Contribution aux architectures adaptatives : Etude de l'efficacité énergétique dans le cas des applications à parallélisme de données

Xun Zhang

▶ **To cite this version:**

Xun Zhang. Contribution aux architectures adaptatives : Etude de l'efficacité énergétique dans le cas des applications à parallélisme de données. Autre. Université Henri Poincaré - Nancy 1, 2009. Français. NNT : 2009NAN10106 . tel-01748353

HAL Id: tel-01748353
https://hal.univ-lorraine.fr/tel-01748353

Submitted on 29 Mar 2018

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

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

AVERTISSEMENT

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

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

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

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

LIENS

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

Nancy-Université
Université Henri Poincaré

U.F.R. Sciences et Techniques, Mathématiques, Informatiques, Automatique
École Doctorale Informatique Automatique Electronique Electrotechnique Mathématiques
Département de formation Doctorale Électronique Électrotechnique

Contribution aux architectures adaptatives : étude de l'efficacité énergétique dans le cas des applications à parallélisme de données

THÈSE

présentée et soutenue publiquement le 15/09/09

pour l'obtention du

Doctorat de Nancy Université – Nancy 1

(spécialité Systèmes Électroniques)

par

Xun ZHANG

Composition du jury

Président :	Olivier SENTIEYS	Professeur, Université Rennes 1
Rapporteurs :	El-Bay BOURENNANE	Professeur, Université de Bourgogne
	Lionel TORRES	Professeur, Université Montpellier 2
Examinateurs :	Serge WEBER	Professeur, UHP (Directeur de thèse)
	Hassan RABAH	Maître de Conférences HDR, UHP (Co-directeur de thèse)
	Yves BERVILLER	Maître de Conférences, UHP

Laboratoire d'Instrumentation Électronique de Nancy
Faculté des Sciences - 54506 Vandoeuvre-lès-Nancy

Remerciement

Je remercie en tout premier lieu mes directeurs de thèse **M. Serge WEBER** et **M. Hassan RABAH**, pour la confiance qu'ils m'ont témoignée tout au long de ces années de travail, leurs participations, leurs encouragements quotidiens et leurs précieux conseils au niveau scientifique.

J'adresse mes remerciements à **M. Oliver SENTIEYS**, professeur, ENSSAT, Lannion, qui m'a fait l'honneur de présider le jury. Je voudrais exprimer ma gratitude à toutes les personnes qui m'ont fait l'honneur de participer à ce jury de thèse : **M. Lionel TORRES**, professeur à l'université de Montpellier 2 et **M. El-bay BOURENNANE**, professeur à l'Université de Bourgogne, pour l'attention qu'ils ont accordée à la lecture de ce mémoire de thèse et pour avoir bien voulu en être les rapporteurs.

Mes remerciements vont en particulier à **M. Yves BERVILLER**, Maître de conférences à Nancy Université qui lors de nombreuses discussions m'éclaira sur mon travail et eu la gentillesse de corriger mon manuscrit et de me faire part de ses commentaires.

Mes remerciements les plus chaleureux à tous les membres du laboratoire LIEN. Avec vous, la bonne humeur est indissociable du bon café, du thé et autres douceurs.

Mes remerciements vont également à ma famille, en particulier mes parents et ma soeur, cette thèse n'aurait jamais vu le jour sans leur soutien inconditionnel. Je ne saurais remercier assez **HUI** qui m'a encouragé tout au long de mes études et qui m'a apporté sont réconfort dans ma thèse et les moments difficiles.

Table des matières

Chapitre 2 Méthode et stratégie d'adaptation

Chapitre 3 Étude et validation expérimentale

Table des figures

Introduction générale

Le circuit configurable est apparu au début des années 80, beaucoup d'évolutions importantes ont eu lieu en ce qui concerne l'idée même de reconfigurer dynamiquement un composant. Une architecture reconfigurable est une architecture dont les ressources peuvent être redéfinies, complètement ou partiellement, pour réaliser un traitement donné. La reconfiguration peut être statique ou dynamique, dans le sens où l'architecture est reconfigurable au fil de l'exécution. Le travail s'inscrit en majeure partie dans le cadre de la reconfiguration partielle dynamique.

Aujourd'hui peu d'architectures reconfigurables sont effectivement réalisées sur silicium ce qui met les utilisateurs soucieux de cibler une architecture flexible et évolutive, face au manque de composants susceptibles de mettre en œuvre leurs applications.

La problématique qui motive ce travail de thèse peut se résumer ainsi : face aux multi-contraintes de l'application et variations dans le temps, est-il possible de développer une méthode permettant de choisir et d'adapter l'organisation d'une architecture respectant la contrainte temps réel tout en minimisant la consommation énergétique ?

Les circuits reconfigurables actuels, notamment les permettent d'intégrer un grand nombre de ressources matérielles à granularité hétérogène allant de plusieurs microprocesseurs, d'opérateurs mathématiques jusqu'aux portes logiques élémentaires en très grand nombre [CH02, BBW+01]. Ceci amène à se poser la question sur la façon de concevoir les systèmes pour couvrir l'ensemble ou une partie des besoins (une meilleure utilisation des ressources, une puissance réduite, et une bonne vitesse d'exécution). Une possibilité est de partitionner une application en tâches matérielles et logicielles avant la synthèse, et de placer les tâches matérielles sur un accélérateur spécifique et les tâches logicielles sur un processeur à usage général. Certaines techniques de gestion [QSN07, LM05, NMB+03, MNC+03, RMVC05] sont proposées pour permettre aux tâches multiples d'être programmées de manière optimale.

Une autre possibilité est d'utiliser les SoPC. Dans ce cas de multiples noyaux fonctionnels et reconfigurables sont définis dans le système. Avec la technique de reconfiguration partielle et dynamique [Rab97], le dispositif reconfigurable permet de confi-

gurer une partie de logique de FPGA tandis que le système continue de fonctionner. En outre, comme mentionné ci-dessus, les plate-formes FPGAs actuelles pourraient également inclure le noyau de processeur et les accélérateurs spécifiques à l'application. Différentes architectures de ce type ont déjà été proposées aussi bien par des laboratoires universitaires que par des fabricants de SOC [UHGB04, AZB+07].

Pour ces deux possibilités, les solutions sont basées sur une topologie globale fixe et une architecture hétérogène associant un ensemble de GPP, DSP, et de processeurs spécifiques et d'architecture reconfigurables en grain fin ou gros grain. La contribution de ces dernières recherches est principalement focalisée sur les méthodes de gestions efficaces de ressources matérielles et logicielles. l'auto-adaptation demande non seulement une architecture hétérogène, mais aussi une évolution partielle et dynamique de sa topologie selon l'application exécutée et l'environnement de l'application.

L'objectif de ce travail de thèse est d'explorer la possibilité de maintenir des contraintes de temps d'exécution et de minimiser la consommation grâce à la reconfiguration dynamique et à la gestion dynamique de fréquence d'exécution dans un contexte d'applications où la quantité de donnée à traiter est variable.

Plan du mémoire

Le mémoire de thèse est organisé de la manière suivante : Le premier chapitre dresse un état de l'art des architectures adaptatives. Au niveau matériel, l'adaptation consiste en une reconfiguration partielle de l'architecture. La reconfiguration matérielle augmente la flexibilité de l'architecture en permettant l'implémentation d'un grand choix de fonctions. Dans ce chapitre, nous analysons les différentes possibilités d'adaptation des systèmes sur puce (SoC). Nous étudions ensuite la structure d'un SoC adaptatif. En mettant en évidence ses principaux éléments constitutifs. Nous complétons la caractérisation des SoCs adaptatifs par la présentation de différentes architectures et la comparaison des différentes structures.

Le deuxième chapitre détaille la méthodologie d'auto-configuration proposée. Il débute par une mise en évidence des différentes formes d'adaptations possibles dé-

crites dans le chapitre précédent. Dans notre travail, nous nous concentrons sur deux méthodes : DRP et DFS. La reconfiguration partielle (DRP) nous permet de changer le niveau du parallélisme de tâche. La gestion dynamique de fréquence (DFS) de traitement permet d'obtenir la fréquence de traitement juste suffisante pour respecter la contrainte de temps. Le modèle d'un système adaptatif ainsi que ses éléments principaux sont présentés. Ensuite, nous montrons qu'il est possible d'améliorer le rendement des implantations à l'aide de notre méthode (adaptation du niveau de parallélisme et de la fréquence de traitement) tout en respectant la contrainte de temps de traitement.

Le troisième chapitre concerne la réalisation d'un prototype basé sur le modèle d'architecture décrit dans le chapitre précédent. La plate-forme expérimentale et le flot de conception utilisés seront d'abord présentés. Nous présentons ensuite les résultats issus de l'implémentation de fonctions clés de la compression d'images selon la norme JPEG2000 et analysons le comportement de notre architecture sur cette application. Les résultats expérimentaux sur le compromis efficacité énergétique et performance dans le cas de l'adaptation aux variations de quantité de données à traiter seront présentés et discutés.

La dernière partie de ce mémoire dresse un bilan du travail de thèse ainsi que les nombreuses perspectives ouvertes par ces travaux.

Chapitre 1

Supports pour les architectures adaptatives

Sommaire

1.1 Introduction

Dans ce chapitre, nous allons présenter une étude consacrée aux supports pour les systèmes sur puce (SoC) adaptatifs. Un SoC adaptatif (ou auto-adaptatif) dispose (1) des mécanismes logiciels capables de reprogrammer les fonctions de traitement de son architecture (2) des mécanismes matériels capables de redéfinir complètement ou partiellement des caractéristiques de son architecture, de manière statique ou dynamique. Dans un premier temps, nous analyserons les différentes possibilités d'adaptation des SoC. Nous étudierons ensuite la structure d'un SoC adaptatif, en mettant en évidence ses principaux éléments constitutifs. Nous compléterons la caractérisation des SoCs adaptatifs par la présentation de différentes architectures et une comparaison de leurs différentes structures. Enfin, les méthodes d'adaptation seront étudiées.

1.2 Caractérisation et classification des SoCs adaptatifs

1.2.1 Définition générale

Nous considérons ici l'adaptation comme le processus de *réutilisation* et de *continuité de service* d'un système sur puce lorsque les besoins ou les conditions changent. L'adaptation permettra à la fois d'étendre l'utilisation d'un SoC pour d'autres applications et de maintenir et assurer la continuité de fonctionnement. L'adaptation est envisageable pendant la phase de conception (*modèle adaptable*) et pendant l'utilisation (*système adaptable*). Le modèle adaptable permet en particulier la réutilisation du même modèle auquel des modifications sont apportées pour réaliser des *variétés* ou une familles de SoC qui ne sont pas forcément adaptables. Nous appelons *SoC adaptable* le produit final qui peut être adapté aux besoins et aux conditions d'utilisation par l'utilisateur. On définira un *SoC adaptatif* comme le système qui est capable de détecter les modifications de son environnement et de s'adapter en conséquence.

On peut distinguer deux types d'adaptations. La première est l'*adaptation spécifique* où les adaptations particulières sont prévues et réalisées en ajoutant ou supprimant des modules ou bien en exploitant de manière intelligente les potentialités de cer-

taines fonctions. La seconde est l'*adaptation générale* qui doit en particulier répondre au problème de changement imprévisible de l'environnement. Cette adaptation est intéressante et convient aux systèmes plus complexes et rejoint la problématique d'auto-organisation qui n'est pas l'objet de cette thèse.

Le sujet de l'adaptation est largement étudié dans plusieurs disciplines aussi diverses que l'urbanisme avec les architectures adaptatives et l'informatique avec les logiciels adaptatifs. Dans toutes ces études, parmi les éléments clef de l'adaptabilité on trouve toujours la *modification*et la *modularité*. Ainsi, pour qu'un système soit plus adaptable, il faut autoriser et faciliter les modifications. Dans le cas de l'adaptation spécifique, un SoC doit être capable d'ajouter ou de supprimer des modules ou de modifier certains modules et d'assurer la communication entre eux en fonction des besoins et des conditions d'utilisation. Ces besoins et conditions doivent être introduits dans le flot de conception et utilisés comme contraintes qui doivent donc être prévues.

Les systèmes adaptatifs doivent réagir aux besoins de l'application mais également aux conditions de leurs fonctionnement internes ou externes. Ils doivent donc être capables de détecter l'état de l'environnement. La figure 1.1 représente un système adaptatif en relation avec son environnement. Le module d'adaptation surveille simultanément l'état du système et les conditions extérieures.

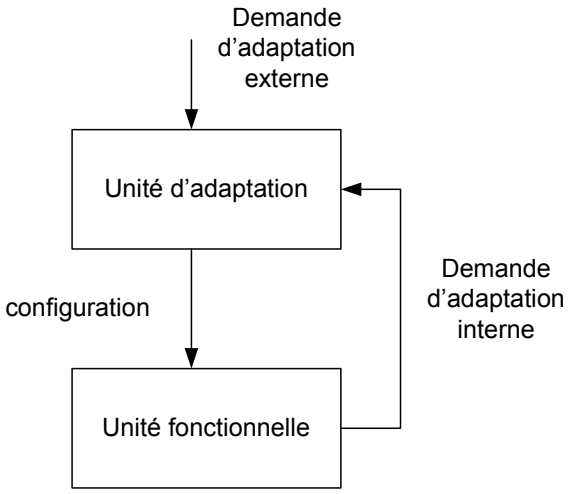

FIGURE 1.1 – Le système adaptatif et son environnement

L'action d'adaptation étant de modifier la structure du système, cette modification peut être effectuée au niveau logiciel ou matériel définissant ainsi les deux niveaux d'adaptation.

1.2.2 Niveau d'adaptation

1.2.2.1 Adaptation au niveau logiciel

L'adaptation au niveau logiciel est réalisée par modification du programme s'exécutant sur un processeur en fonction de contraintes de l'application. L'objectif de la modification est d'optimiser le programme selon un certain nombre de critères dans un environnement où les paramètres de ces critères sont connus et fluctuent au cours de l'exécution du programme.

Plusieurs composants logiciels prouvent être assemblés pour une application. Ces composants logiciels permettent de définir des tâches plus ou moins complexes. Un programme peut être modifié par l'ajout, la suppression ou la modification d'une ou plusieurs tâches pour s'adapter à un nouveau service ou à un nouvel environnement.

Le travail [MLw98] propose une méthodologie pour la construction de programmes capables de s'adapter au changement de contraintes de l'application. Cette méthodologie est basée sur le choix des conceptions d'architecture de l'application prédéfinies. Un contrôleur est souvent installé dans le système et différents programmes prouvent être programmés dans le processeur. Dans les travaux [KBW03, BFY+02, NB02] un modèle à composants où chaque composant est constitué d'une partie contrôle et d'une partie fonctionnelle est proposé. La partie contrôle gère le cycle de vie du composant logiciel (démarrer, verrouiller, mis à jour, etc.). D'autres approches traitent l'aspect réutilisation des composants logiciels. Ces composants logiciels sont programmés dynamiquement par la partie contrôle. Évidemment, la reprogrammation est plus flexible que la modification d'architecture matérielle. Cependant, la solution logicielle est moins performante en terme de traitement et de consommation par rapport à la solution matérielle basée sur les ASIC et FPGA.

1.2.2.2 Adaptation au niveau matériel

L'adaptation au niveau matériel est possible grâce aux architectures reconfigurables qui offrent la flexibilité des architectures programmables tels les processeurs à usage général et les performances des circuits ASICs [Har06]. Le concept de reconfiguration a largement été étudié depuis plusieurs années et reste encore un domaine de recherche actif. Plusieurs aspects de la reconfiguration ont été abordés et plusieurs architectures ont été proposées. Toutes ces architectures sont basées sur l'utilisation de blocs de calcul et d'éléments de communications reconfigurables. Les principales différences entre les architectures existantes résident dans les granularités des blocs de calcul qui peuvent être fin, épais ou hétérogènes aussi bien au niveau éléments de calcul que de communication [DeH96, Bob05].

Dans les architectures à grain fin (FPGA classique), la reconfiguration se fait au niveau bit permettant de réaliser des opérations de traitement ou de contrôle dédiées. Cependant, elles restent inefficaces dans le cas de certaines applications et posent le problème de la taille de configuration qui rend la reconfiguration dynamique complexe. Dans les architectures à grains épais, les interconnexions et les opérateurs sont configurables, mais travaillent au niveau arithmétique. Ceci a pour effet d'améliorer les performances sur les traitements arithmétiques, au détriment des optimisations au niveau bit. L'avantage des architectures à gros grain est de réduire la taille des configurations permettant d'envisager la reconfiguration dynamique de manière efficace.

Les FPGAs actuels en revanche offrent une alternative intéressante. En effet, ils intègrent des ressources matérielles allant d'une simple LUT, permettant de réaliser des fonctions au niveau bit, jusqu'à quelques microprocesseurs en dur en passant par des opérateurs plus ou moins complexes tels que les additionneurs, les fonctions multiplication accumulation, etc. Par ailleurs, ces circuits sont conçus et organisés sous forme de régions dont ont peut contrôler la fréquence et la tension d'alimentation et offrent également des possibilités de reconfiguration partielle. Il est ainsi possible de modifier et d'agir de manière dynamique et partielle sur les paramètres tension et fréquence en plus de la logique et des interconnexions.

L'idée de base de la reconfiguration dynamique est d'exécuter un ensemble de tâches matérielles sur un ensemble de surfaces logiques. Le nombre de tâches matérielles doit nécessairement être supérieur au nombre de surfaces logiques. Dans cette optique, la reconfiguration dynamique est vue comme un processus d'optimisation de surface. Les tâches matérielles sont alors obtenues par un découpage de l'application sous forme de partitions temporelles qui doivent être exécutées séquentiellement [KVGO99, TBWB03]. Les surfaces sur lesquelles doivent s'exécuter ces partitions sont le résultat du découpage spatial du circuit reconfigurable. Le processus de reconfiguration dynamique est aussi un processus d'allocation spatial et temporel des tâches d'une application. Cet aspect de la reconfiguration dynamique a largement été exploré depuis plusieurs années et reste encore un sujet assez ouvert. Parmi les problématiques abordées dans ce cas, il y a les aspects méthodes et outils pour le partitionnement spatial et temporel, la défragmentation, l'ordonnancement et exécutifs temps réel [RMVC05] [TSMM06]. L'architecture reconfigurable est une opportunité pour la réalisation d'un système adaptatif. Au niveau matériel, l'adaptabilité du système est obtenue par la configuration du circuit.

1.3 Adaptation par reconfiguration matérielle (RM)

Nous considérons qu'une application peut être constituée par un certain nombre de tâches. Chaque tâche peut représenter une opération ou un ensemble d'opérations de l'application. La reconfiguration matérielle (RM) nous offre une solution d'adaptation via la modification de la structure du système. La stratégie d'adaptation est le remplacement d'une application complète par une autre , ou le changement d'une partie de l'application pour répondre à des besoins de l'application.

1.3.1 Reconfiguration statique (RS)

La reconfiguration statique(RS) est l'approche la plus simple et la plus commune de la RM pour implémenter une application avec de la logique programmable. La RS

concerne le changement de la fonctionnalité d'un ou de plusieurs composants d'un système indépendant de l'exécution d'une application. C'est-à-dire, le contenue de configuration reste statique pendant toute la vie de l'application.

La RS est similaire à l'utilisation de l'ASIC pour l'accélération d'applications. D'un point de vue application, il n'y a pas de grandes différences dans l'utilisation d'un FPGA ou d'un ASIC. Au niveau du coût système, le FPGA présente un grand avantage par rapport à l'ASIC. Pour une même surface de silicium, le FPGA peut être réutilisé plusieurs fois pour des applications différentes, ce qui n'est pas le cas pour un ASIC.

Le trait distinctif de la configuration statique est qu'elle consiste en un seul système. Avant de commencer une opération d'une fonction, les ressources reconfigurables sont chargées avec leurs configurations respectives. Une fois l'opération débutée, les ressources reconfigurables resteront dans cette configuration pendant toute l'opération de l'application. Ainsi, les ressources matérielles restent statiques pendant toute la durée de vie de la conception. La RS ne peut pas répondre aux besoins dans le domaine des systèmes adaptatifs. Concrètement, la RS s'emploie pour l'initialisation du prototype du système.

1.3.2 Reconfiguration dynamique (RD)

Ce mode de reconfiguration est basé sur la RS. Il offre la possibilité de changer de fonctionnalité pendant l'exécution d'une application pour reconfigurer le système en exécution. L'avantage de la RD a été plusieurs fois démontré [MB98, SD99, DLRN00]. Les caractéristiques principales de la RD sont : flexibilité, performance et coût par rapport à la RS.

Flexibilité : La RD supporte la reconfiguration au cours d'exécution. Ainsi, la durée de vie d'une configuration n'est plus celle de l'application entière. Une application peut être coupée en plusieurs parties qui seront exécutées avec un ordre de configuration. Donc la RD est plus adaptée à la conception de systèmes adaptatifs que la RS.

Performance : Par rapport la RS, la RD propose plus de moyens pour optimiser les performances de calcul grâce à plusieurs versions de fonctionnalité d'un compo-

sant qui peut être changé au cours d'exécution. Elle peut optimiser la performance ou ajuster la fonction du système en utilisant le circuit le plus optimisé qui est chargé ou déchargé dynamiquement pendant le traitement du système. Cela est la principale différence par rapport à la RS.

Coût : l'intérêt majeur de la RD est la possibilité d'une reconfiguration illimitée d'implémentation de plusieurs fonctions différentes avec le même matériel. L'un des principaux inconvénients des FPGAs par rapport aux ASICs est qu'ils sont moins denses en nombres de portes utiles et ont des fréquences de fonctionnement réduites. La RD est donc un moyen d'améliorer certains de ces handicaps en reconfigurant le circuit lorsque celui-ci a montré un accroissement des capacités de fonctionnement [BDHM05]. En utilisant la RD, une petite surface de logique programmable peut supporter une grande application. Le coût de circuit est donc réduit grâce à l'utilisation de la RD. Il y a deux aspects de la RD : la reconfiguration totale dynamique(RTD) et la reconfiguration partielle dynamique(RPD).

1.3.2.1 Reconfiguration total dynamique (RTD)

Dans ce type de reconfiguration, la totalité du contenu de la mémoire de reconfiguration du FPGA est remplacée par de nouvelles données de configuration. Ici, le terme "total" signifie la taille du FPGA. La figure 1.2 illustre ce mode de reconfiguration. Nous considérons un traitement constitué de trois segments d'un ou d'un groupe d'opérateur O_1, O_2 et O_3. La reconfiguration totale dynamique consiste en l'implémentation de ces trois opérateurs suivant leurs dépendances les uns par rapport aux autres en reconfigurant entièrement le système à chaque nouvelle étape.

L'avantage de la RTD est de supporter une grande flexibilité par rapport à la reconfiguration statique. Cependant, cette méthode est toujours supposée comme étant une reconfiguration entière du FPGA. Le seul problème principal pour l'utilisation de la RTD est tout d'abord d'arriver à un équilibre entre le partitionnement temporel de l'algorithme et la surface de FPGA.

