté les travaux de recherches et de développements entrepris permettront de préciser les besoins en recherches supplémentaires ainsi que les verrous scientifiques ou technologiques qu'il conviendra de lever à l'avenir, d'un autre côté les évaluations applicatives et les analyses de performances nous suggérerons les suites industrielles et commerciales qu'il conviendra d'entreprendre.
Le domaine des composants reconfigurable est à l'heure actuelle en pleine émergence. Chaque industriel du domaine propose son architecture, ses solutions de configuration et son API de programmation voire même son propre langage de description (Altera). Il nous paraît opportun de proposer une réflexion rassemblant tous les acteurs du domaine sur les possibilités de convergence et de standardisation. Ce besoin d'un minimum de standardisation est essentiel dans le domaine des technologies reconfigurables afin de permettre un développement solide des méthodes de conception associées. Notre partenariat au travers de ses participations a de nombreuses instances normatives tels VSIA et OMG pourrait ainsi participer ainsi á la création d'un groupe de travail sur la normalisation dans le domaines des composants reconfigurables.

Résumé

De nombreuses applications, en particulier en traitement des images, ont des temps d'exécution variables en fonction de la nature des données à traiter. La mise en œuvre des traitements de bas niveau en traitement d'images (par exemple la détection de contours, étiquetage) dans des architectures embarquées utilise généralement des systèmes spécialisés.

D'autres part, l'apparition de nouvelles architectures basées sur la programmation de circuits matériels, tels que les composants programmables, est un facteur important dans l'évolution des systèmes électroniques modernes. Les récentes évolutions des différentes familles autorisent aujourd'hui l'intégration de systèmes de plus en plus complexes avec des contraintes de performances de plus en plus fortes. La flexibilité offerte par ce type de technologie fait des FPGAs (Field Programmable Gate Arrays) une cible architeturale promise à un bel avenir.

L'objectif du travail présenté dans ce mémoire consiste à proposer une méthode de partitionnement logiciel/matériel qui permet de répartir les traitements à temps d'exécution variables sur une architecture formé par un processeur connecté à un circuit reconfigurable.

L'approche présentée est basée sur un algorithme génétique et s'intègre dans un flot de conception conjointe logiciel/matériel.

Mots-clés : ***partitionnement, graphe conditionné, architecture reconfigurable, paramètres de corrélation, système enfouis.***