Actuellement, l'évolution technologique des FPGAs permet d'implémenter de plus en plus d'applications dans un même circuit et ceux-ci peuvent supporter la reconfigu-

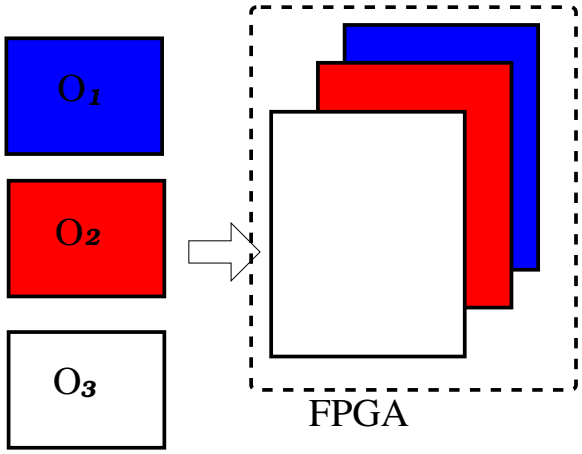

FIGURE 1.2 – Reconfiguration dynamique totale

ration partielle au cours d'exécution. Le problème de la taille des FPGAs est de moins en moins contraignant.

1.3.2.2 Reconfiguration partiel dynamique(RPD)

Comme nous l'avons indiqué précédemment, l'inconvénient de la RTD est la nécessité de trouver un équilibre entre la taille des partitions. S'il n'est pas possible d'uniformiser la taille des partitions de l'application, l'utilisation des ressources du FPGA est inefficace, ce qui est une problématique. Pour résoudre cet inconvénient, il existe une technique, appelée reconfiguration partielle dynamique (RPD). La RPD supporte une approche de reconfiguration plus flexible que celle de la RTD. Elle peut reconfigurer une partie du matériel sans l'interruption des fonctions des autres parties. La figure 1.3illustre ce mode de reconfiguration. Précédemment, nous avons considéré un traitement constitué de trois opérateurs O_1,O_2 et O_3. La reconfiguration partielle dynamique consiste à implémenter ces trois opérateurs suivant un ordre d'ordonnancement.

Par rapport à la RTD, la RPD ne consomme pas toute la surface du FPGA et permet de gérer efficacement cette surface. De plus, c'est une bonne alternative pour les algorithmes qui ne peuvent pas être partitionnés à des segments temporels liés avec la taille de la plate-forme entière. Parce que nous n'avons plus besoins de partitionner une ap-

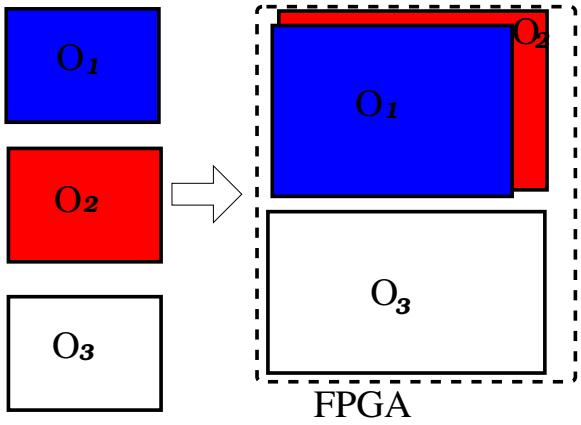

FIGURE 1.3 – Reconfiguration partiel dynamique

plication en segments temporels en liant absolument la taille de la plate-forme entière. Nous pouvons donc nous concentrer sur l'ordonnancement des tâches reconfigurables pour avoir l'efficacité de l'implantation. De plus, l'optimisation de la contrainte de reconfiguration (le temps de reconfiguration, l'allocation de l'application sur la puce) avec des méthodes de RPD est plus efficace que RTD grâce à certaines méthodes de RPD (masquant le temps de reconfiguration, l'ordonnancement de tâche, etc.). La RPD permet d'optimiser l'équilibre de performance et de consommation du système plus facilement. La RPD est utilisée pour implanter des tâches d'application sur la base de l'architecture FPGA SRAM. Plusieurs travaux de recherches ont exploité cet avantage dans la conception de système reconfigurables afin d'obtenir une bonne réutilisabilité [BBD05, SB06, BBD06, SEJN06, JPGCP04, BFY+02, HG07, AKM06].

Banerjee et al. [BBD05] présente une approche qui sélectionne la granularité du parallélisme de données pour maximiser les performances d'une application s'exécutant dans une architecture RPD. Cette approche choisit entre le niveau de parallélisme et les contraintes de temps de chaque niveau. Comme dans l'exemple qui est illustré dans la figure1.4, trois granularités différentes se présentent pour différents niveaux de parallélisme, et aussi différents temps d'exécution. La figure 1.4-a est une chaîne simple avec deux tâches. Supposons que nous ayons à notre disposition les ressources suffisantes afin d'exécuter simultanément trois copies de la même tâche T_1 ou 2 copies de la tâche T_2, (1.4-b) et (1.4-c) montrent des configurations possibles du graphe de tâches

après cette transformation.

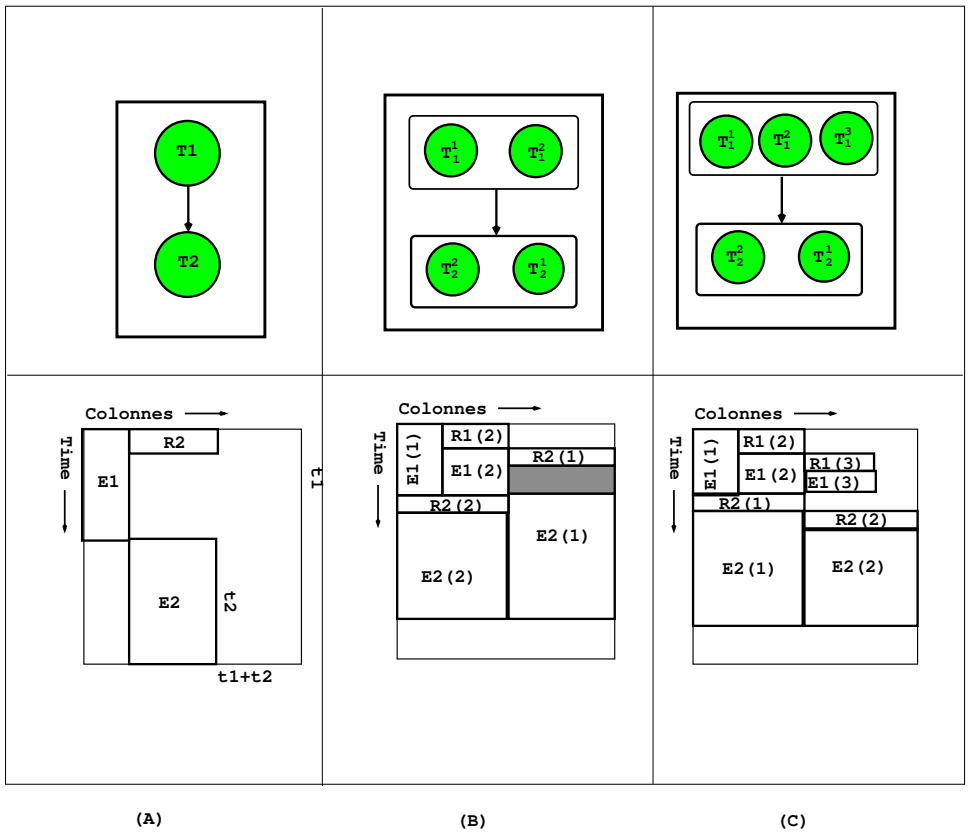

FIGURE 1.4 – La granularité du parallélisme pour maximiser la performance d'application

Cependant, la transformation est sur l'architecture RPD où chaque copie de tâche ajoute une contrainte de reconfiguration significative. Donc, la transformation a besoin d'être guidée par une sélection de la granularité de parallélisme correcte. Mais, cette méthode n'a pas considéré la réutilisabilité de tâches.

Singhal et al. [SB06] mentionne le sujet de la réutilisation de composant. L'objectif est d'explorer le chevauchement potentiel entre deux tâches données avec une série de composants communs afin de réduire la latence de reconfiguration dynamique. Mais l'ordre de dépendance ou d'indépendance de tâches n'est pas considéré si la zone reconfigurable a besoin de mettre en place une nouvelle tâche.

Banerjee et al. [BBD06] a proposé une approche qui considère le problème de partitionnement matériel/logiciel au niveau tâche. Dans cette approche, toutes les tâches

ont deux versions, matérielle et logicielle. L'auteur propose une solution d'ordonnancement au cours d'exécution pour avoir le plus de performances. Comme indiqué sur la figure 1.5, une application est représentée par un graphe de tâches 1.5-a. Les temps d'exécution de chaque tâche pour les deux différentes versions et la surface occupée sur puce sont mesurés (figure1.5-b). En utilisant ces paramètres, la solution optimisée de placement de tâche peut être trouvée (figure 1.5-c).

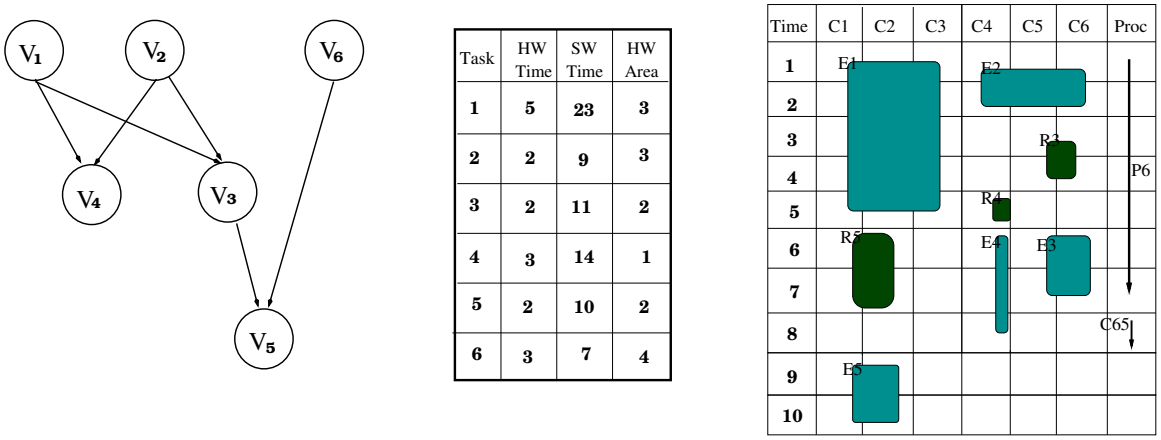

FIGURE 1.5 – La granularité du parallélisme pour maximiser la performance

1.4 Gestion dynamique des fréquences et d'alimentations

La modification dynamique des fréquences de fonctionnement et des tensions d'alimentation permet d'obtenir un équilibre efficace entre performance et consommation du système. La gestion dynamique des fréquences et alimentations (Dynamic Frequency and Voltage Scaling(DFVS)) correspond à l'association de la gestion des fréquences (DFS [1]) et gestion dynamique d'alimentation (DVS [2]). C'est une approche utilisée dans les ASIC ou dans un système basé sur le processeur pour ajuster la consommation d'énergie en temps réel. Les travaux [PBB98, TTR00, LLHC04, SN05, TBB98, TTR00] ont traité la configuration de la fréquence d'exécution et l'alimentation pour réduire la consommation d'énergie. Cependant sur les FPGAs, cette méthode n'est pas aussi fa-

1. Dynamic Frequency Scaling
2. Dynamic Voltage Scaling

cile à utiliser en raison de la structure spéciale du réseau interne de l'horloge sur FPGA et surtout la méthode DVS est très limitée sur FPGA. Actuellement, des fabricants de FPGA telle que xilinx proposent des cartes évaluation pour support DVS en utilisant un composant externe FPGA pour ajuster la tension de FPGAs [3]. Il existe également des recherches académiques, qui s'intéressent à cette problématique [LLHC04, LL05, PHBB07].

Gestion dynamique de fréquence La fréquence de fonctionnement influence directement la vitesse de calcul du circuit. Beaucoup de méthodes d'optimisations reposent sur le changement dynamique de la fréquence de fonctionnement comme les méthodes DFS(*Dynamique Fréquency Scaling*). La gestion dynamique de fréquence (DFS) est déjà connue dans les microprocesseurs et ASIC [DPF+00]. Elle est largement utilisée pour optimiser de la consommation d'énergie et les performances de calcul [DPF+00, SN05].

Gestion dynamique d'alimentation L'adaptation dynamique de la tension d'alimentation, appelée aussi : gestion dynamique d'alimentation [4], permet à un processeur de modifier dynamiquement sa tension d'alimentation et apporte un gain très significatif de la consommation. Beaucoup de méthodes d'optimisation [UH95, TBB98] reposent sur le changement dynamique de la tension d'alimentation V_{dd} pendant le fonctionnement du système. Cela suppose que le processeur soit capable d'ajuster à la demande sa tension et sa fréquence, et donc sa vitesse de traitement et sa consommation. Dans la littérature, il y a beaucoup de travail sur la gestion dynamique d'alimentation (DVS) [TBB98, BB95, KST01]. Nous donnons ci-après quelques exemple d'utilisation DVFS.

lpARM

Ce processeur [PBB98, TTR00], est une version de Berkekey de l'ARM8 conçu en technologie CMOS 0.6 um. Ses performances sont estimées à 6Mips à 5MHz-1.2v et 85Mips à 80MHz-3.8v. Lorsqu'il est en mode "IDLE"(en veille), le lpARM consomme 0.8mW et prend un cycle pour se réveiller. En ajustant la tension d'alimentation, le

3. info sur site : www.xilinx.com
4. Dynamic Voltage Scaling

système permet, non seulement fonctionner dans un état performante (80MHz-3.8v), mais aussi économiser la puissance consommé pendant le temps veille (5MHz-1.2v).

FPGA à multi-tension

Un autre exemple basé sur l'architecture FPGA est proposé dans [LLHC04]. Les multiples tensions (V_{dd}) ou tensions de seuil(V_t) peuvent aussi améliorer l'efficacité énergétique. La réduction de V_{dd} limite la consommation de puissance dynamique, alors que l'augmentation de V_t limite la consommation de puissance statique. Cette architecture avec $dual - V_{dd}$ et $dual_V t$ obtient $13,6\%$ et 14% de réduction de la puissance totale en moyenne pour les circuits combinatoires et séquentiels, respectivement [LLHC04]. Un exemple de sa structure est illustré dans la figure 1.6. L'avantage de cette architecture est d'optimiser la performance et l'efficacité d'énergie sur FPGA avec la technique multitension. La technique $muti\ V_{dd}/multi\ Vt$ ont été employées sur ASIC [UH95,CSS01]. Cette architecture FPGA à multi-tension n'existe pas encore dans les plate-formes FPGA commerciales à cause de la limite des outils CAO et des structures de FPGAs.

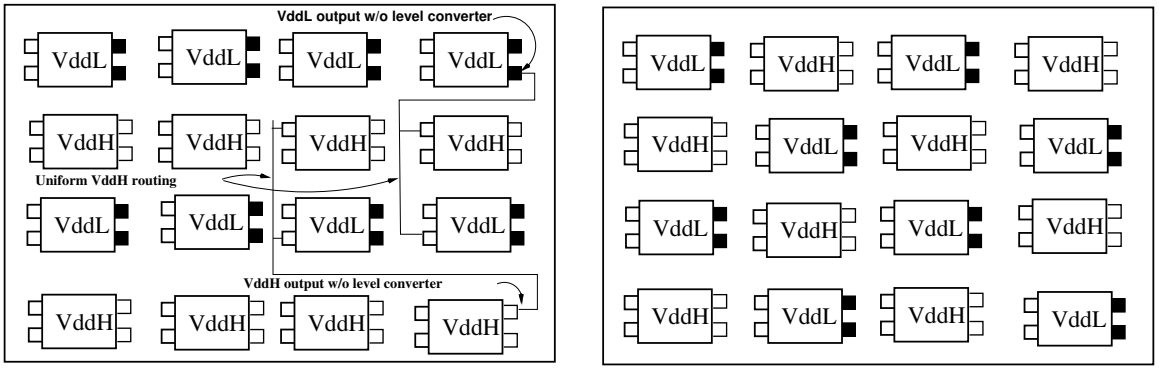

FIGURE 1.6 – Deux différents modèles layout pour un dual-Vdd FPGA [LLHC04]

DVS dans les FPGAs du commerce

Une méthode pour supporter DVS sur les FPGAs commerciaux est proposée dans [LL05]. La tension de FPGA peut être gérée en temps réel. Pour maîtriser efficacement la réduction de la tension d'alimentation, un circuit spécial,LDMC(Logic Delay Measurement Circuit) est utilisé pour aider à ajuster dynamiquement la tension du FPGAs. La figure 1.7 illustre l'architecture du système avec l'implémentation de DVS. Dans ce

système, la tension est réglée par circuit externe du FPGAs. Parce qu'il n'y a pas encore de solution permettant au FPGA d'ajuster la tension par lui-même. Par conséquent, la technique DVS est souvent utilisée dans les microprocesseurs modernes et ASIC. Il y a moins d'exemples pour l'utilisation dans les FPGAs commerciaux.

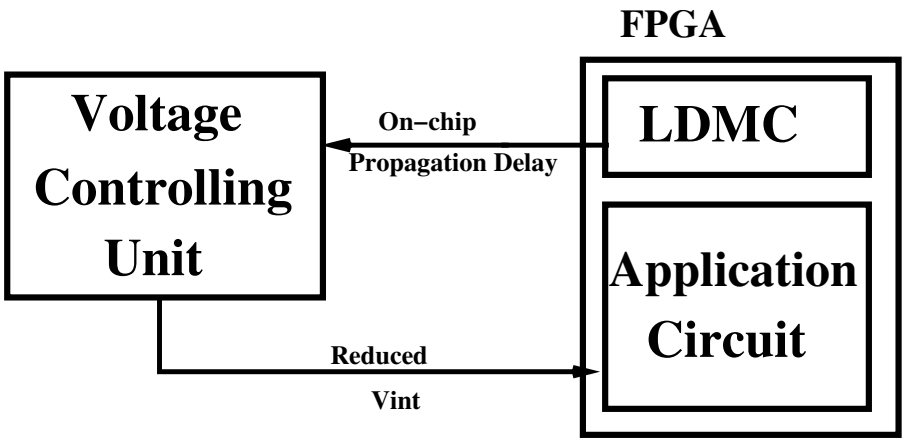

FIGURE 1.7 – Exemple d'implémentation de DVS [LL05]

1.5 Potentiels des architectures reconfigurables

Dans la suite de ce travail, nous allons nous intéresser essentiellement aux architectures reconfigurables et nous présenterons des exemples d'architectures classées en tenant en compte de l'aspect granularité.

1.5.1 Architecture : les tendances principales

1.5.1.1 Architecture reconfigurable à grain fin

Les premiers circuits reconfigurables (FPGA) ont été basés sur des éléments de traitement et des communications reconfigurables à grain fin. L'élément de traitement reconfigurable à grains fin est la LUT (Look Up table) qui permet de réaliser des fonctions combinatoires simples. A chaque élément configurable est associée une mémoire de configuration. Les technologies basées sur les mémoires SRAM permettent la reconfiguration par chargement de différents bitstreams.

Les FPGAs commerciaux contiennent plusieurs LUT avec 3 à 6 entrées. La figure 1.8 illustre un exemple de LUT du FPGAs virtex-5. Par rapport aux anciennes versions de LUTs de FPGA à 4 entrées, elle permet de réaliser des fonctions de plus grandes tailles comme des RAM distribuées de 256 bits, registres à décalage de 128 bits et fonction à 8 entrées dans un même CLB. La reconfiguration de LUT est très flexible, et peut être utilisée pour réaliser n'importe quel circuit numérique. Cependant, les éléments fonctionnels reconfigurables à grain fin occupent substantiellement plus de surface, ont une grande latence, et consomment plus par rapport à des structures à grain épais. Néanmoins, dans les FPGAs modernes, on rencontre de plus en plus d'éléments de calcul à grain épais, tels que les multiplieurs, les DSPs jusqu'à des structures complexes en dur telles que les processeurs RISCs.

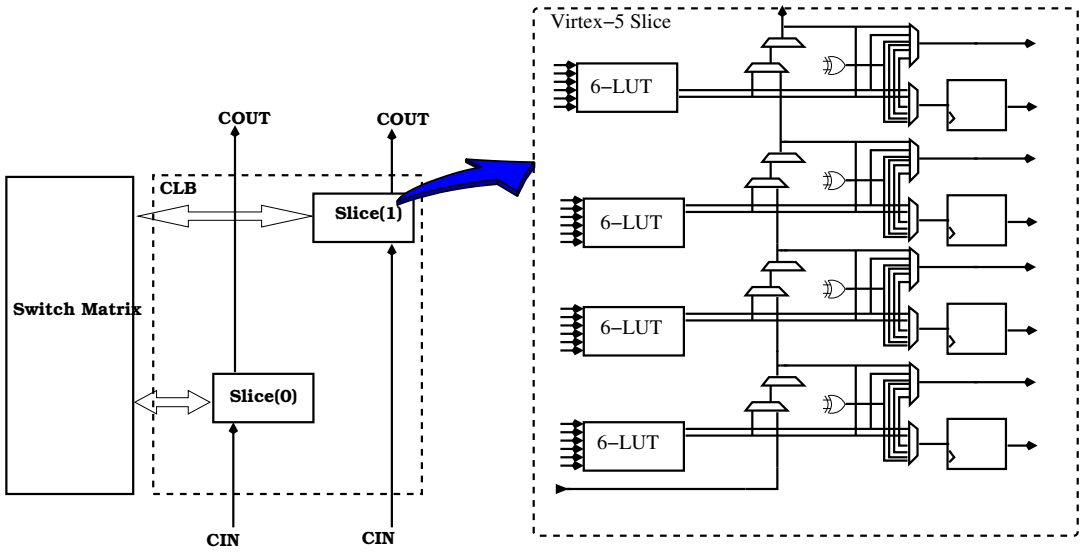

FIGURE 1.8 – Blocs logique configurable(CLBs) Virtex-5 contenant deux slices

Les circuits FPGAs reconfigurables L'architecture interne des circuits FPGAs de la famille Virtex est détaillée dans les références [Inc05b] [Inc06] et [Inc05c]. L'unité reconfigurable est une trame (frame). Une trame est un ensemble de cellules logiques (CLB). Dans Virtex-II, celle-ci est une colonne qui occupe toute la longueur du circuit. Cependant, dans le cas des Virtex-4, 5 et 6, la trame a une taille de 16 CLBs (1.1). L'avantage de l'organisation de ces derniers circuits par rapport à Virtex-II est la possibilité de

définir les deux dimensions de la zone reconfigurable, alors que pour les Virtex-II une dimension est fixée (la colonne). La table 1.1 résume les caractéristiques principales des FPGAs à la famille Virtex. En plus, il est possible de définir des zones reconfigurables avec leur propre horloge fonctionnant grâce à une organisation en région d'horloge. La figure 1.9 illustre l'architecture d'un circuit Virtex-4 et la distribution de l'arbre de l'horloge.

TABLE 1.1 – Exemple de caractéristiques des FPGAs Virtex-II, Virtex-4 et Virtex-5

Composant	XC2VP	XC4V	Virtex-5
architecture CLB	2 LUT4+ 2FF	2LUT4+2FF	4LUT6+4FF
Unité logique	LUT à 4 entrées	LUT à 4 entrées	LUT à 6 entrées
zone minimale de RP	trame (longueur du circuit)	trame(16 CLB)	trame16 CLB
RAM distribuée	128 bits par CLB	64 bit par CLB	256 bits par CLB

Exemples d'architectures *CASA :*est un système de traitement et d'acquisition. Celui-ci est utilisé dans les applications météo pour détecter les conditions dangereuses [RLAR05]. Un diagramme du système est donné dans la figure 1.10.

Un des deux FPGAs de CASA est dédié au traitement du signal (FPGA à gauche dans la figure), l'autre FPGA est responsable de la communication du résultat, mais peut aussi traiter les données en fonction de la configuration. Un microcontrôleur ARM exécutant Linux gère la ressource de configuration du FPGA. CASA contient aussi une série de mémoire (multi-banked memory), interfaces de réseau multiples, et des convertisseurs analogiques numériques (A/D). Dans cet exemple, le FPGA s'occupe du processus de communication. C'est un exemple appliqué dans un environnement qui demande beaucoup de puissance de calcul et différents algorithmes. L'un des avantages de cette architecture est les algorithmes de traitement exécutés sur le circuit FPGA qui peuvent être changés en réponse au changement des conditions météo. Cette configuration est effectuée au niveau grain fin dans le circuit FPGA. De plus, une fonction de contrôle à distance par le port ethernet augmente l'adaptabilité du système.

System on-demand : proposé par Ullmann et al. [UHGB04], est un système reconfigurable à grain fin sur la plate-forme de circuit FPGA Virtex-II Xilinx. Il est déve-

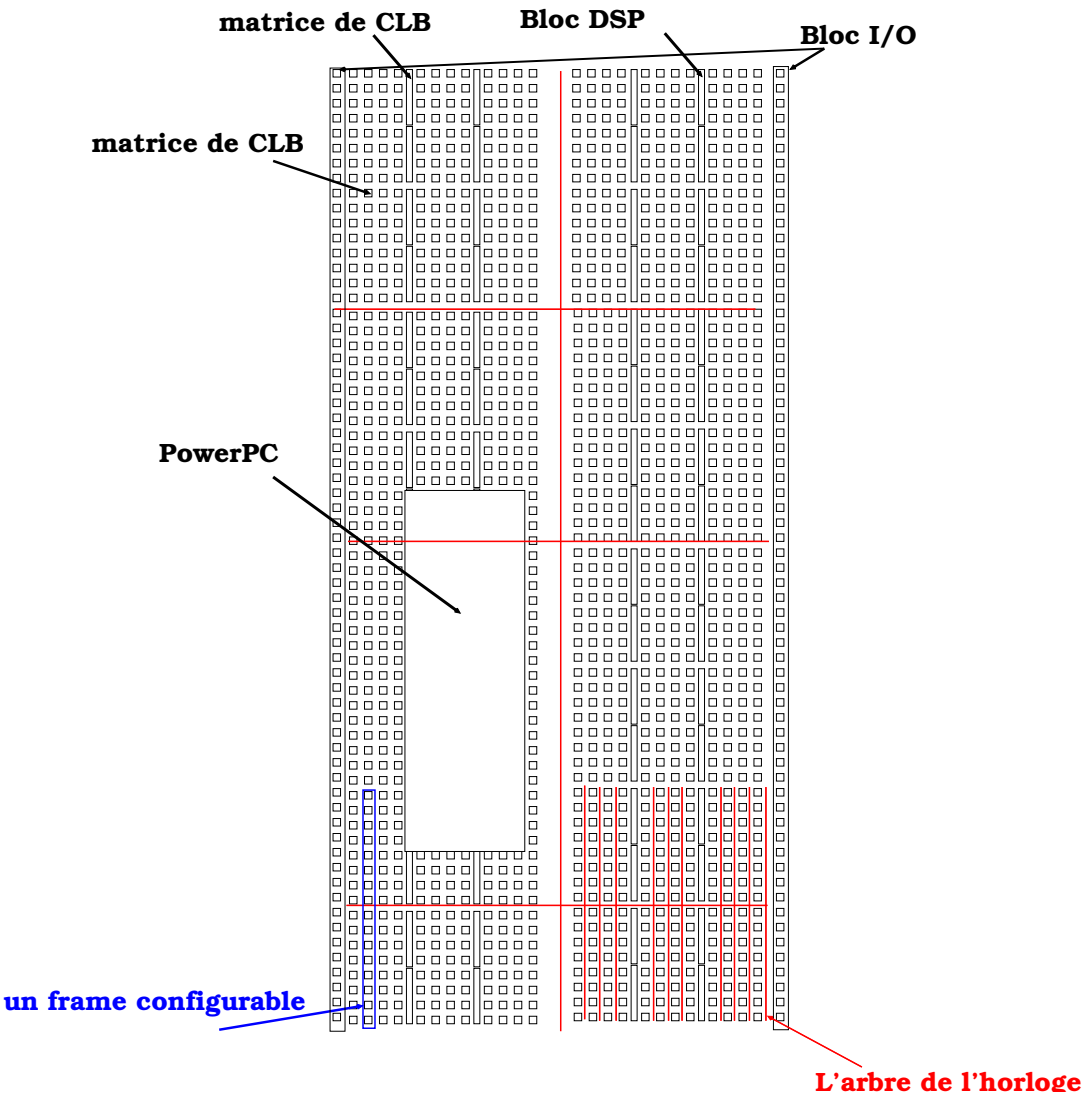

FIGURE 1.9 – Organisation interne de Virtex-4 de Xilinx

loppé pour les applications automobile industrielles [UHGB04]. Dans l'architecture de cette exemple 1.11, plus de trames reconfigurables sont connectées ensemble avec un bus du système. chaque trame peut être reconfigurée par des données de configuration (bitstreams) qui sont enregistré dans une mémoire flash . Un dialogue existe entre le contrôleur (microprocesseur) et le module reconfigurable pour identifier l'état du module reconfigurable en temps réel. Ce système est auto-adaptatif au niveau grain fin. Il supporte plus de flexibilité en utilisant la reconfiguration partielle. Et puis, il garde aussi le maximum de performance d'exécution grâce à l'implémentation du sys-

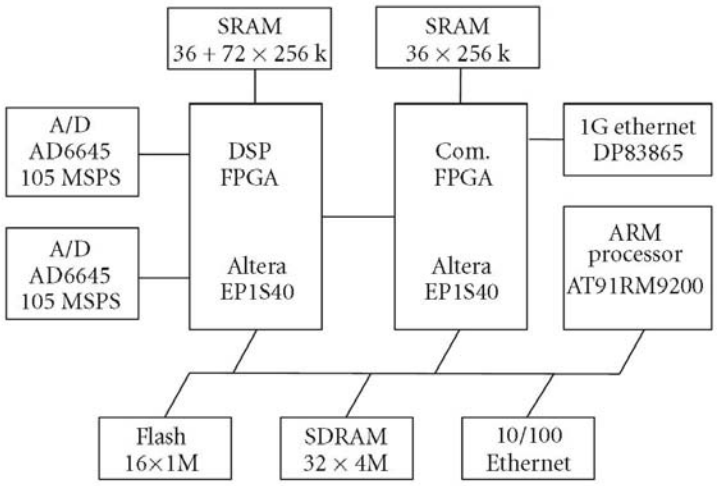

FIGURE 1.10 – Exemple de l'architecture CASA [RLAR05]

tème au niveau grain fin. Cependant, le choix de la taille de trame de reconfiguration est indépendant du nombre de fonctions logiques nécessaires à cause de la limitation de la méthode de reconfiguration dans le circuit FPGA. Celui-ci peut évoluer avec les nouveaux circuits FPGAs tels que Virtex-4,5. A cause de cette limitation, il n'y a pas une méthode pour distinguer comment choisir le nombre de zones reconfigurables ainsi que la consommation du circuit.

Bilan La reconfiguration d'une architecture à grain fin offre la plus grande flexibilité ; ce qui peut être un point fort pour un système adaptatif. Elle permet de configurer les cellules logiques du circuit FPGA en fonction des différents besoins de l'application, pour spécialiser chaque tâche au maximum ce qui en général améliore l'efficacité. On peut par exemple ajuster la taille des opérateurs, le nombre de niveaux de pipeline, le taux de parallélisme, etc. Un autre intérêt est que les composants matériels ainsi que les outils bas niveau de génération de configurations sont facilement disponibles à faible coût. Enfin, certains de ces circuits possèdent une interface interne de reconfiguration, ce qui permet une forte intégration du système adaptatif complet.

Ces points forts des architectures reconfigurables à grain fin offrent une grande flexibilité pour la réalisation de système adaptatifs. Elles ont également quelques in-

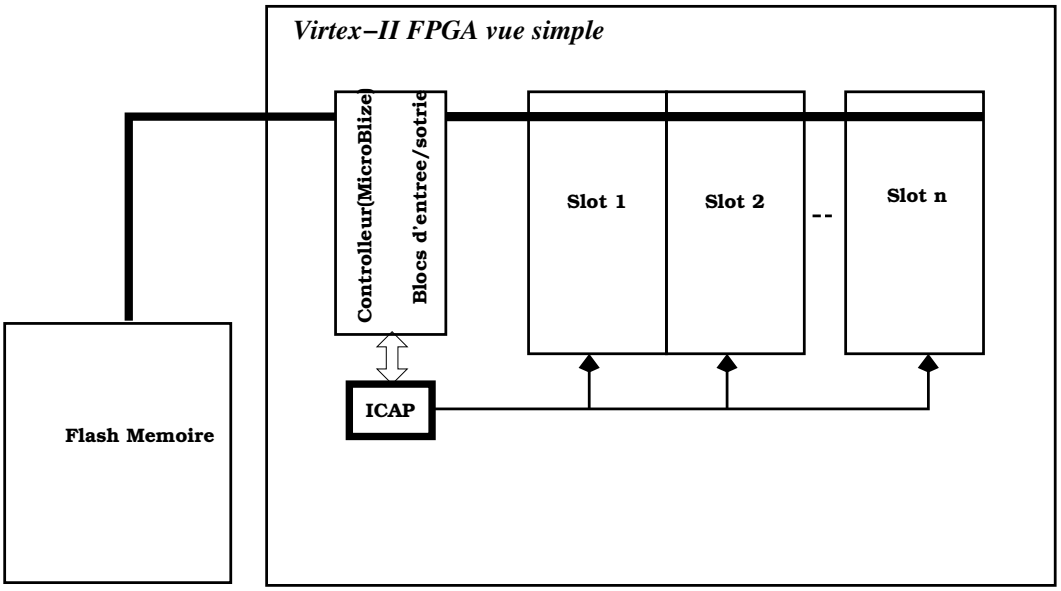

FIGURE 1.11 – Exemple d'une architecture reconfigurable sur Virtex-II de [Ua04]

convénients tels que des temps de reconfiguration élevés, une augmentation de la surface, ainsi qu'une augmentation de la consommation d'énergie. Cependant, les dispositifs à grain fin sont en général limités par des temps de reconfiguration, une surface de silicium et une consommation d'énergie plus élevés que les structures à grains épais. A l'heure actuelle, ces architectures restent très efficaces d'un point de vue de l'adaptabilité de fonctionnement, mais inefficace d'un point de vue énergétique.

1.5.1.2 Architecture reconfigurable à grain épais

Les circuits reconfigurables ont évolués depuis quelques années des grains fin à des structures à grain plus épais qui peuvent être des fonctions simples arithmétiques et logiques voir des processeurs élémentaires plus complexes. Alors que les architectures à grain fin sont plus adaptées aux traitements au niveau bit, les architectures à gros grains sont plus adaptées au traitement des données sur des mots de plus grandes tailles.

L'inadéquation des FPGAs avec certaines applications a amené les concepteurs à proposer d'autres modèles d'architectures. Les interconnexions et les opérateurs sont configurables dans ces architectures, mais travaillent au niveau arithmétique (par exemple

8 ou 16 bits). Ceci a pour effet d'améliorer les performances sur les traitements arithmétiques, au détriment des optimisations au niveau bit. Ces approches proviennent du fait que les applications sont constituées de cœurs de boucles très réguliers [ZBR00]. Ces architectures réduisent par ailleurs la taille des configurations, permettant d'envisager des stratégies de reconfiguration dynamique. La plupart de ces architectures utilisent un réseau d'interconnexion [ZWGR99] à deux dimensions, éventuellement hiérarchiques. On peut citer, par exemple, les architectures DRP-1 [DRP], Morphosys [LKS00], RaPiD [Cro99] ou le KressArray [HK95], Systolic Ring [G. 02]. Certaines architectures présentent une topologie linéaire comme le Piperench [GSM+99]. La taille des reconfigurations étant limitée, des mécanismes évolués peuvent ainsi être mis en œuvre pour distribuer les informations de configuration et réduire le coût (en consommation) du contrôle de ces architectures. Ces considérations ont été mises en œuvre dans les architectures Pleiades [Abn01]et DART [DCPS02], spécifiquement conçues pour la faible consommation.

Exemples d'architectures *ADRES :* est une architecture reconfigurable à grain épais composée de deux parties : un processeur de type de VLIW et une matrice reconfigurable (fig.1.12) [MVV+03]. Dans la partie VLIW, certaines unités fonctionnelles (FU) sont connectées à un fichier de registres multi-ports (RF). La matrice reconfigurable contient des circuits reconfigurables (RC)(fig. 1.12). Chaque RC est un élément de calcul reconfigurable gros grains et correspond à une ALU 32 bits. La configuration de ces ALUs permet de réaliser des fonctions telles que l'addition, la multiplication et les fonctions logiques.

ADRES réduit d'abord la communication globale et la complexité de programmation grâce au partage des fichiers de registres et l'accès mémoire entre le processeur VLIW et la matrice reconfigurable.

DAP/DNA est un processeur reconfigurable d'IPflex[5] qui possède une version de plate-forme reconfigurable dynamiquement. Le DAP/DNA consiste en une matrice d'éléments de calcul (PE) et un processeur RISC (DAP). La matrice permet une re-

5. www.IPflex.com

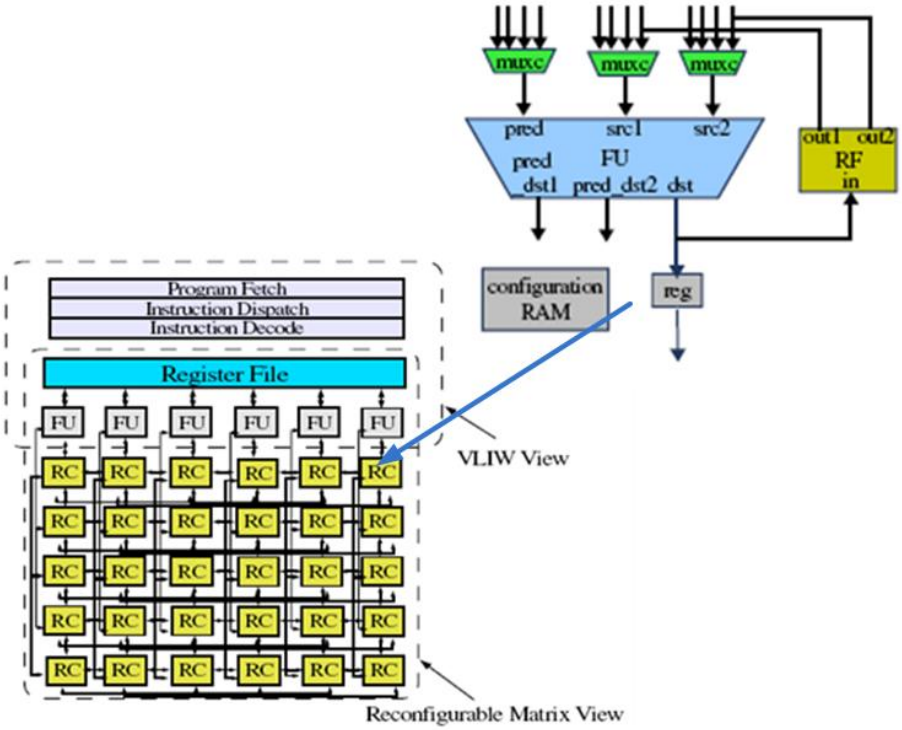

FIGURE 1.12 – Architecture ADRES

configuration en un cycle des 376 éléments de calcul et les interconnexions (fig. 1.13). Chaque élément de calcul peut être reconfiguré soit en mode séquentiel, soit en mode parallèle.

L'environnement de développement DAP/DNA-II permet d'assister le concepteur dans les différentes étapes du processus de développement jusqu'à l'implantation sur le composant. La flexibilité est offerte par la configuration d'interconnexion et la configuration de fonction de PE. La taille de DNA et la matrice de PE sont fixés.

Bilan Les systèmes reconfigurables à grain épais se présentent comme un bon compromis entre FPGAs et circuits spécifiques (ASICs). Ce type d'architecture est basé sur une adaptation au niveau du jeu d'instructions. Différentes fonctions sont réalisées par la reconfiguration de l'organisation du chemin de données dans chaque PE et l'interconnexion des PEs.

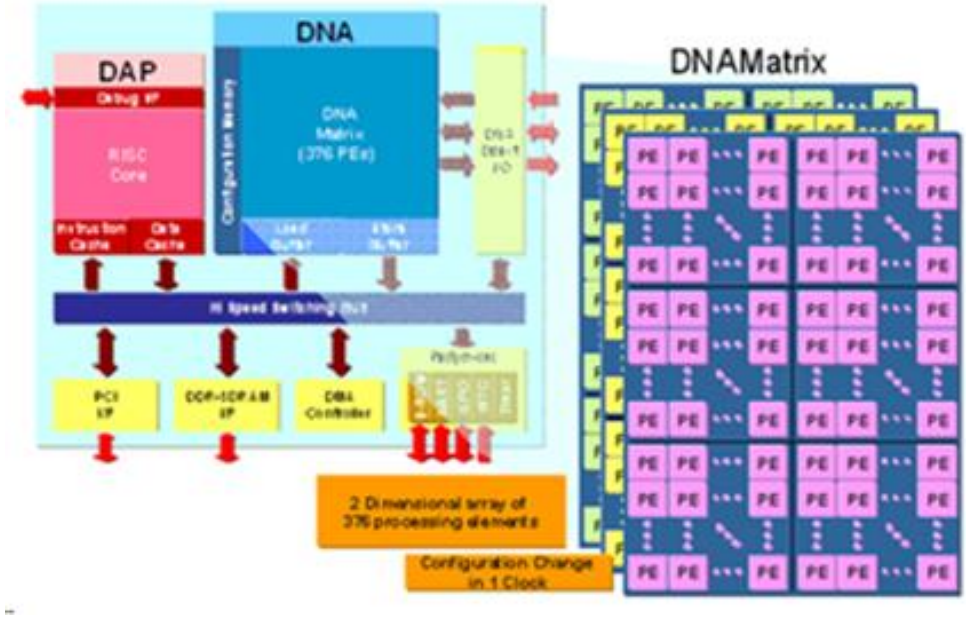

FIGURE 1.13 – Architecture DAP/DNA-2

1.5.1.3 Architecture reconfigurable hétérogène

Afin de trouver un compromis entre ces différentes caractéristiques, des architectures hétérogènes ont été conçues notamment par des fabricants de FPGAs tels que Xilinx et Altera qui proposent différentes familles de circuits intégrant plus ou moins de blocs spécialisés (DSP, mémoire, transceiver, proceseurs) [Wol00]. Dans les circuits FPGAs actuellement commercialisés, on rencontre de plus en plus un mélange de granularité. En effet, certains éléments de calcul à grains épais sont embarqués dans ces FPGAs. Ce sont par exemple des multiplieurs, des modules "DSP", ou encore des processeurs sous forme d'IP (Nios, Microblize) ou en dur (PowerPCs pour Xilinx [Inc05b] [Inc04a]).

La structure d'architecture reconfigurable hétérogène est composée de différents types d'éléments de calcul à grain fin et/ou à grain épais. L'adaptation est réalisée par le choix de type de élément de calcul et les interconnexions entre ces éléments. Le type d'élément peut être changé en utilisant la reconfiguration partiel [Inc04b] [Inc05c] tel que [MCM+04]. L'interconnexion entre les éléments peuvent être également reconfigurée comme dans l'exemple $aSoC$ [LLTB]

Exemples d'architectures *aSoC :*est une architecture basée sur un réseau d'interconnexions en point à point [LLTB]. Comme illustré sur la figure 1.14, les briques constitutives sont des éléments de traitement hétérogènes. L'architecture est constituée de briques qui peuvent être de granularités différentes. L'aSoC est très flexible puisque le concepteur peut définir les modules de calcul (IP) qu'il souhaite intégrer à son architecture.

Certaines briques peuvent être des briques reconfigurables de gros grain ou de grain fin. Les interfaces de communication de l'aSoC relient point à point les briques entre elles. Les instructions de connexions contrôlent un crossbar qui peut établir des liaisons entre les quatre briques du voisinage direct (au nord, au sud, à l'est et à l'ouest) de la brique dont il fait partie.

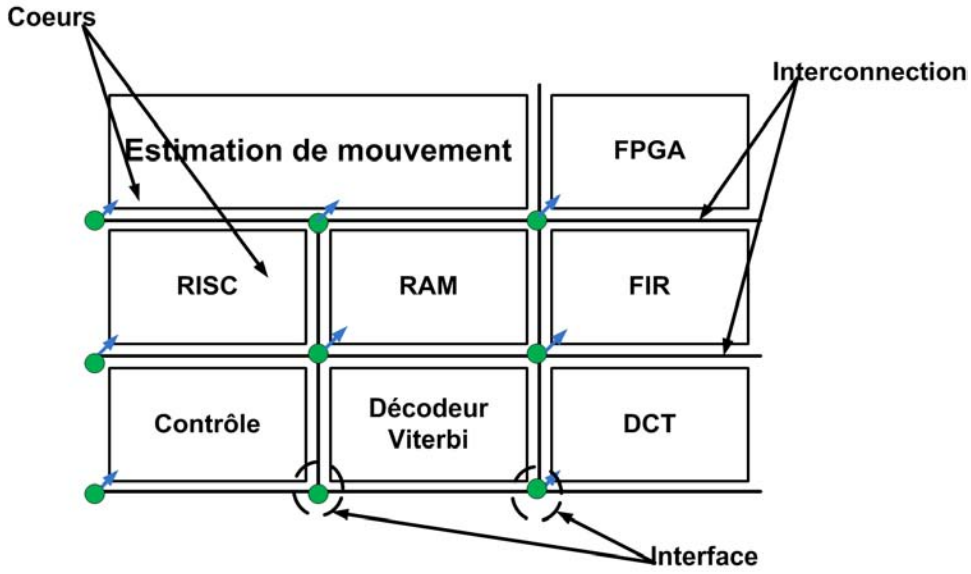

FIGURE 1.14 – Architecture aSoC

Le crossbar peut aussi créer une liaison entre une de ces briques de voisinage et le module de calcul avec lequel il peut communiquer. La figure 1.14 montre l'interface de communication d'une brique avec le crossbar et la mémoire d'instructions. Sur cette figure nous trouvons aussi un dispositif permettant la gestion dynamique de la consommation de puissance de la brique [LLTB]. L'un des avantages qu'offre cette architecture est qu'elle supporte des transferts de données ordonnancés que ce soit de manière statique hors ligne ou bien dynamique en ligne.

FiPRe : est un système reconfigurable basé sur du multi-processeurs [MCM+04]. Le coprocesseur, RISP (Reconfigurable instruction set processeur [BL00] [BLD02]), embarqué est responsable de l'accélération de certaines fonctions critiques. Ce modèle multi-processeur a été implémenté en utilisant la technologie Virtex-II Xilinx. De plus, il peut être inséré ou remplacé en temps réel. Le processeur RISPs est implémenté dans une zone reconfigurable. Ces processeurs peuvent être remplacés par des co-processeurs différents. Cependant, la taille de la zone reconfigurable est fixée dans la conception (4 colonnes CLBs/1280LUTs). En comparant avec $aSoC$, les modules de calcul $RISP$ ne sont pas pré-implémentés sur le circuit. Chaque module de calcul est implémenté dans une zone reconfigurable. Il peut être remplacé par un autre co-processeurs tels que FFT opérateurs, filtrage d'image, etc.

DyCORE : est un système adaptatif qui est capable de reconfigurer une partie du système et peut s'adapter à différents comportements [AKM06] . Il est très similaire à $aSoC$ dans lequel les éléments à grains épais et/ou à grain fin sont connectés ensemble sur un bus. Cependant, la grande différence avec ce dernier exemple tient dans le fait qu'il est modifiable par reconfiguration dynamique. Il est capable de réaliser une architecture multiprocesseur et aussi un architecture multi PEs matériel selon les besoins d'application. Un contrôleur d'adaptation est installé à côté du système.

Bilan Ce type d'architectures contient différents type d'éléments de calcul permettant d'adapter le fonctionnement d'une application spécifique. De nombreuses architectures (ex : $ASoC$) reconfigurables hétérogènes peuvent reconfigurer leur réseau d'interconnexion entres les éléments de calcul . Un des avantages de ces architectures, est la possibilité d'adapter le parallélisme de l'application à celui de l'algorithme exécuté. Cette spécificité autorise l'ajustement la fréquence de fonctionnement et/ou la tension d'alimentation en fonction du niveau de parallélisme de l'algorithme.

Très peu d'architectures (ex :$DyCORE$) reconfigurables hétérogènes peuvent reconfigurer leurs éléments de calcul. Un des avantages de ces architectures, est la possibilités de changer le type d'éléments de calcul pour adapter le fonctionnement spécifique de l'application.

Les trois types d'architectures décrivent les possibilités d'adaptation au niveau matériel. La plupart des études menées sur ce thème se concentrent uniquement sur le gain en performances et l'adaptabilité aux multi-fonctions au prix d'une consommation d'énergie non maîtrisée.

1.6 Conclusion

Dans ce chapitre, nous avons présenté les caractéristiques des architectures pouvant être utilisées comme SoCs adaptatifs au niveau matériel. Les éléments principaux sont décrits selon la granularité des éléments de calcul et d'interconnexion. Pour chaque type de granularité, différentes architectures sont analysées. Malgré la quantité d'architectures existantes, le marché est encore dominé par les FPGAs, en particulier ceux de Xilinx et Altera.

Grâce à la reconfiguration partielle dynamique dans le circuit FPGA, il est possible de réduire le problème des temps de reconfigurations prohibitifs. Le surplus de surface peut être partiellement compensé par une augmentation de la fonctionnalité apparente permise par la flexibilité du système. Le problème principal à résoudre reste la consommation énergétique, induite principalement par le système de routage intercellules [SN05] [DPF+00]. Cette contrainte, bien que unanimement reconnue comme l'une des plus critiques pour les systèmes embarqués [Bol07], n'est quasiment jamais prise en compte lors de la conception des architectures reconfigurables.

Notre objectif n'est pas de concevoir un système avec une meilleure performance ou une meilleure consommation d'énergie, mais de proposer un moyen permettant de trouver un compromis entre la consommation d'énergie et les performances lors de l'exécution de l'application. Et surtout, d'adapter ce compromis aux variations des conditions de fonctionnement de l'application.

Pour atteindre cet objectif, nous exploitons le parallélisme offert par les circuits FPGAs et sa mise en oeuvre dynamique par reconfiguration partielle dynamique en permettant d'implémenter différentes versions et le nombre adéquat d'instances d'une tâches. La gestion dynamique du parallélisme est associée à la gestion dynamique

de fréquence afin de permettre au système de s'adapter au nouvel environnement et de garantir toujours l'efficacité énergétique et les performances nécessaires. Cette méthode sera développée dans le chapitre suivant.

Chapitre 2

Méthode et stratégie d'adaptation

Sommaire

2.1 Introduction

Ce chapitre présente la méthodologie d'adaptation proposée. Dans ce travail, nous nous concentrons sur les deux méthodes : DRP et DFS. La reconfiguration partielle (DRP) nous permet de modifier le niveau du parallélisme des tâches. La reconfiguration de fréquence (DFS) de traitement permet d'obtenir la fréquence de traitement juste suffisante pour respecter la contrainte de temps et de réduire le temps d'attente. Le prototypage d'un système adaptatif selon ces méthodes ainsi que les éléments principaux de son architecture sont présentés. Ensuite, nous montrons qu'il est possible d'améliorer le rendement des implantations à l'aide de notre méthode (le niveau de parallélisme et la fréquence de traitement) tout en respectant la contrainte de temps.

2.2 Problématique et applications cibles

Dans la majorité des applications multimédia, un flot de données subit des traitements successifs qui peuvent être organisés sous forme de tâches avec une dépendance de données. Le temps d'exécution de ces applications dépend des temps d'exécutions des tâches et de la cible d'implémentation. Dans le cas d'une exécution sur un processeur unique, ce temps est la somme des temps d'exécution de chaque tâche. Plusieurs techniques permettent de réduire ce temps d'exécution comme la parallélisation par pipeline. Généralement, les traitements dans ces applications multimédias sont effectués de manière périodique et la période de traitement T_D corresponds à une contrainte à respecter.

Les temps d'exécution des tâches ne sont pas les mêmes et sont inférieurs à la période de l'application. La plupart des architectures sont surdimensionnées pour que le temps d'exécution des tâches soit toujours inférieur à la période. De ce fait, la durée d'attente des processeurs (matériels ou logiciels) peut être non négligeable et a un effet sur la consommation de puissance et bien entendu sur l'efficacité énergétique et l'efficacité d'utilisation de l'architecture.

Le temps d'attente est souvent exploité pour réduire la consommation en échelon-

nant dynamiquement la fréquence de fonctionnement et la tension d'alimentation. On peut ainsi atteindre un état de fonctionnement optimal pour des conditions de fonctionnement fixées. Lorsque ces conditions changent, par exemple dans le cas où la quantité de données à traiter augmente il est alors nécessaire d'agir sur l'organisation du système pour assurer le fonctionnement et d'atteindre un nouvel état optimal. Cette action peut se faire en allouant des ressources supplémentaires et en agissant sur les fréquences et tensions d'alimentation dans les limites possibles. Il est en conséquence nécessaire de prévoir un mécanisme permettant d'assurer conjointement ces actions. Évidemment, on ne peut utiliser que les ressources disponibles en les réorganisant de manière efficace.

Notre contribution dans ce domaine est d'associer la gestion dynamique de fréquence et de tension avec la reconfiguration dynamique partielle pour adapter la consommation d'un système en fonction des variations de la quantité de données traitées.

2.3 Puissance et énergie dans les applications périodiques

Dans cette étude nous nous intéressons essentiellement à la puissance dynamique dissipée dans les circuits CMOS de type FPGA. La puissance statique est considérée constante et dépendant du type de circuit choisi. Elle est dissipée quand le circuit lorsqu'il est alimenté, et elle est indépendante de l'activité du circuit.

2.3.1 Puissance dynamique

La puissance dynamique est dissipée lorsque le circuit est actif et les valeurs des signaux changent [Cur07]. Les signaux peuvent être des signaux de logique, d'horloge ou des signaux entrées/sorties. La dissipation est due aux commutations (charge et décharge des capacités du circuit) et aux courents de court-circuit pendant le phase de commutation. Les études menées pour estimer la dissipation de puissance dans les FPGAs [A.D00, LAK02] montrent toutes que 60 à 70% de la puissance est due aux interconnexions, 20% à la distribution d'horloge et le reste se répartit entre les blocs

de calcul et les entrées/sorties. L'expression simplifiée de la puissance dynamique est donnée par l'équation 2.1 [TGM07] :

$$P_{dyn} = C_L * V_{dd}^2 * P_{trans} * f_{clk} \tag{2.1}$$

Où, P_{trans} est la probabilité de transition, C_L est la capacité de charge et f_{clk} est la fréquence de l'horloge de communication. Si nous définissons :

$$C_{eff} = P_{trans} * C_L \tag{2.2}$$

Nous pouvons aussi décrire la puissance dynamique avec une expression plus familière :

$$P_{dyn} = C_{eff} * V_{dd}^2 * f_{clk} \tag{2.3}$$

2.3.2 Énergie consommée

L'énergie consommée correspond à la puissance dissipée dans un intervalle de temps $[t_0, t_1]$. Elle est calculée en fonction de la puissance par l'équation 2.4.

$$E = \int_{t_0}^{t_1} P(t) * dt \tag{2.4}$$

L'énergie consommée est proportionnelle à la puissance dissipée $P(t)$ et au temps. Le problème de la réduction de l'énergie consommée par le système inclut l'aspect temporel. Ainsi, deux tâches qui consomment la même puissance ne dissipent pas forcément la même énergie, cela dépend du temps d'exécution des tâches. Dans le domaine d'applications considéré, le temps d'exécution T_e est généralement lié au nombre de cycle d'horloge N_c et à la fréquence d'horloge f_{clk} par la relation 2.5 :

$$T_{exe} = \frac{N_c}{f_{clk}} \tag{2.5}$$

L'énergie consommée en phase d'exécution est, selon les relations 2.3 et 2.5, donnée par l'expression 2.6 :

$$E = C_{eff} * V_{dd}^2 * N_c \tag{2.6}$$

Cela signifie par exemple que l'énergie est la même pour deux puissance P_1 et P_2 avec des temps d'exécutions respectifs T_{exe1} et T_{exe2} ($P_1 * T_{exe1} = P_2 * T_{exe2}$).

Par contre, est ce que cette situation est toujours valable pour une tâche périodique ? (Où la période est composée d'une phase d'exécution et d'une phase de repos). Si nous supposons qu'une puissance P_{e1} est dissipée à la fréquence f_1. Le temps d'exécution est T_{exe1} dans une période fixée T_D et T_{repot1} le temps de repos ($T_D = T_{exe1} + T_{repot1}$). Le système va continuer à consommer pendant le temps de repos T_{repot1} à cause du routage d'horloge dans le circuit cible.

La puissance dissipée au repos est plus faible que la puissance en fonctionnement et peut être exprimée sous la forme $P_{repot1} = \alpha * P_{exe1}$, avec $\alpha < 1$. La figure 2.1 illustre deux cas de fonctionnement avec deux puissances P_{exe1} et P_{exe2} correspondant à deux fréquences différentes f_1 et f_2.

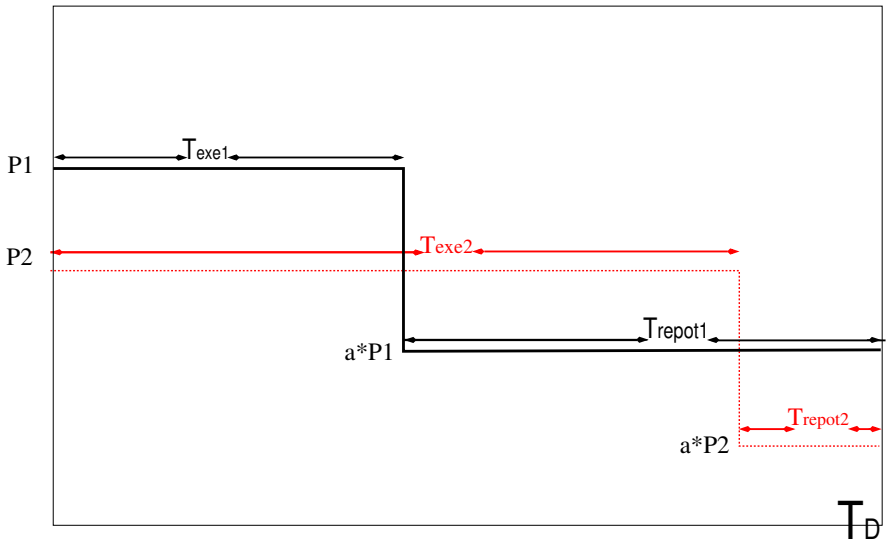

FIGURE 2.1 – Les puissances pendant le temps d'exécution et le temps de repos dans une période fixée

L'énergie consommées avec la puissance P_{exe1} est calculée de la manière suivante :

$$E_1 = P_{exe1} * T_{exe1} + \alpha * P_{exe1} * T_{repot1} = (1 - \alpha) * P_{exe1} * T_{exe1} + \alpha * P_{exe1} * T_D \quad (2.7)$$

Si nous réduisons la fréquence de traitement, la puissance P_{exe1} sera réduite à P_{exe2} et

le temps d'exécution augmentera à T_{exe1}. L'énergie consommée avec la puissance P_{exe2} est :

$$E_2 = P_{exe2} * T_{exe2} + \alpha * P_{exe2} * T_{repot2} = (1 - \alpha) * P_{exe2} * T_{exe2} + \alpha * P_{exe2} * T_D \quad (2.8)$$

L'énergie consommée pendant le temps d'exécution étant la même dans les deux cas, on peut remplacer $(P_{exe2} * T_{exe2})$ par $(P_{exe1} * T_{exe1})$ dans l'équation 2.8. on obtient :

$$E_2 = (1 - \alpha) * P_{exe1} * T_{exe1} + \alpha * P_{exe2} * T_D \quad (2.9)$$

En comparant les équations 2.8 et 2.9, l'énergie E_2 serait inférieure à l'énergie E_1 si la puissance P_{exe2} est inférieure à P_{exe1}. Ainsi, la consommation d'énergie pour une application périodique peut être réduite en choisissant de manière adéquate la fréquence de fonctionnement et la structure du cœur de traitement.

2.4 Efficacité énergétique

L'efficacité énergétique sur une période de traitement est le rapport entre le nombre d'opérations d'une tâche et l'énergie consommée, elle est définie par l'équation 3.15. Dans cette équation, N_{opt} représente le nombre d'opérations par période($T_D = T_{exe} + T_{repot}$). Le produit $(T_{exe} * P_{exe} + T_{repot} * P_{repot})$ représente l'énergie consommée pour une période.

$$e_E = \frac{N_{opt}}{(T_{exe} * P_{exe} + T_{repot} * P_{repot})} = \frac{N_{opt}}{(T_{exe} * (P_{exe} - P_{repot}) + T_D * P_{repot})} \quad (MOPS/W) \quad (2.10)$$

Comme le montre l'équation 2.10, l'efficacité énergétique dépend en particulier de la période de traitement T_D. Pour une période donnée, on peut trouver une architecture efficace en terme d'énergie et contraintes de traitement. Cependant, si cette période change, ce qui est le cas par exemple dans les applications où la quantité de données est variable, l'optimum n'est plus assuré. Plus précisément, si la période diminue, en toute logique l'efficacité devrait augmenter. Or, l'optimum est obtenu lorsque le temps

d'exécution est le plus proche et inférieur à la période, ainsi la diminution de la période risque de provoquer une violation des contraintes de temps de traitement. Dans le cas contraire où la période augmente, l'efficacité énergétique diminuera. Dans les deux cas, il est donc nécessaire de modifier les paramètres de l'architecture pour revenir au point optimum.

Dans plusieurs travaux, les méthodes proposées essayent de réduire l'influence d'une ou de plusieurs sources [Abe04]. Ces méthodes interviennent à différents niveaux d'abstractions (niveau algorithme, architecture, système et logique : 2.2) [BB95, ZWGR99, SN05, Ess07, AN04, PTDK08, PHB09].

Á titre d'exemple, les auteurs [PTDK08] proposent une méthode d'optimisation de la transmission de paquets vidéos dans le réseau WIFI. La méthode se situe au niveau algorithme en effectuant des optimisations du comportement de l'application pour l'adapter à une plate-forme cible. Pour les méthodes au niveau système deux approches existent : l'approche codesign hardware/software traite toutes les parties du système logicielles et matérielles [BFY+02]. La deuxième approche correspond à la gestion dynamique de la consommation dans laquelle le comportement du système est surveillé en temps réel. Au niveau circuit FPGA, des travaux proposés tels que [Ess07] s'intéressent au développement de nouvelles techniques de fabrication pour réduire la consommation statique. Le FPGA Virtex-4, par exemple, de Xilinx économise 40% de consommation de puissance statique par rapport aux versions précédentes [Bol07, Inc05b, Inc05a].

Pour les méthodes au niveau architecture, la gestion de consommation utilise les techniques de reconfiguration dynamique pour modifier la granularité du parallélisme au niveau de pipeline [Bol07]. Dans la gestion d'alimentation et de fréquence, on fait évoluer la tension ou la fréquence du circuit pour optimiser la consommation. Ces deux méthodes sont souvent précédées par une étape d'estimation de la consommation des tâches du système sur les implémentations possibles. Un des inconvénients de cette méthode est l'ordonnancement statique de tâches de l'application. Pour s'adapter à un nouveau comportement du système, il faut refaire entièrement le flot de conception.

Pour avoir plus d'efficacité sur la réduction de la puissance dissipée et plus de flexi-

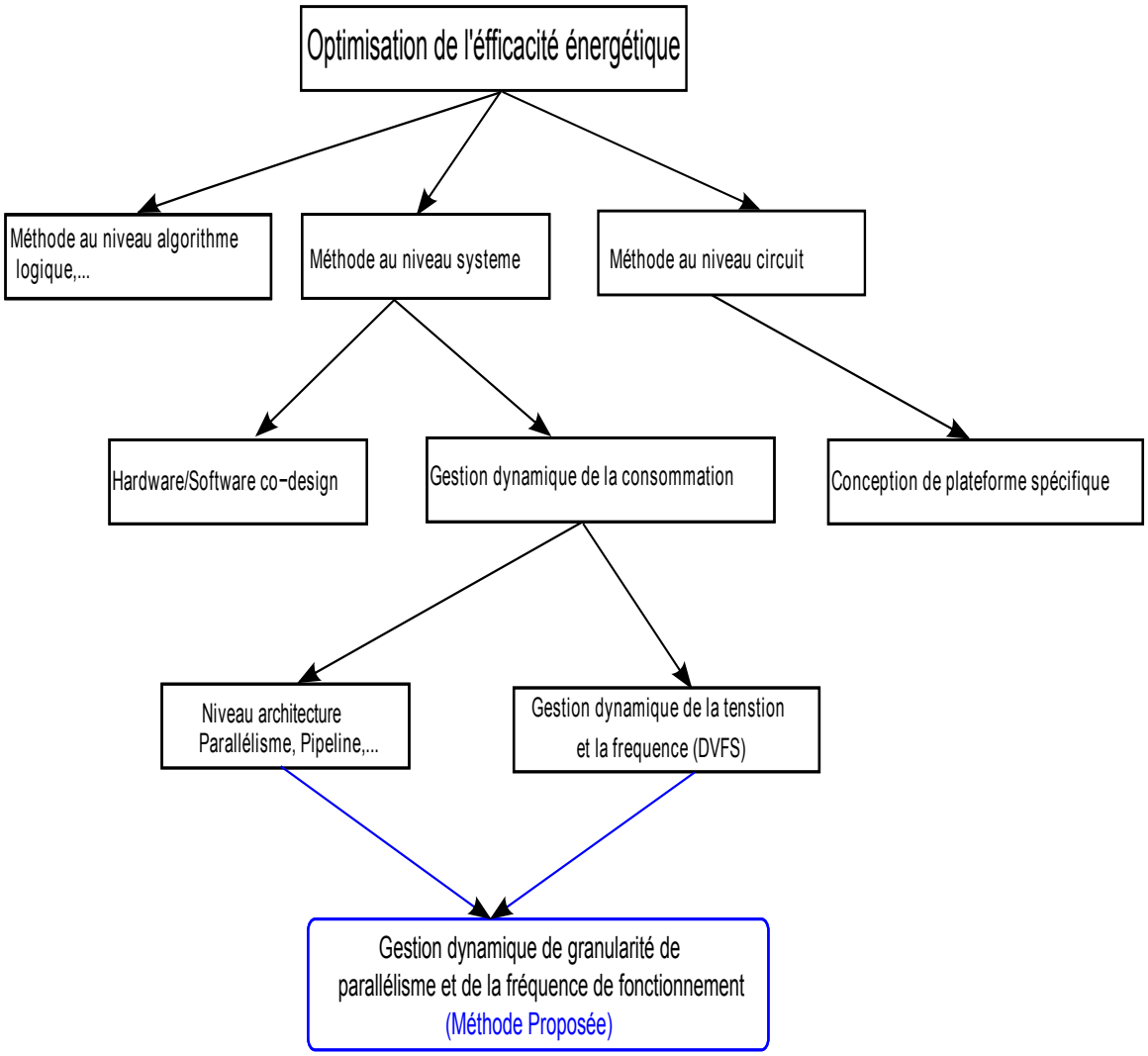

FIGURE 2.2 – Classifications des méthodes de l'optimisation de la consommation et solution proposée

bilité potentielle, certains travaux utilisent conjointement plusieurs méthodes [CA05]. La stratégie de la minimisation de la puissance dynamique est située entre le niveau algorithme et le niveau système. Les auteurs [PHB09] proposent une approche de gestion dynamique de puissance utilisant la reconfiguration partielle pour modifier la fréquence du système pendant le temps d'exécution et pendant le temps de repos. Cependant, la puissance pendant la configuration et le temps de configuration limitent l'utilisation de cette méthode.

2.5 Modèle architectural

Le modèle architectural d'un système adaptatif doit satisfaire en particulier les points suivants :

Adaptabilité :

- **Prise en compte de l'application** qui peut spécialiser les fonctionnalités avec ses propres implantations, l'application doit pouvoir se spécialiser en conservant des possibilités d'adaptation.
- **Prise en compte de l'implantation** qui peut se référer à la structure du système existant, mais elle doit pouvoir adapter ses stratégies.
- **Prise en compte du changement de l'environnement** pour réagir et s'adapter aux variations, le système d'adaptation doit : (a) s'intégrer avec la fonctionnalité de détection et de notification de changements de l'environnement représentés par les réactions externes et internes et (b) permettre de spécifier, au sein d'une ou plusieurs stratégies d'adaptations, les choix d'adaptations qui doivent alors être appliqués.

Dynamicité : L'adaptation du système ne doit pas entraîner son arrêt. À chaque fois qu'il est nécessaire d'installer ou de supprimer des modules pour répondre aux exigences d'adaptation, il n'est pas envisageable d'arrêter l'application et le système, de modifier ceux-ci puis de les relancer pour prendre en compte ces nouveaux modules. L'implémentation des différentes actions d'adaptation doit donc être dynamique. Afin de se conformer à ces caractéristiques, l'architecture est organisée sous forme de modules isolés illustrés par la figure 2.8.

Efficacité énergétique : l'efficacité étant définie par le nombre d'opérations effectuées par seconde et par unité de puissance. L'environnement est souvent variable, ce qui induit de nombreux changements, se traduisant par l'application de plusieurs stratégies d'adaptation. Nous faisons une hypothèse : lorsque l'environnement passe d'un état (E) à un autre état (E'), il y a n solutions adaptées. Cependant pour l'adaptation à une fonctionnalité donnée, l'efficacité de chaque solution peut être différente en raison des variations de paramètres utilisés. L'objet de ce chapitre est de proposer une

méthode de recherche de la solution adaptée la plus efficace.

2.5.1 Unité de calcul Reconfigurable-RPM

Plusieurs RPM (Reconfigurable Partial Module) sont connectés ensemble pour constituer une partie fonctionnelle d'un système adaptatif que nous appellerons $cluster$. Ici, un cluster réalise une tâche parallélisable d'une application. Un exemple d'un cluster est présenté sur la figure 2.8. Dans cette figure, chaque RPM est une unité de calcul élémentaire. Dans une partie de l'étude, nous nous sommes intéressés à la détermination du nombre de RPMs à exécuter en parallèle en fonction des variations des conditions de fonctionnement.

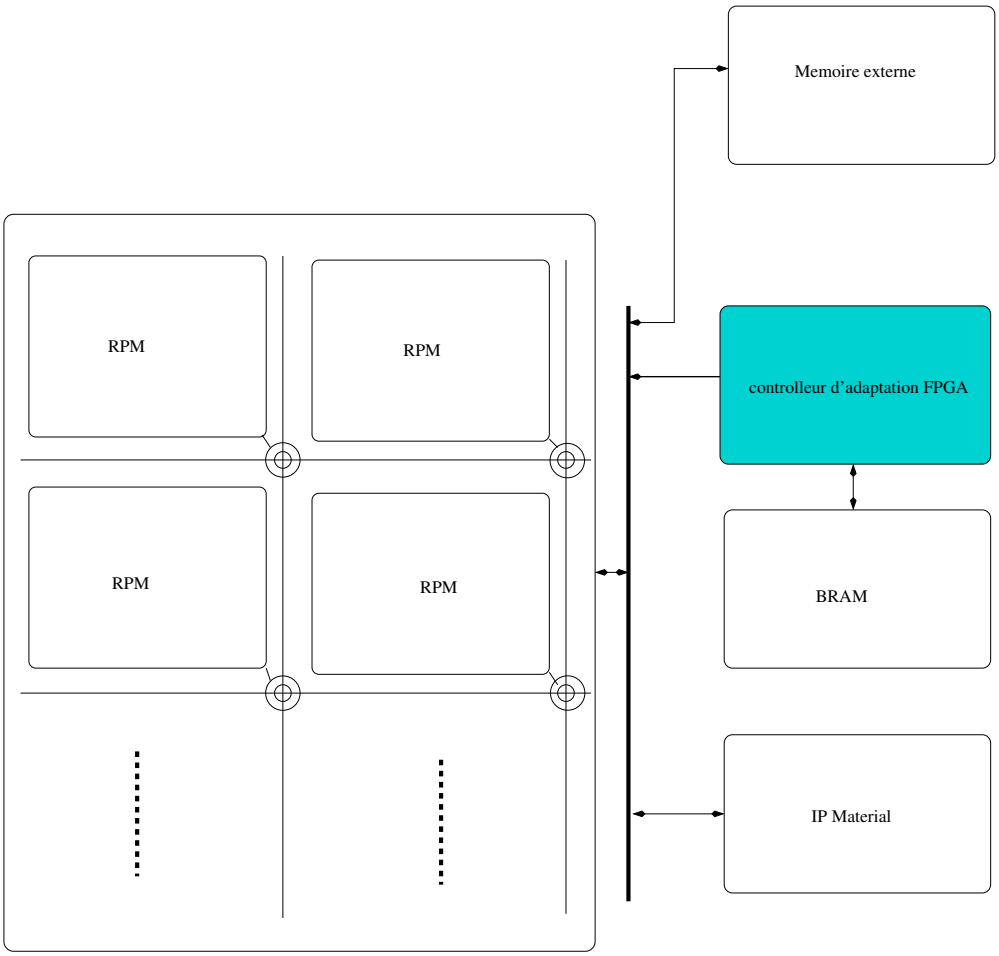

FIGURE 2.3 – Organisation de l'architecture du système adaptatif

2.5.1.1 Placement et nombre de RPM

Un ensemble de RPM est utilisé pour représenter une tâche d'une application. Chaque RPM est un module reconfigurable et peut exécuter différentes tâches d'une application par reconfiguration. Pour une tâche donnée, les RPMs sont identiques, puisqu'on se situe dans le cas de parallélisme de données. Nous nous intéressons ici à l'étude et la détermination de l'influence du nombre de RPM en fonction d'un certain nombre de paramètres tels que : la surface totale de la plate-forme, la contrainte de temps d'exécution, le temps de reconfiguration ainsi que la bande passante des accès mémoire.

i. La surface totale du circuit cible : Le nombre d'instances d'une tâche parallélisable est lié à la surface du circuit cible et à la surface du module reconfigurable(RPM) sur lequel s'exécute cette tâche. L'équation2.11 donne l'expression du nombre maximum N_1 de RPM en fonction de la taille S_{RPM} d'un RPM et de la taille totale S_{totale} de la zone reconfigurable :

$$N_1 = \frac{S_{totale}}{S_{RPM}} \tag{2.11}$$

La taille S_{totale} est toujours limitée et constante. Si nous ne considérons pas la consommation d'énergie des cellules logiques, le maximum de surface du circuit doit être utilisé pour maximiser l'efficacité d'utilisation du circuit et le débit de traitement. Un exemple que nous avons publié dans [ZRW08] est illustré sur la figure 2.4. Nous utilisons l'algorithme IDWT de décodage d'image comme un module fonctionnel. Le temps d'exécution en ordonnée dans cette figure illustre la diminution du temps d'exécution en fonction du nombre de modules fonctionnels.

Cependant, l'occupation de cellules logiques (LUT dans le FPGA) représente l'utilisation des ressources de la puce. La consommation d'énergie augmente dès qu'il y a plus de cellules logiques utilisées. Ceci est dû à l'accroissement de la consommation d'énergie dynamique (voir la section 2.2). Dans la constitution des FPGAs utilisés ici, les ressources non utilisées après la configuration du FPGA

ont une consommation dynamique nulle. Une autre illustration est donnée par l'exemple de la figure 2.5. La consommation augmente proportionnellement avec le nombre de cellules logiques utilisées. Ceci est vrai bien entendu pour la même fréquence de fonctionnement.

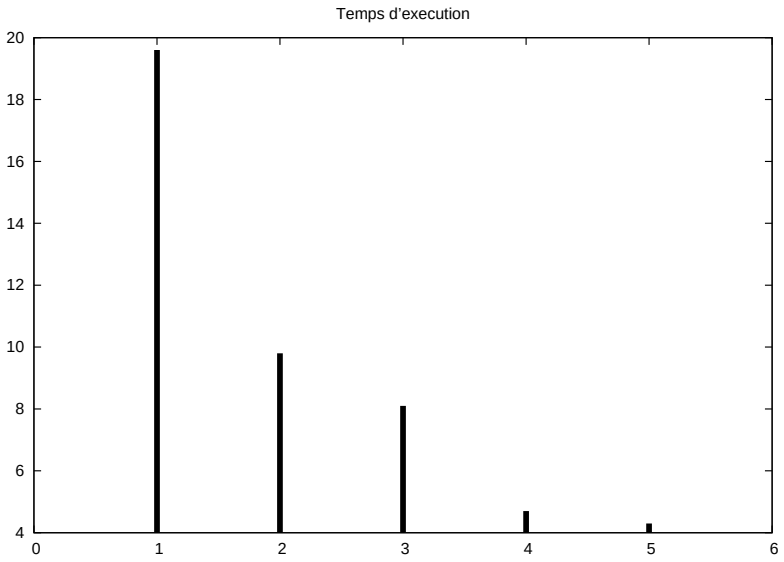

FIGURE 2.4 – Les temps d'exécution en fonction du nombre de modules fonctionnels pour la même fréquence de fonctionnement

Donc, nous devons rechercher un compromis entre la consommation d'énergie et la performance en agissant sur le nombre de RPM et la fréquence de fonctionnement.

ii. La contrainte de temps d'exécution : Pour une application devant traiter des données périodiquement, le traitement doit se terminer avant la période fixée. Pour simplifier, on peut considérer que la période de traitement est la contrainte de temps pour la tâche parallélisée. Pour un bloc de données à traiter, ce temps est constant.

Cependant, dans le cas d'applications multimédia par exemple, la contrainte de temps dépend de l'environnement du système où des paramètres tels que la vi-

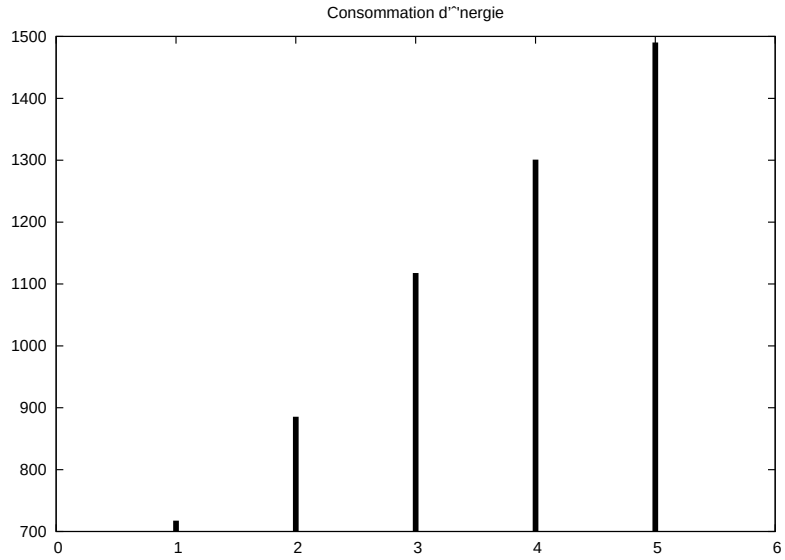

FIGURE 2.5 – Augmentation d'énergie en fonction de nombre de modules fonctionnels pour la même fréquence de fonctionnement

tesse de transmission, la qualité du service, la résolution des images à traiter, etc, peuvent varier. Dans ce cas, le système a besoins de réagir à ce changement en modifiant des paramètres architecturaux. Dans notre cas, le nombre de RPM et la fréquence d'exécution sont les paramètres d'adaptation. Nous notons T_d la contrainte de temps d'exécution, t_{RPM} est le temps d'exécution minimal d'un seul RPM. Si $T_d < t_{RPM}$, nous augmentons le nombre de RPMs pour réduire le temps d'exécution jusqu'à ce qu'il soit inférieur à T_d. On suppose une variation linéaire, t_{RPM}/N_2 est le temps d'exécution pour N_2 RPM travaillant en parallèle. Évidemment, il faut respecter la contrainte : $t_{RPM}/N_2 \leq T_d$. Donc le nombre de RPM minimal peut être calculé par l'équation2.12

$$N_2 = \frac{t_{RPM}}{T_d} \tag{2.12}$$

iii. La contrainte de temps de reconfiguration :

La variation du nombre de RPM est réalisée par la reconfiguration partielle. La

RPD n'est pas seulement utile pour modifier le nombre de RPM, mais aussi pour changer le type de RPM. Cependant, le temps de reconfiguration est toujours une contrainte importante pour étudier un système reconfigurable. Le temps de reconfiguration limite le nombre de RPM additionnels qui peuvent être ajoutés. Les travaux [BBD06,BBD09] présentent la relation entre la granularité du parallélisme et le temps d'exécution. Dans la figure 2.6, une relation entre la granularité du parallélisme et le temps de reconfiguration est formalisée. Si le temps d'exécution d'une tâche avec un RPM est T_{E1}, le temps de reconfiguration d'un RPM est T_{RPD}. Si on prend en compte du temps de reconfiguration T_{RPD}, le temps d'exécution total de deux RPM est $T_{RPD} + (T_{E1} - T_{RPD})/2$ (figure 2.6-b). On s'intéresse au temps d'exécution de la deuxième tâche (T_{E2}) d'après la configuration $T_{E2} = (T_{E1} - T_{RPD})/2$ (Regarder dans la figure 2.6-b). En parallélisant une tâche sur j RPM travaillant en parallèle, le temps d'exécution du j^{ime} module de calcul (T_{Ej}) est donné par l'équation 2.13.

$$T_{Ej} = \frac{((T_{RPD} - r_1 * (j * (j-1)/2)))}{j} \tag{2.13}$$

En se basant sur ce raisonnement, le temps d'exécution total (T_{total}) du système en prenant compte le temps de reconfiguration partielle dynamique (T_{RPD}) est donc donné par l'équation 2.14.

$$T_{totale} = j * T_{RPD} + \frac{((T_{E1} - T_{RPD} * (j * (j-1)/2)))}{j} \tag{2.14}$$

Où, j présente le niveau de parallélisme maximal, ce qui est présenté dans l'équation 2.15.

$$N_3 = j, t_j \leq r_1 \tag{2.15}$$

iv. La bande passante : Le nombre de RPM va dépendre également de la bande passante de l'interface de communication. Si la bande passante de l'interface est BP_{Total} bits/s et celle de chaque RPM est BP_{RPM} bits/s, nous pouvons connecter

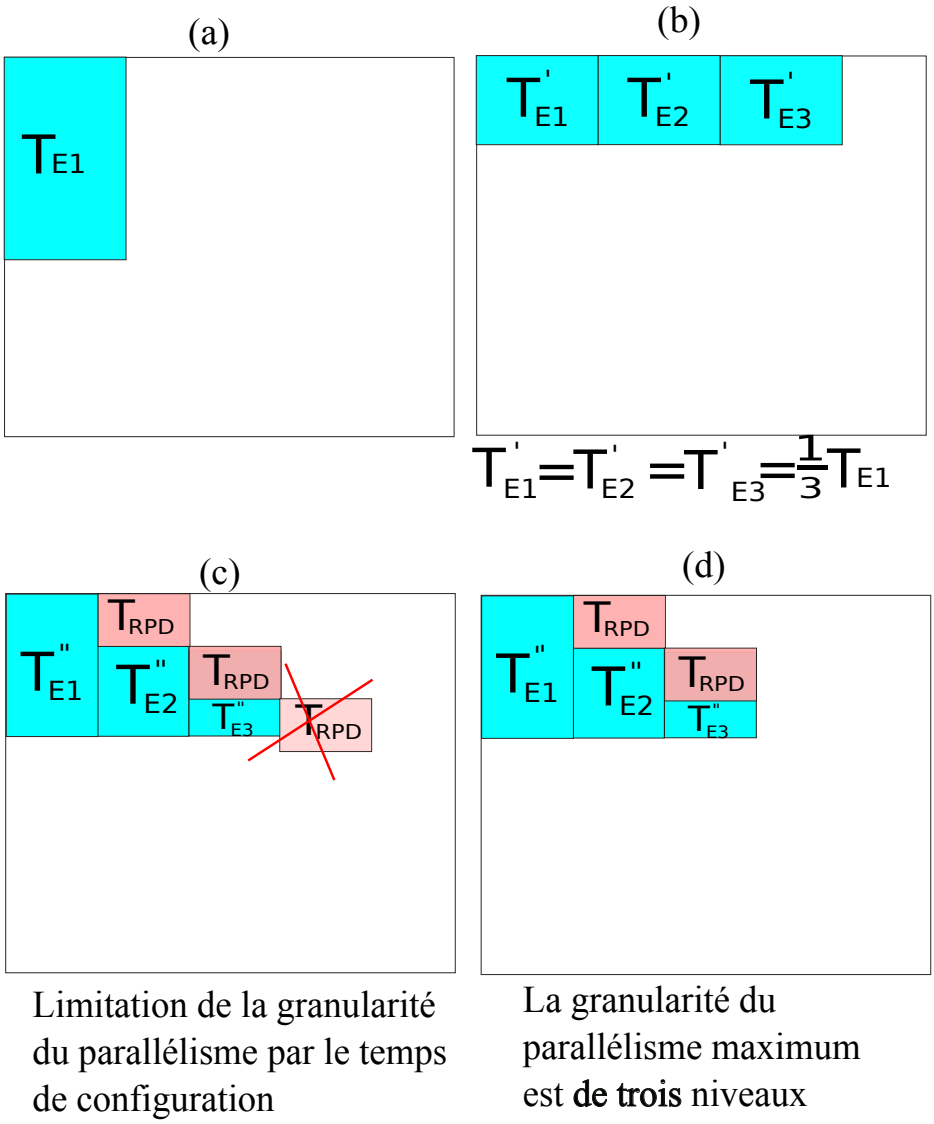

FIGURE 2.6 – Exemple de granularité de parallélisme et le temps de configuration partiel ; (a) un module de calcul fonctionne ; (b) trois modules de calcul fonctionnent en parallèle sans le temps de reconfiguration partiel ; (c) la granularité est limitée par le temps de configuration partielle ; (d) la granularité du parallélisme maximum sous condition du temps de configuration partielle

N_4 RPM au maximum. Ce nombre est donné par la partie entière P_E du rapport des deux bandes passantes (équation 2.16).

$$N_4 = P_E(\frac{BP_{Total}}{BP_{RPM}}) \tag{2.16}$$

Donc, le niveau de parallélisme maximal de l'architecture pour réaliser une tâche est la valeur minimale de l'ensemble des possibilités. Il est donnée par l'équation 2.17 :

$$N_{max} = min\{N_1, N_2, N_3, N_4\} \tag{2.17}$$

Nous savons que le nombre de RPM influe directement sur la consommation du système ainsi que sur les performances de calcul. Le but est de choisir un nombre juste suffisant de RPM et d'ajuster leur fréquence d'exécution pour obtenir le plus de performances en maîtrisant la consommation d'énergie pour une classe d'applications spécifiques et périodiques (2.2).

2.5.2 Contrôleur d'adaptation

Dans le paragraphe précédent, nous avons détaillé la partie fonctionnelle d'un système adaptatif. Des fonctions du système peuvent être modifiées à la demande. La décision de déclencher l'adaptation n'est pas incluse dans cette partie fonctionnelle. Un des objectifs du système adaptatif est de pouvoir réagir à différents événements comme, notamment, les variations observables avec le changement de l'environnement. La décision d'adaptation doit donc pouvoir être prise par le système lui-même. Ceci est réalisé dans un contrôleur sur puce que nous décrivons ci-dessous.

2.5.2.1 Rôle du contrôleur

Le but du contrôleur est de gérer l'état du système en exécution en utilisant les schémas prédéfinis. Nous développons notre contrôleur à partir d'un contrôleur d'adaptation classique [A.M95]. Ce dernier doit mettre en œuvre la stratégie de reconfiguration, c'est-à-dire la modification soit du nombre de RPM, soit de la fréquence d'exécution. L'objectif est d'optimiser les performances et la consommation du système en adaptant

le système à l'environnement. Dans ce cas, le contrôle est modélisé comme un processus qui fonctionne dans un certain nombre de régimes opératoires. Chaque régime opératoire correspond à un état de l'environnement, quand ce dernier change, le régime opératoire est obligatoirement modifié pour l'adapter au nouvel environnement du système. La figure 2.7 présente la structure d'un modèle de contrôleur adaptatif :

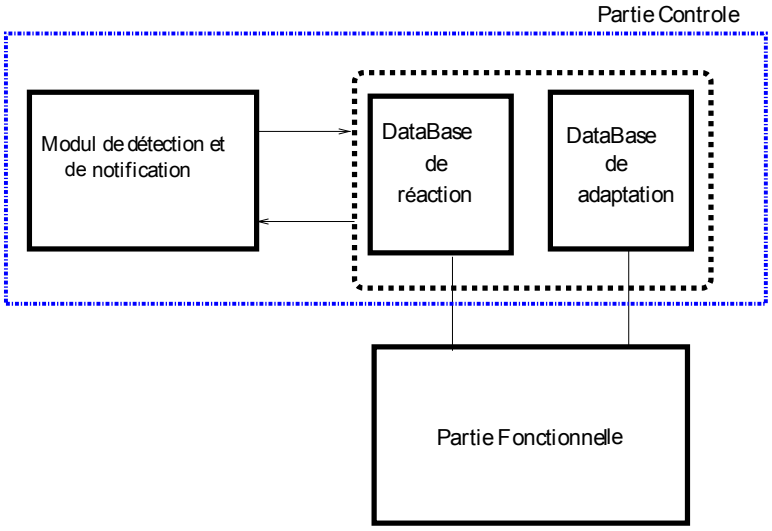

FIGURE 2.7 – Structure d'une partie contrôle du système d'auto-adaptation

Un module de détection et de notification est placé dans le contrôleur d'adaptation pour appliquer la stratégie d'adaptation en chargeant la configuration correspondante. La stratégie d'adaptation consiste à choisir dans une base de données de réaction, la configuration adaptée pour un régime d'opération de ce système. La base de données d'adaptation contient également les données de configuration des RPM.

Le contrôleur peut être implémenté comme un module matériel ou un module logiciel tel qu'un processeur RISC. Nous avons choisi cette dernière solution, plus souple dans la phase de développement. Une nouvelle génération de contrôleur sous forme matérielle est en cours de développement. Un exemple de ce contrôleur en version MicroBlaze de Xilinx est présenté sur la figure 2.8. Afin d'utiliser ce contrôleur, la génération des deux bases de données est importante pour indiquer quand et comment configurer le système. Dans la section suivante, une méthode d'adaptation permettant la génération des deux bases de données est proposée.

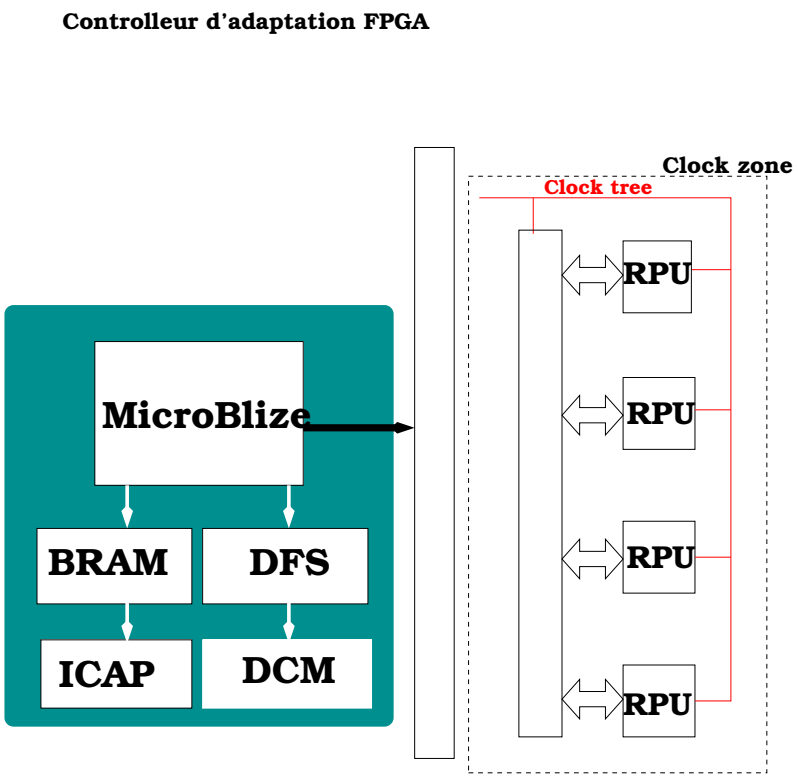

FIGURE 2.8 – Exemple d'un contrôleur en version MicroBlaze dans le circuit FPGA de Xilinx

2.6 Méthode d'adaptation dynamique

La solution proposée permet l'implémentation de chaque état du système de manière dynamiquement selon une stratégie d'adaptation qui prend en compte les sollicitations internes ou externes. La méthode est basée sur la détection des points d'inadaptation, cherche et applique aux différents modules fonctionnels du système les modifications nécessaires. Dans la méthode proposée, le système possède un ensemble d'états fini E=$\{$ E_0, E_1, $\cdots$, E_j, E_{j+1}, $\cdots$, $E_{\gamma-1}$ $\}$, $\gamma \in \mathbb{N}$. Chaque état correspond à un ensemble de contraintes du système. Lorsque les contraintes du système changent, le système doit passer d'un état à un autre afin de respecter les nouvelles contraintes.

2.6.1 Fonction de transition

Le passage du système d'un état à un autre est défini par une fonction de transition. Cette fonction de transition est une manifestation des contraintes du système. Dans cette étude, nous considérons deux contraintes x et y : la **contrainte de temps** et l'**efficacité énergétique**. La fonction de transition est notée : $X_{E_a->E_o}(x, y)$, où, E_a est l'état actuel du système, E_o est l'état optimal. La fonction $X_{E_a->E_o}(x, y)$ est égale 1 si les deux conditions sont vraies. Dans ce cas la transition d'un état du système à un autre est active. Le prochain état est déterminé par la relation de transition et l'état actuel.

Il existe une fonction de transition $X_{E_a->E_o}(x, y)$ entre chaque deux état du système. Par exemple, pour un état E_j, une série de valeurs $X_{E_a->E_o}(x, y)$ de cet état est définie par : $\{X_{E_j->E_1}(x, y), X_{E_j->E_2}(x, y), \cdots, X_{E_j->E_{j+1}}(x, y), \cdots, X_{E_j->E_{\gamma-1}}(x, y)\}$, $\gamma \in \mathbb{N}$. Le passage de l'état actuel à l'état optimal peut se faire de manière indirecte. En effet, si on prends le cas de la contrainte de temps, elle peut être satisfaite de plusieurs manières. Autrement dit, il peut exister plusieurs états du système pour lesquels la contrainte de temps peut être satisfaite lorsque les conditions changent. La recherche de l'état optimal peut se faire ainsi en deux étapes : recherches des états intermédiaires qui satisfont la contrainte de temps suivi de la recherche parmi ces états celui qui satisfait l'efficacité énergétique.

2.6.2 Contexte d'applications spécifiques pour un système temps réel

La méthode que nous proposons ne s'applique qu'à une classe d'applications, celle des applications sous contrainte de temps et de précédence avec des tâches parallélisables telles que des codeurs/décodeur JPEG, filtre de Sobel,etc. [BBMP04]. Dans ces applications, le temps d'exécution de certaines tâches, telles que IDCT, IDWT, Quantification, etc. est prévisible et borné. De plus, ces tâches présentent un fort parallélisme potentiel. Par exemple, les résultats de calcul de transformation sont indépendants pour chaque bloc de $8 * 8$ pixels. Sur une architecture RPD, il est possible d'améliorer le temps d'exécution par ajustement dynamique de la granularité du parallélisme de tâche. Par exemple, la reconfiguration de l'architecture permet d'instancier de mul-

tiples copies de ces tâches. Chaque instance utilise la même quantité de ressources matérielles, mais seulement une partie des données ($SIMD$). Le temps d'exécution de ces tâches est directement proportionnel au volume de données traitées, par conséquent le débit traitement augmente en théorie proportionnellement avec le nombre d'instances matérielles. Un exemple est illustré dans la figure 2.9. Dans ce cas, chaque tâche possède un temps d'exécution réel (T_{τ_1},T_{τ_2},T_{τ_3},T_{τ_4}) qui dépend du nombre d'unités de calcul instanciées et de la fréquence de fonctionnement qui peut être ajustée selon les besoins. Cependant, il existe également une contrainte propre à l'application, la contrainte d'exécution en temps réel. Donc, la somme des temps d'exécution des tâches (T_{EXE}=T_{τ_1}+T_{τ_2}+T_{τ_3}+T_{τ_4}) doit être inférieure ou égale à la contrainte de temps réel de l'application T_D. Cela est nécessaire pour garantir le fonctionnement de l'application.

Dans cette thèse, nous nous intéressons au cas de tâches parallélisables avec différents niveaux de parallélisme et exécutables à différentes fréquences de fonctionnement. Dans notre méthode d'adaptation, une base de données avec les deux paramètres π et η est construit afin d'identifier l'action d'adaptation et l'architecture optimal. Ces deux paramètres (π, η) dépendent respectivement du niveau de parallélisme et de la fréquence de fonctionnement. Les définitions des deux paramètres sont données par les équations 2.18 et 2.19.

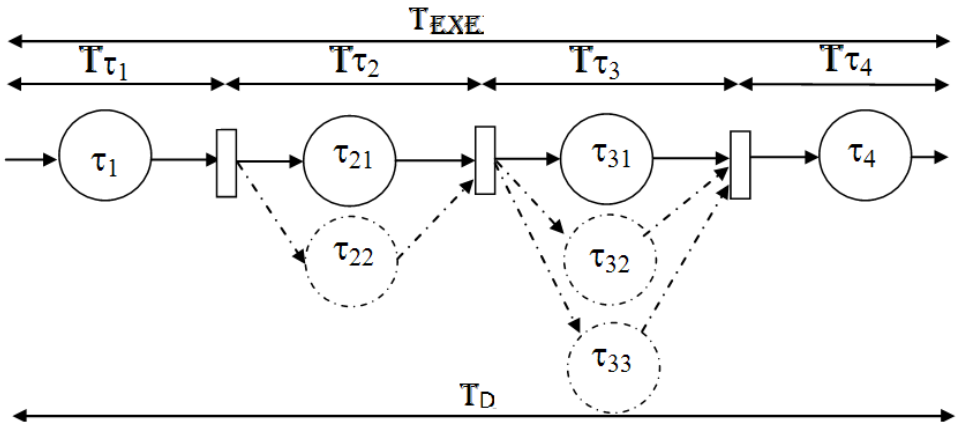

FIGURE 2.9 – Graphe de données d'une application sous contrainte de temps

Pour certains tâches parallélisables de cet exemple (par exemple les tâches : τ_2 et τ_3), nous définissons un ensemble de partitions de tailles identiques $\{\tau_{i1},\tau_{i2},\cdots,\tau_{in}\}$ où n est

le niveau de parallélisme maximal selon la définition présentée dans la section 2.5.1.1 par l'équation 2.17. Chaque unité fonctionnelle (RPM) est identique et exécutable en parallèle. La composante liée au taux de parallélisme est définie comme :

$$\pi_{\tau_i} = (L_{\tau_i} - L_{\tau_i,min})/(L_{\tau_i,max} - L_{\tau_i,min}) \tag{2.18}$$

Où $(L_{\tau_i,max})$ est le niveau de parallélisme maximal pour la tâche τ_i. L_{τ_i} est le niveau de parallélisme de la configuration courante. $L_{\tau_i,min}$ est le niveau de parallélisme minimal de la tâche τ_i. La définition de $L_{\tau_i,min}$ dépend de l'équation 2.12.

Par ailleurs, la fréquence de fonctionnement de chaque tâche est également modifiable. Chaque tâche possède un ensemble de fréquences d'exécution {$f_{i,min}$,$f_{i,1}$,$\cdots$,$f_{i,max}$ }. Nous nous référons à [ZRW08] pour définir un facteur de vitesse lié à la fréquence de fonctionnement par l'équation 2.19.

$$\eta_{\tau_i} = (f_{\tau_i} - f_{\tau_i,min})/(f_{\tau_i,max} - f_{\tau_i,min}) \tag{2.19}$$

où $f_{\tau_i,max}$ et $f_{(\tau_i,min)}$ représentent la fréquence de fonctionnement maximale et minimale de la tâche τ_i et f_{τ_i} est la fréquence de fonctionnement actuelle. La fréquence de fonctionnement maximale peut être estimée par les outils de synthèse. Ces définitions, nous permettent de calculer un ensemble de paramètres d'adaptation B ainsi que les valeurs de ses composantes $(\pi_{i,n}, \eta_{i,k})$ pour chaque tâche. Ces combinaisons permettent de construire différentes architectures du système pour réaliser l'application complète. B est représenté par la matrice ci-dessous :

$$\text{B=} \begin{bmatrix} (\pi_0,\eta_0) & (\pi_0,\eta_1) & \cdots & (\pi_0,\eta_n) \\ (\pi_1,\eta_0) & (\pi_1,\eta_1) & \cdots & (\pi_1,\eta_n) \\ \vdots & \vdots & \ddots & \vdots \\ (\pi_m,\eta_0) & (\pi_m,\eta_1) & \cdots & (\pi_m,\eta_n) \end{bmatrix}$$

La matrice 2.6.2 représente toutes les configurations d'architectures possibles.

Selon l'équation **??**, quand la contrainte de temps d'exécution du système change, le contrôleur d'adaptation cherche une nouvelle architecture optimisée pour s'adapter

au nouvel environnement. Cela revient à trouver le couple optimal de valeurs (π, η) dans la matrice **??** à partir de la valeur de la nouvelle contrainte de temps d'exécution.

2.6.3 Prise en compte de l'efficacité énergétique

Dans notre travail nous nous intéressons à l'efficacité énergétique d'applications traitant des données périodiques, comme le précise la définition de l'efficacité énergétique vue dans le chapitre précédent. Nous définissons la règle pour trouver l'architecture optimale selon le critère d'efficacité énergétique comme suit :

Au cours de la période d'arrivée des données, l'architecture possédant l'efficacité énergétique optimale est celle dont le temps de traitement est le plus proche de la contrainte de temps d'exécution tout en étant inférieur à cette dernière.

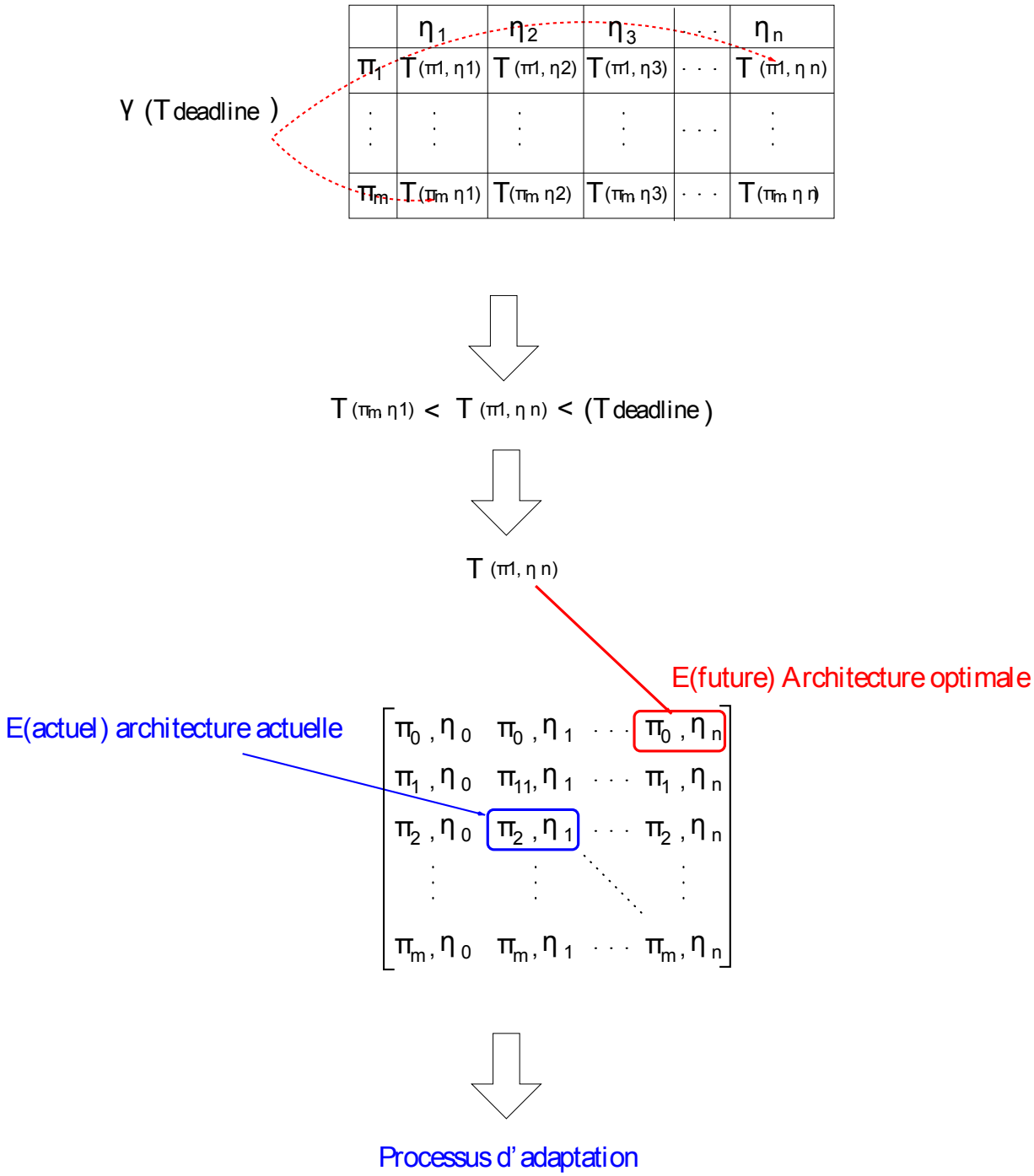

FIGURE 2.10 – Le processus de configuration d'architecture

Pour trouver l'architecture optimale du point de vue de l'efficacité énergétique, il faut comparer le temps d'exécution effectif de l'architecture avec la contrainte de temps. La figure 2.10 présente le schéma pour trouver l'architecture optimale. Tous les temps d'exécution liés aux paramètres architecturaux $B = \{(\pi_i,\ \eta_i)\}$ peuvent être préparés dans une base de données du système. En comparant la contrainte de temps $T_{deadline}$ avec les temps effectifs d'exécution, les architectures candidates sont identifiées dans 2.10-1. Parmi elles, les caractéristiques (couple de valeurs (π, η)) de l'architecture optimale sont trouvées en cherchant le temps d'exécution le plus élevé. Les étapes de la stratégie d'adaptation sont présentées ci-après.

2.6.4 Étapes de l'adaptation et leur objectif

L'adaptation dynamique doit produire les réponses aux deux questions suivantes :

Premièrement, combien de modules de calcul ? et quelle fréquence utiliser ?

Deuxièmement, comment réaliser la stratégie de configuration ?

En étudiant les deux questions ci-dessus, nous proposons une méthode permettant, à partir d'une estimation des ressources exploitées du système, une configuration en temps réel du système pour s'adapter aux changements de son environnement interne et/ou externe. Cette méthode exige trois étapes qui sont illustrées dans la figure 2.11.

2.6.4.1 Construction de la base de données du système

La précision de l'adaptation dépendant de l'étendue des choix d'architectures. Dans notre travail, cela est lié à la taille de la matrice 2.6.2. Plus il y a d'architectures cibles, plus le système a de choix pour s'adapter à la nouvelle contrainte de temps. Cependant, le nombre de RPM est limité par la taille du circuit et le temps de configuration, etc. (détaillé dans la section 2.5.1.1). La fréquence d'exécution est limitée par les caractéristiques de la cible, les capacités du générateur/synthétiseur d'horloge et les besoins de la classe d'application (pour la limite inférieure).

La construction de la matrice est la première étape du processus, elle est réalisée hors ligne, pour chaque tâche de la classe d'applications. C'est l'estimation des carac-

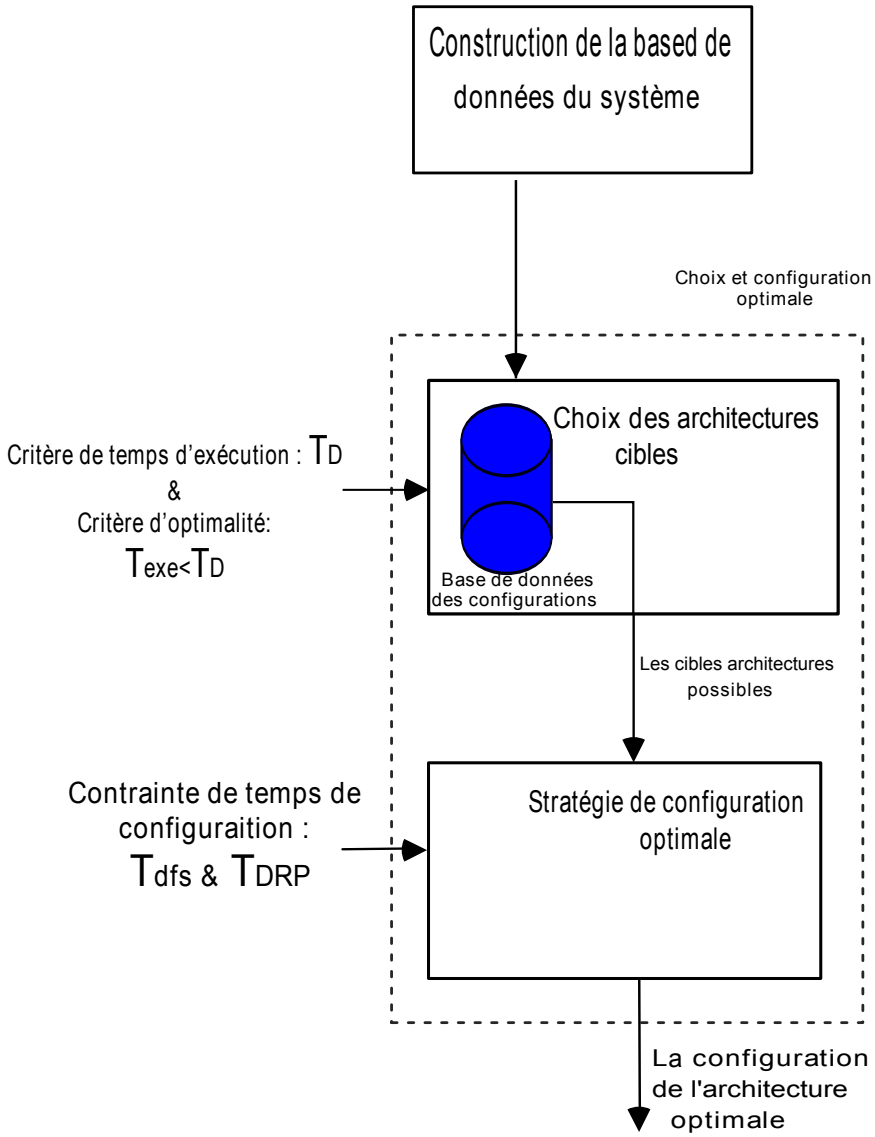

FIGURE 2.11 – Étapes de configuration d'architecture

téristiques de l'architecture du système. Le but est de créer une base de données des caractéristiques du système. Elle est utilisé pour identifier l'action d'adaptation et trouver l'état optimal en correpondant le changement de contrainte du système.

Dans cette thèse, nous estimons le temps d'exécution en fonction des combinaisons de π_i et η_i. Nous arrangeons ces architectures possibles dans une base de données 3.5.3.1. Le tableau 3.5.3.1 nous indique les valeurs de temps d'exécution pour une tâche de l'application sur différentes architectures pour différents couples $(\pi,\ \eta)$.

TABLE 2.1 – Base de donnée de temps d'exécution avec la combinaison des paramètres du système (π, η)

	η_1	η_2	$\cdots$	η_n
π_1	$T_{\tau_i}(\pi_1, \eta_1)$	$T_{\tau_i}(\pi_1, \eta_2)$	$\cdots$	$T_{\tau_i}(\pi_1, \eta_n)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
π_n	$T_{\tau_i}(\pi_m, \eta_1)$	$T_{\tau_i}(\pi_m, \eta_2)$	$\cdots$	$T_{\tau_i}(\pi_m, \eta_n)$

Ce tableau est estimé statiquement avec des outils de simulation et estimation fournis par le fabricant de plate-forme FPGAs. Une illustration d'utilisation de ce type de tableau sera présentée dans le paragraphe suivant.

2.6.4.2 Choix et configuration de l'architecture optimale

Dans cette section, la recherche de l'architecture optimale adaptée ainsi que sa configuration vont être présentées. Le but est de choisir une architecture optimale pour le nouvel environnement. Par ailleurs, le système d'adaptation va élaborer une séquence de configurations transistoires pour garantir la continuité de fonctionnement.

Le choix de l'architecture adaptée optimale :

La détection de la nécessité d'adaptation est activée par le changement des contraintes A. Un système de supervision de l'application ou un système de mesure des contraintes, qui n'est pas présenté ici, signale qu'il faut adapter le système à un nouvel environnement. La réaction d'adaptation, modélisée par X_i, va signaler les architectures adaptées. Dans cette thèse, l'adaptation est définie par rapport à la contrainte de temps. Ce problème est détaillé ci-dessous :

Critère de temps d'exécution : Le temps d'exécution du système dépend de l'architecture (le nombre de ressources à calculer, la consommation d'énergie, la température du système, etc.). Le critère de temps d'exécution T_{exe} est une fonction du temps d'exécution et de la contrainte de temps T_D. Ce critère permet de recenser les architectures adaptées. Pour ce faire, une base de données des temps d'exécution a été créée dans la première étape. Les architectures adaptées parmi B_i peuvent être trouvées en utilisant l'algorithme 2.1. Cela correspond à toutes les architectures pour lesquelles la fonction caractéristique $X_i(a_j, b_i)$ est égale à un.

Critère d'optimalité : Pour trouver l'architecture optimale parmi les architectures adaptées, le critère utilisé est la consommation d'énergie moyenne pendant la contrainte de temps. L'analyse de la consommation d'énergie dans la section 2.5 montre qu'il faut que le temps d'exécution soit le plus proche de la contrainte de temps tout en étant inférieur. C'est le deuxième cas illustré dans la figure 2.12.

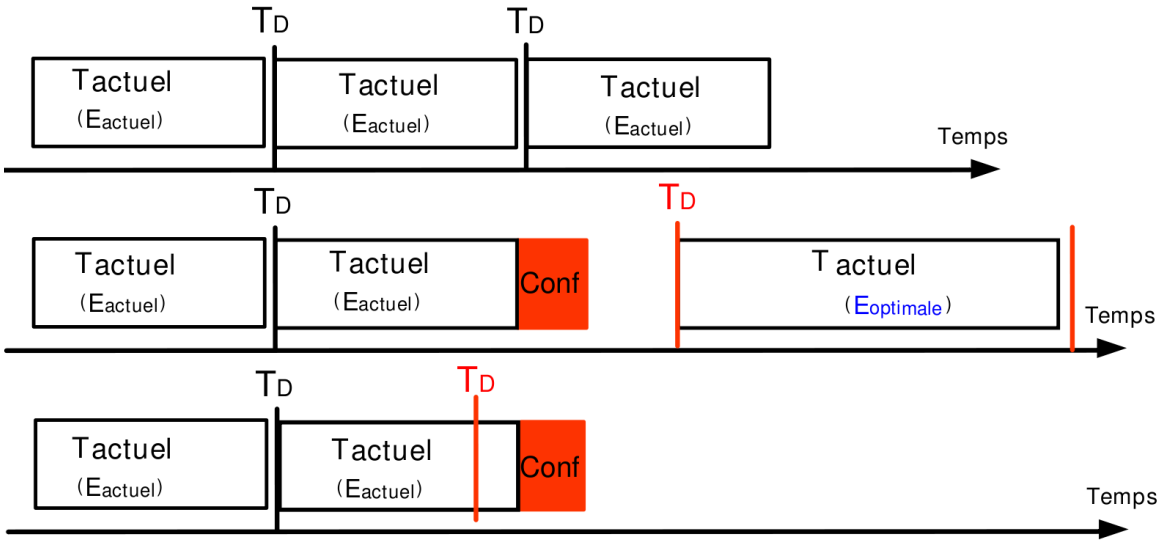

FIGURE 2.12 – Les trois cas de relations entre la contrainte de temps et le temps d'exécution réel

La stratégie de configuration optimale :

Nous connaissons maintenant la configuration optimale à placer dans le système. Pour charger cette configuration, il faut utiliser une partie de la contrainte de temps, il n'est pas toujours possible de configurer immédiatement l'architecture optimale. Il

Algorithm 2.1 – Choix de l'architecture optimale

Require: la base de données de temps d'exécution de la tâche τ_i (tableau de $T_{\tau_i}(\pi, \eta)$ et la contrainte de temps T_D

Ensure: $T_{\tau_i}(\pi, \eta)$ est inférieur à T_D

for all Architecture actuel de tâche τ_i : $B_{\tau_i}(\pi_j, \eta_k)$ **do**
  Recherche $Slack = T_d - T_{\tau_i}(\pi_j, \eta_k)$
  if $Slack < Slack_{min}$ and $Slack \geq 0$ **then**
    $B_{optimale}(\pi_{optmail}, \eta_{optimal}) = B_i(\pi, \eta)$
  end if
end for

faut parfois passer par une configuration transitoire qui permet, moyennant une augmentation temporaire de la consommation d'énergie, de traiter plus rapidement pour compenser le temps perdu pour configurer.

1. la contrainte de temps est supérieure au temps d'exécution :

   Quand T_D est supérieur au temps d'exécution effectif plus le temps de configuration de l'architecture optimale (T_{RPD} ou T_{DFS} ou les deux), la configuration peut être effectuée pendant la période d'adaptation. C'est le cas dans la figure 2.13-B.

2. la contrainte de temps est inférieure au temps d'exécution :

   Quand la nouvelle T_D est inférieure au temps d'exécution actuel et de configuration (cf. 2.13-C), une configuration transitoire est utilisée afin de ganratir la continuité de fonctionnement. En utilisant la reconfiguration partielle dynamique, nous pouvons calculer le niveau de parallélisme pour configurer et finir le travail tout en respectant le changement de contrainte de temps. La méthode pour calculer le nouveau temps d'exécution $T_{\tau_i}(\pi, \eta)$ et le niveau de parallélisme η sont données dans l'équation2.14 tirée de [BBD09].

La configuration transitoire est présenté par l'algorithme 2.2 dans lequel T_{RPD} est le temps de configuration partielle pour un RPM et T_{DFS} est le temps pour générer un

nouvel fréquence de fonctionnement.

Algorithm 2.2 – Stratégie de configuration

Require: la base de données de temps d'exécution ($T_{exe}(\pi, \eta)$), le temps de configuration d'un RPM : T_{RPD}, le temps de configuration de la fréquence de fonctionnement : T_{DFS} et la contrainte de temps : T_D

Ensure: $T_{exe} + T_{RPD} + T_{DFS} \leq T_D$

1: Si $T_{exe} + T_{RPD} + T_{DFS} \leq T_D$ la configuration optimale peut être chargée immédiatement.

2: Sinon il faut passer par une configuration comportant j unités RPM. La valeur de j est donnée par :

$$\frac{-T_{RPD} - \sqrt{T_{RPD}^2 + 8 * T_{exe} * T_{RPD}}}{2 * T_{RPD}} \leq j \leq \frac{-T_{RPD} + \sqrt{T_{RPD}^2 + 8 * T_{exe} * T_{RPD}}}{2 * T_{RPD}}$$

3: si j est déjà le niveau de parallélisme maximum, on augmente la fréquence de fonctionnement au maximum pour obtenir la performance maximum.

4: la période suivante, on peut revenir à la configuration optimale en effaçant les unités RPM en trop par rapport à la configuration optimale.

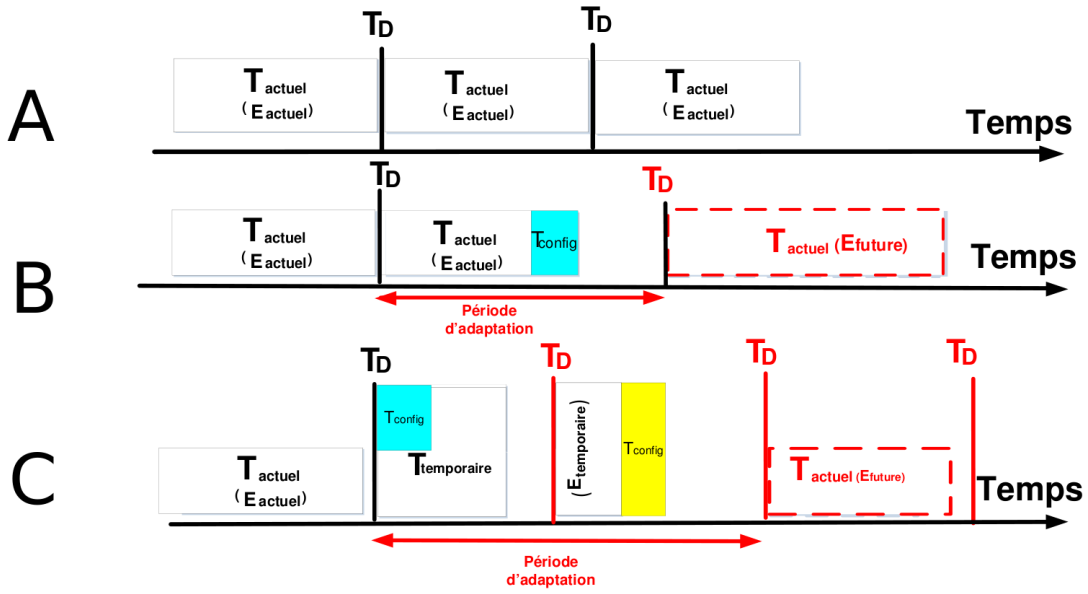

FIGURE 2.13 – Stratégies de la reconfiguration

2.6.5 Méthode d'adaptation d'un traitement multi-tâches

Jusqu'à présent nous n'avons abordé que le cas de système ayant à traiter une seule tâche. Nous allons montrer le processus d'adaptation à utiliser dans le cas d'une application multi-tâches. Nous supposons que les tâches sont à exécuter en parallèle et en pipeline avec une dépendance de données au niveau tâche. Un exemple classique est illustré dans la figure 2.14. A la différence de l'architecture avec une seule tâche, il y a plusieurs tâches à exécuter pendant une période et chaque tâche peut avoir un temps d'exécution différent.

L'adaptation est divisée en deux parties : **adaptation en performance** et **adaptation en efficacité énergétique** comme dans le cas de l'adaptation mono tâche.

La stratégie de configuration pour la première partie respecte *la règle du First-Fits* qui est répétée jusqu'à la dernière tâche. Notons que cela nécessite une période d'exécution pour chaque tâche, comme l'illustre la figure 2.14. L'algorithme 2.2 est appliqué sur chaque tâche à configurer pour garantir la continuité de la fonction globale. Á cause de la limite de la technique de la reconfiguration partielle dynamique, une seule tâche peut être modifiée pendant chaque période de traitement.

L'algorithme 2.1, qui correspond à la seconde partie, suit *la règle de Best-Fits* qui permet de configurer la solution optimale.

2.6.6 Résumé du fonctionnement du système d'adaptation

Ici, nous proposons un processus d'adaptation (illustré dans la figure 2.15) qui résume les trois étapes d'adaptation déjà présentées : l'initialisation du système, le choix de l'architecture adaptée optimale et la stratégie de configuration illustrés dans la figure2.15-a. Supposons que le système est dans un état optimal d'efficacité énergétique quand une demande d'adaptation externe survient. Cela indique le changement d'environnement du système et implique une nouvelle contrainte de temps T_D. Ensuite, le temps d'exécution effectif T_{exe} est soustrait de la contrainte de temps T_D et le résultat conditionne la prochaine étape du processus. Si la différence est inférieure à zéro, l'algorithme de configuration transitoire est lancé pour garantir la continuité de la fonc-

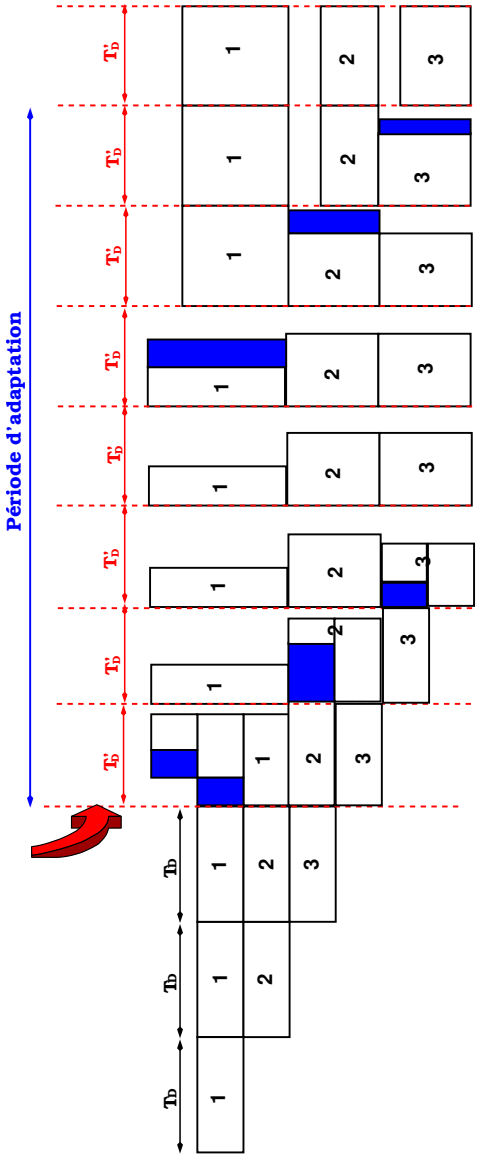

FIGURE 2.14 – exemple d'un système d'adaptation d'une application multi-tâche en pipeline

tionnalité du système pendant la contrainte de temps illustré dans la figure 2.15-b. L'algorithme 2.2 est alors utilisé pour calculer le niveau de parallélisme de la configuration transitoire qui est appliquée grâce à la reconfiguration partielle dynamique. Cependant, quand le niveau de parallélisme est déjà maximal, l'architecture la plus performant est choisie pour garantir le fonctionnement du système.

Si la différence est supérieure à zéro, cas 2.15-c, nous passons à l'étape de configuration optimale. Dans cette étape, l'algorithme 2.1 est utilisé pour déterminer l'architecture optimale. La reconfiguration partielle et/ou la gestion dynamique de fréquence de fonctionnement sont lancées pour réaliser l'architecture optimale dans la dernière étape 2.15-d.

2.7 Conclusion

Dans ce chapitre, nous avons présenté notre travail qui vise principalement à définir un modèle d'architecture et une méthode pour l'adaptation dynamique. Cette architecture prend en compte le taux de parallélisme et les différentes fréquences d'exécution. Il est capable de choisir la stratégie de configuration qui permet de garantir la continuité de fonctionnement en apportant les modifications nécessaires. De plus, l'optimisation est réalisée grâce à la sélection des paramètres de l'architecture qui minimisent la consommation de puissance du système.

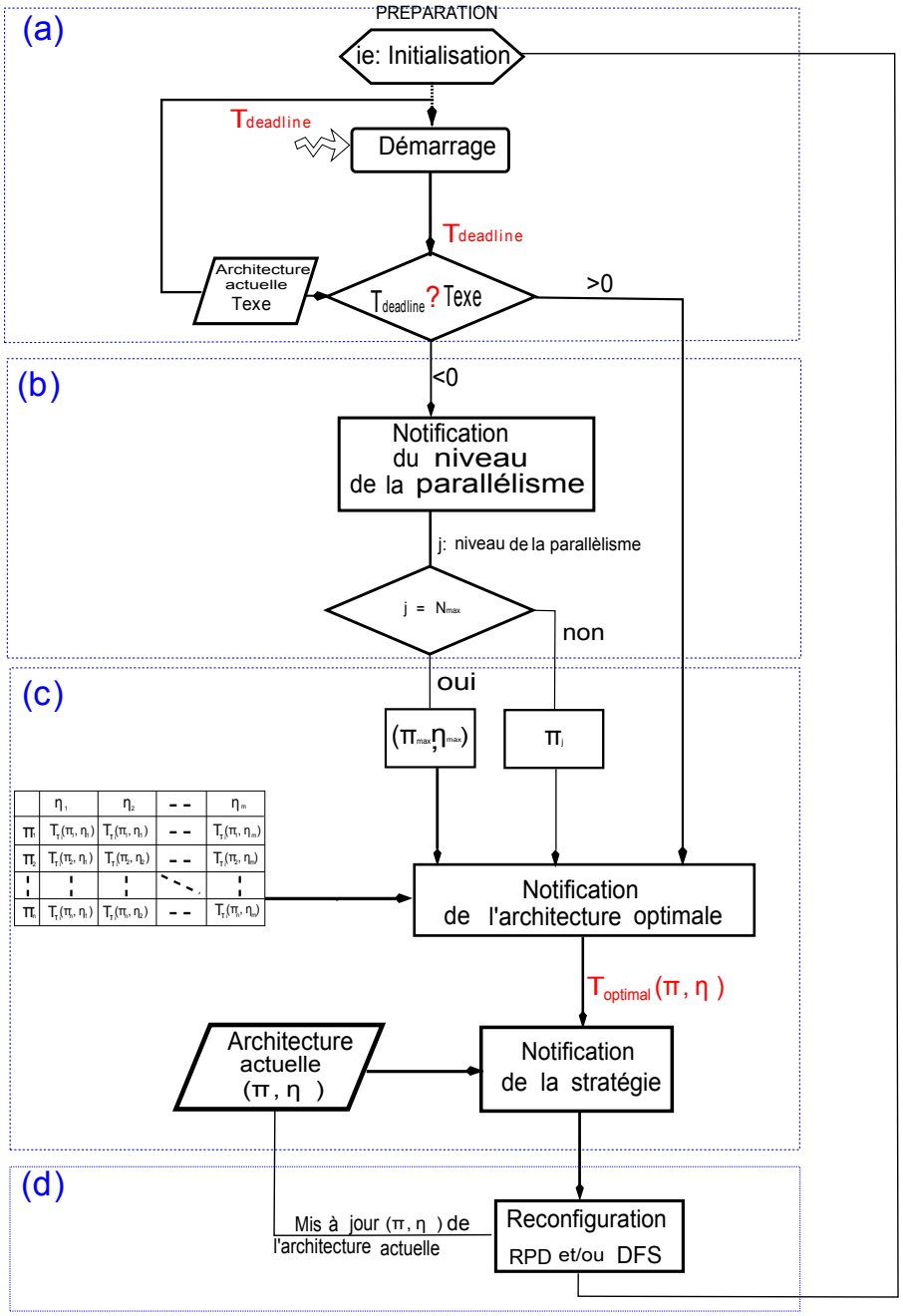

FIGURE 2.15 – le flot d'adaptation

Chapitre 3

Étude et validation expérimentale

Sommaire

3.1 Introduction

Dans le chapitre précédent, un modèle d'architecture et une méthode d'adaptation dynamique ont été proposés. La méthode défini l'état optimal de l'architecture et le choix de la stratégie de configuration. Afin de mettre en œuvre cette méthode, la plate-forme expérimentale et le flot de conception utilisés seront d'abord présentés. Ensuite, l'algorithme de la transformée en ondelettes ainsi que l'architecture cible pour valider la méthode proposée seront détaillés. Les résultats expérimentaux sur le compromis efficacité énergétique et performance dans le cas de l'adaptation aux variations de la quantité de données seront présentés et discutés. En fin, les détails de la réalisation de la méthode proposée sont présentés avant la conclusion.

3.2 La plate-forme expérimentale

Dans le but de démontrer la faisabilité de la méthode d'auto-adaptation proposée, nous avons choisi une plate-forme reconfigurable basée sur le circuit FPGA Virtex-4 [Inc04a] de la famille Xilinx (pour plus de détails voir l'annexeA). Le choix est justifié par le fait que cette famille de circuits est la seule disponible offrant les caractéristiques nécessaires pour l'auto-adaptation. Ces caractéristiques sont les suivantes :

- **Reconfiguration partielle**
- **Possibilité d'auto-reconfiguration**
- **Gestion dynamique de fréquence (DFS)**

La figure 3.1 illustre un modèle générique de l'architecture cible composé d'éléments nécessaires pour réaliser l'auto-adaptation. On distingue en particulier les unités reconfigurables partiellement (RPU), le port permettant l'auto-reconfiguration (ICAP), le module de la gestion dynamique de la fréquence (DCM) et les éléments de contrôle de ces différentes interfaces.

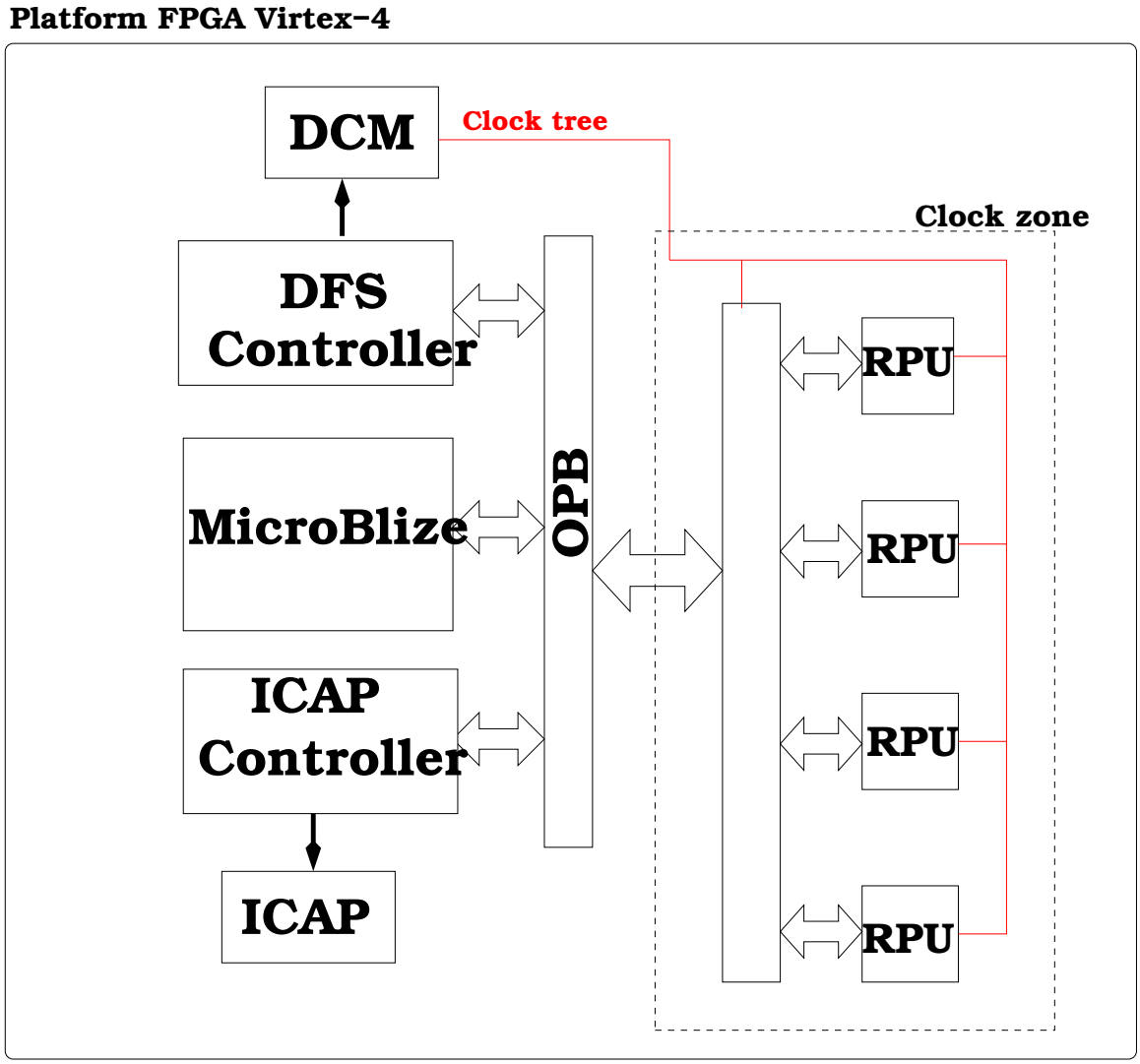

FIGURE 3.1 – La plat-forme auto-adaptative basée sur circuit FPGA de Xilinx

3.2.1 Étude préliminaire

3.2.1.1 Contrainte de reconfiguration

Dans toute la famille Virtex, les données de configuration sont envoyées vers une interface de configuration. La famille des circuits FPGA Xilinx offre trois modes d'interface de configuration : JTAG, SelectMap,et ICAP. Ceux-ci sont employés pour configurer le circuit par un ordinateur hote ou par un processeur intégré dans le FPGAs. Dans le cas d'une configuration par un processeur intégré, l'interface ICAP [6] est utilisée.

Une des contraintes principales de la reconfiguration est le temps de reconfiguration. Celui-ci dépend de l'interface de reconfiguration utilisée et de l'emplacement de données de configuration. La configuration par l'interface ICAP est la plus rapide par rapport aux autres interfaces. Pour cette interface, les données de configuration sont transférées sous forme de mots de $32\ bits$ avec une fréquence maximale de fonctionnement de $100\ MHz$, permettant ainsi un débit maximal de $400\ Mo/s$. Cependant, les accès mémoire limitent ce débit. Cette limitation est d'autant plus importante que le contrôleur utilisé est un processeur. Le type de mémoire utilisé, interne ou externe, a également une influence sur le débit.

Dans cette étude, nous avons utilisé une architecture basée sur un processeur pour faciliter la gestion de configuration. Cette architecture utilise un périphérique matériel HWICAP qui contient le port ICAP.

Le processeur transfère les données de la mémoire DDR vers le tampon BRAM du périphérique HWICAP. Lorsque le tampon est plein, le périphérique HWICAP transfère les données de configuration du tampon vers ICAP (3.2).

Dans le cas où on utilise une trame comme unité, le temps de configuration en utilisant HWICAP est :

$$T_{conf} = N_{trame} * (T_{BRAM} + T_{ICAP}) \tag{3.1}$$

Où, N_{trame} est le nombre de trames dans le bitstream, T_{BRAM} est le temps de transfert d'une trame de la mémoire DDR vers la BRAM de HWICAP, et T_{ICAP} est le temps

6. Internal Configuration Access Port

de transfert d'une trame de la BRAM vers ICAP.

Le nombre de trames dans un bitstream est obtenu par relation suivante :

$$N_f = \frac{\text{La taille du bitstream en octets}}{\text{La taille d'une trame en octets}} \tag{3.2}$$

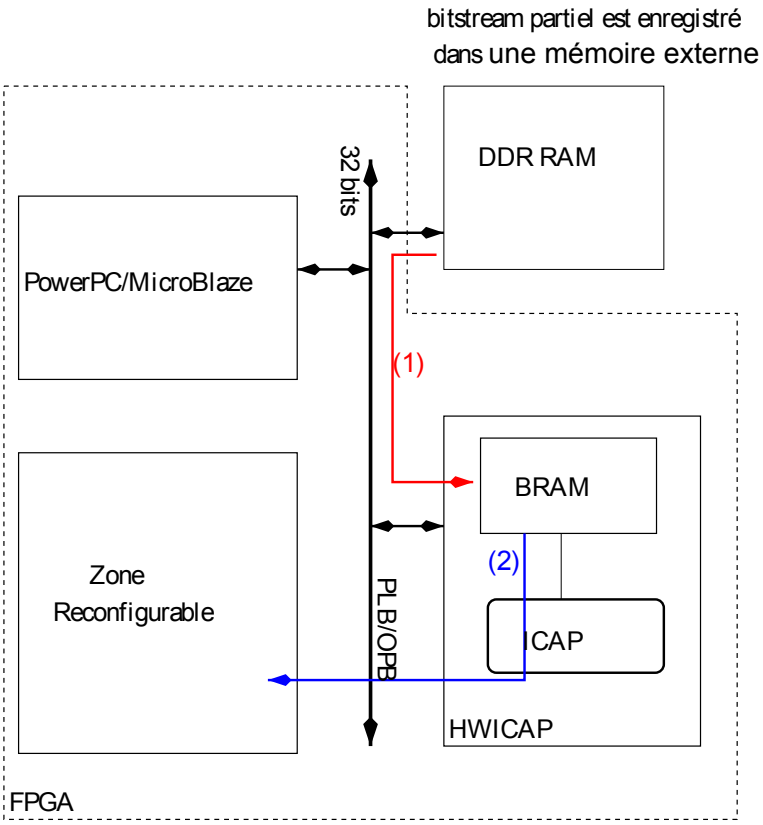

FIGURE 3.2 – Processus de reconfiguration partielle dynamique

Le temps de reconfiguration du FPGA Virtex-4D est déterminé expérimentalement. Le tableau 3.2.1.1 donne un exemple de temps de configuration d'un système composé d'une partie statique et d'une partie dynamique [ZRW07]. Ces temps ont été obtenus expérimentalement en utilisant un compteur matériel.

Le temps de configuration du circuit virtex-4 est obtenu par l'équation 3.1 et est égal à $0.07ms$, ce qui correspond à la valeur mesurée en pratique donnée dans le tableau 3.2.1.1. Si on considère le temps de configuration total du module reconfigurable, une grande partie de temps est perdue dans le transfert entre la mémoire DDR et le BRAM de HWICAP à cause de l'utilisation du processeur $9.7ms$. Ce temps de

TABLE 3.1 – Exemple de temps de reconfiguration

Parties du Système	Taille KB	Contrainte de temps (ms)		
		T_{BRAM}	T_{ICAP}	T_{config}
Partie statique	582	by JTAG	\	2 seconds
Partie dynamique	28	9.7	0.07	9.77

reconfiguration partielle est obtenu expérimentalement en utilisant un compteur matériel. Pour un bitstream $28koctets$, le temps de configuration est $0,07ms$. Ce dernier correspond aux valeurs théoriques du circuit FPGA virtex-4 de Xilinx. De plus, la taille d'une trame (16CLBs) pour toutes les conceptions Virtex4 est de 164 $octets$. Le temps de configuration pour une trame est donc $0,05ms$.

En utilisant un contrôleur matériel spécifique à la place du processeur, nous pouvons réduire considérable le temps de configuration et s'approcher de la valeur minimale caractéristique du circuit.

3.2.2 Gestion dynamique de fréquence

Les circuits FPGAs de Xilinx, tels que Virtex-2,4 et 5,offrent la possibilité de gestion dynamique des fréquences (voir paragraphe 1.5.1.1).

Afin d'implémenter la gestion dynamique de fréquence (DFS), un contrôleur est utilisé pour modifier la fréquence par l'intermédiaire du gestionnaire dynamique de l'horloge (DCM). Un exemple de contrôleur est illustré dans la figure 3.3. Dans cette figure, le microprocesseur MicroBlaze/PowerPC est utilisé pour gérer automatiquement la configuration du DFS via le contrôleur. Ce contrôleur est un périphérique connecté sur le bus système PLB (voir l'annexe A).

La contrainte de reconfiguration est liée principalement au temps de configuration du contenu de la BRAM de DCM. La configuration nécessite 15 $cycles$ d'horloge, ce qui est vérifié dans le résultat de simulation après le placement et routage illustré par la

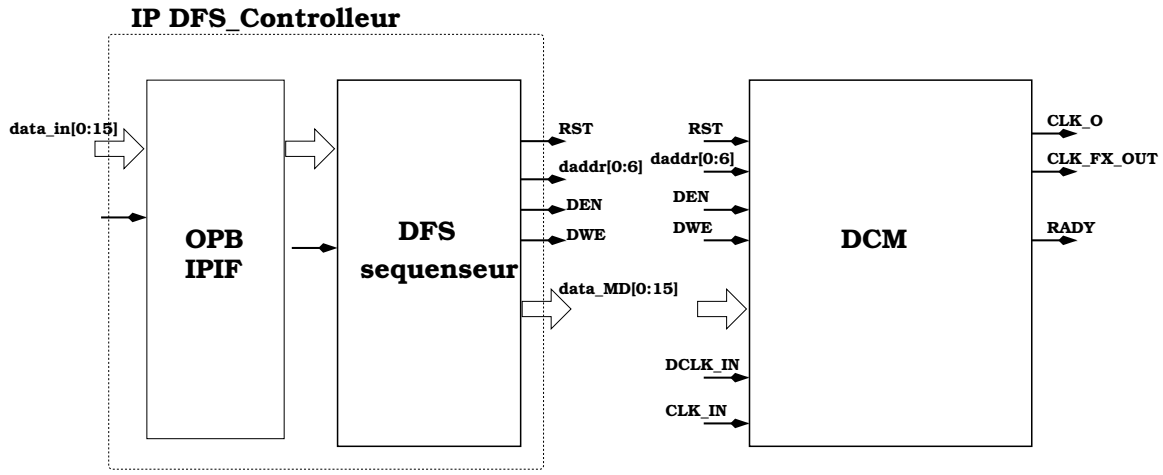

FIGURE 3.3 – Contrôleur de DFS

figure 3.4. La fréquence de l'horloge de configuration est $100\ MHz$ dans notre exemple, ce qui donne un temps de configuration $150\ ns$.

Une limitation de cette méthode est le nombre de fréquences de fonctionnement qui peut être généré par le composant DCM. La génération de fréquence par le DCM est basée sur le principe de multiplication et division de fréquence par des nombre entiers. Le tableau3.2 montre la gamme de fréquences ($Fout$) possibles. La fréquence d'entrée est de $1.000MHz$ à $300MHz$. La fréquence sortie maximale est de $300MHz$ et la fréquence minimale est de $32MHz$.

TABLE 3.2 – Gammes validées pour la grade de vitesse

DFS mode	Fin (MHz)	Fout (MHz)
bas	1.000 -210.000	32.000-210.000
haut	50.000-300.000	210.000 - 300.000

3.3 Application jpge2000

Le standard de compression d'images JPEG2000 [jpe] propose un large ensemble de fonctionnalités [TP04, ACT01] utiles pour les applications multimédias. Parmi ces es fonctionnalités on trouve :

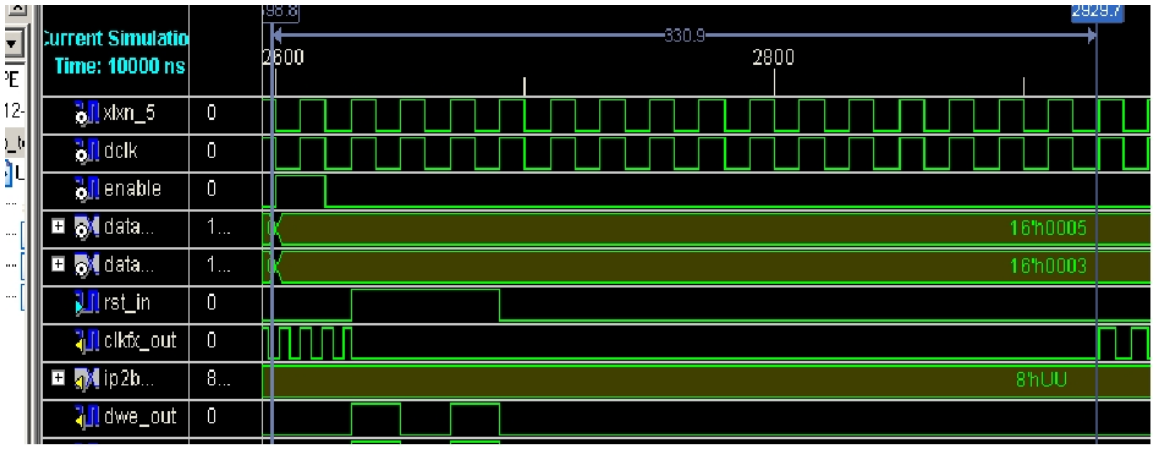

FIGURE 3.4 – Contrôleur de DFS

– Compression avec et sans pertes ;
– Excellente qualité d'image ;
– Scalabilité ;
– Région d'intérêt ;
– Très grand taux de compression ;

Les codeurs et décodeurs JPEG2000 sont constitués de différents blocs fonctionnels 3.5. La structure du décodeur est l'inverse de celle du codeur à l'exception de l'étage du contrôleur de débit. Le système de compression est simplement divisé en trois phases [TP04, Ada01] : Prétraitement d'image ; compression ; organisation du bits-tream.

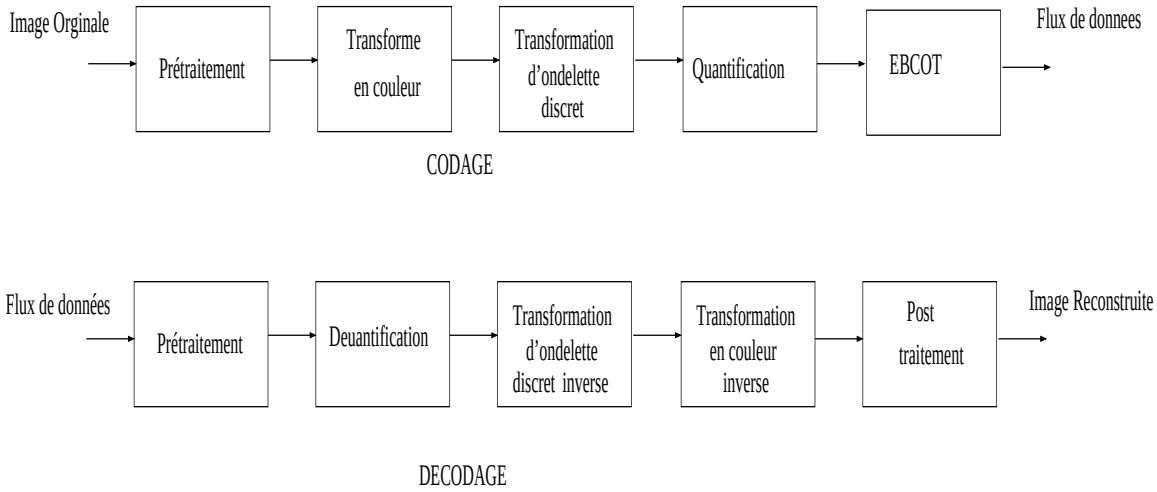

FIGURE 3.5 – Blocs constituant la chaîne de codage et décodage de JPEG2000

Transformée en couleur Cette phase est optionnelle. La transformée en couleur est utilisée dans le cas des images appartenant à l'espace de couleur RVB. Elle permet alors d'obtenir une représentation de l'image dans l'espace de couleur $YCrCb$.

Transformée en ondelettes discrètes C'est ici que l'algorithme de JPEG2000 est radicalement différent de celui de JPEG. Alors que le JPEG utilisait une transformée en cosinus discrète DCT, JPEG2000 utilise les ondelettes. JPEG2000 a sélectionné dans un premier temps deux types de filtres fournissant les meilleurs résultats en termes de décorrélation. Le premier filtre $5/3$, proposé dans [RIWB98] est réversible, il peu être utilisé pour la compression sans pertes. Le deuxième filtre $9/7$, proposé dans [MMPI92] est non réversible, utilisé seulement dans le cas d'une compression avec pertes. Cette tâche parallélisable offre une possibilité de modifier la granularité du parallélisme.

Quantification Quand on code avec pertes, la précision sur les coefficients d'ondelettes obtenus à l'étape précédente est réduite par ce qu'on appelle une quantification scalaire uniforme. On ne conserve qu'un ordre de grandeur plus ou moins précis des coefficients.

Codage entropique Il s'agit maintenant d'appliquer un codage sans pertes aux données obtenues. Le JPEG2000 utilise un codage arithmétique adaptatif avec contexte.

Allocation de débit Il s'agit de fabriquer les paquets de données conformes à la norme qui seront ensuite transmis ou enregistrés dans un fichier.

Tous ces blocs de JPEG2000 fonctionnent séquentiellement. Il y a des tâches qui peuvent être parallélisables telles que DWT/IDWT, EBCOT. Afin de garantir le fonctionnement de cette application, chaque tâche a un temps d'exécution propre dans une période de l'application. On s'intéresse à l'implantation des tâches parallélisables pour étudier l'effet de la modification de la granularité de parallélisme et la fréquence de fonctionnement. Cette modification correspond aux changements de la contrainte de temps qui sont produits par la variation du débit et de la quantité de données à traiter. Le but est de vérifier notre méthode d'adaptation proposée dans le chapitre précédent.

3.3.1 Description de l'algorithme DWT

La transformée en ondelettes discrètes bidimensionnelles (2D DWT) revient à l'analyse multi-résolution d'une image. C'est une application unidimensionnelle sur la direction horizontale et verticale [AIWB98]. Celle-ci est décomposée en un ensemble de sous bandes comme illustré par la figure 3.6, représentant l'information portée par l'image source à différents niveaux de résolution : l'image d'approximation (LL) est une version réduite et lissée de l'image initiale. Tandis que les images de détails "horizontaux" (LH), "verticaux" (HL), "diagonaux" (HH) contiennent uniquement des informations relatives à la texture locale et aux contours des régions de l'image, à une résolution et une direction donnée. Selon la dimension du signal, la DWT/IDWT peut être unidimensionnelle, bidimensionnelle, ou tridimensionnelle. Une étude de l'approche d'algorithme classique est détaillée ci-dessus.

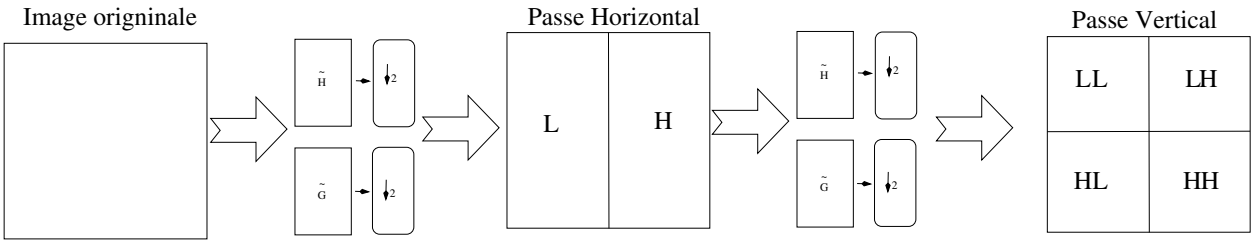

FIGURE 3.6 – Principe de la transformée en ondelettes discrètes

3.3.1.1 Étude de l'algorithme classique 2-D

La transformation en ondelettes $2D$ séparable permet d'utiliser la même architecture de banc de filtres pour chaque passe de décomposition horizontal ou verticale. Au cours du parcours d'une image, l'algorithme en pyramide récursif enchaîne pour chaque pixel des passes de filtrage horizontales et des passes de filtrage verticales. Les performances de cette approche sont fortement limitées par la gestion des données temporaires requises entre deux couches successives et entre le filtre horizontal et vertical. La figure 3.7 présente le codeur en 1D sur deux niveaux et la figure 3.8 donne un exemple de transformée inverse en 1D sur deux niveaux.

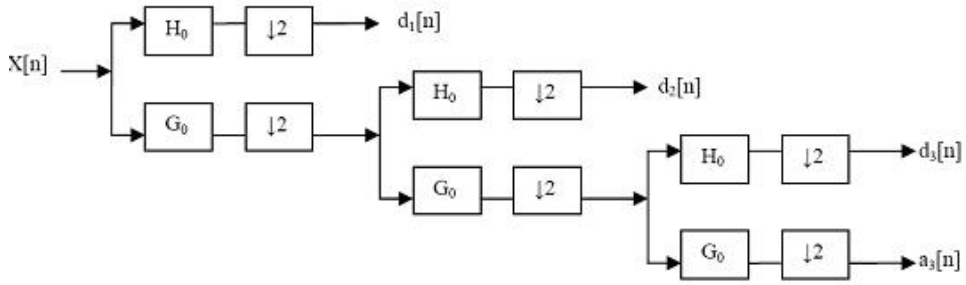

FIGURE 3.7 – Transformée en 1D sur deux niveaux d'une image

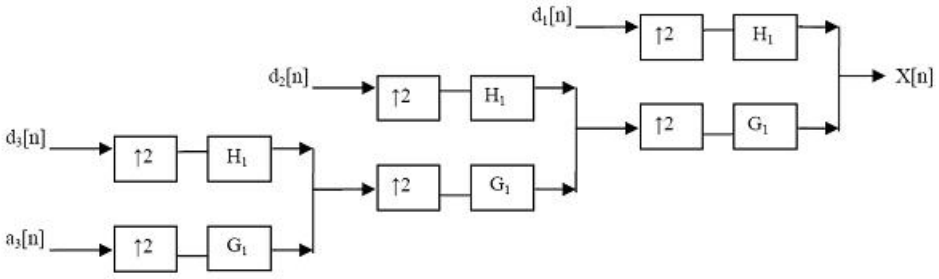

FIGURE 3.8 – Transforme inverse en 1D sur deux niveaux d'une image

Pour une image 2D à N lignes et N colonnes et L niveaux, la quantité de données à filtrer sur chaque couche augmente (pour IDWT) par un facteur de quatre d'une couche à l'autre, et la quantité totale des données traitées tout au long du processus de reconstruction de l'arbre est donnée par l'équation suivante :

$$D = \sum_{i=1}^{L} \frac{N \times N}{4^{i-1}} = \frac{4^L - 1}{3 \times 4^{L-1}} \times N \times N \tag{3.3}$$

Pour traiter un image $N \times N$, la taille de la mémoire temporaire nécessaire est :

$$D - N \times N = (\frac{4^L - 1}{3 \times 4^{L-1}}) \times N \times N \tag{3.4}$$

.

A titre d'exemple, pour $deux$ niveaux de résolution une mémoire temporaire de 0.25 $N \times N$ est nécessaire. Pour une couche, le processus de filtrage est réalisé horizontalement et verticalement. Donc, deux accès en lecture et deux accès en écriture sont nécessaires et la quantité totale de données lue et écrite est exprimé par $D_w = D_r = 2 \times D$. La bande passante de mémoire B, en cas d'accès bidirectionnel, peut être considérée comme le produite entre le nombre total de données traitées pour une image par seconde (fps) $T_{df} = (D_r + D_w) \times fps$ et le nombre de bits N_b d'un coefficient :

$$B = T_{df} \times N_b \tag{3.5}$$

A titre d'exemple, pour une image en niveaux de gris de 512×512 pixels avec 25 trames par seconde, 8 bits par pixel et 2 niveaux de reconstruction, une bande passante de 260 Mb/s est nécessaire. Ces résultats illustrent le problème de la gestion de la mémoire comme le principal goulet d'étranglement de l'approche classique.

3.3.1.2 Exemple de DWT sans perte (5/3)

Le filtre de la transformé directe DWT (5/3) est donné par les équations suivantes :

$$D[n] = S_0[n] - [(D[n] + D[n-1] + 2/4] \tag{3.6}$$

$$S[n] = D_0[n] + [1/2(S_0(n+1) + S_0[n])] \tag{3.7}$$

Le filtre de la transformé inverse DWT (5/3) est donné par les équations suivantes :

$$D[n] = D_0[n] - [1/2(S_0(n+1) + S_0[n])]] \tag{3.8}$$

$$S[n] = s_0[n] + [1/4(D[n] + D[n-1]) + 1/2] \tag{3.9}$$

$D[n]$ est le terme pair et $S[n]$ est le terme impair. Le graphe flot de données (DFG) est représenté par la figure 3.9. Il est composé de deux partitions : paire et impaire.

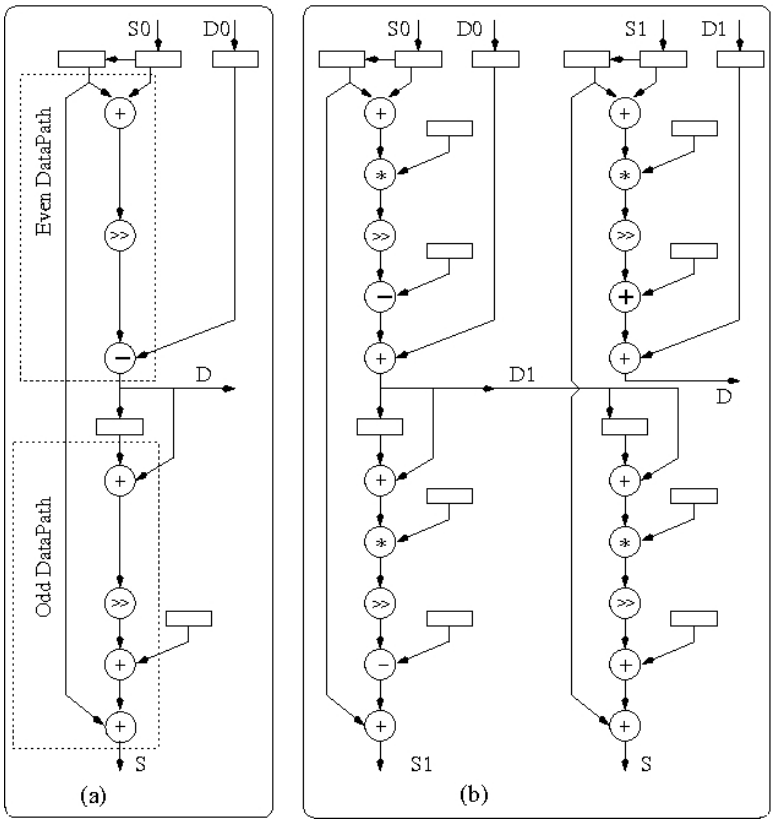

FIGURE 3.9 – IDWT 5/3 (a) et 9/7 DFG (b)

3.3.1.3 Exemple de DWT avec perte (9/7)

Le filtre 9/7 est calculé par l'équation suivant :

$$D_1[n] = D_0[n] + [\frac{203}{128}(-S_0[n+1] - S_0[n]) + 0.5] \quad (3.10)$$

$$S_1[n] = S_0[n] + [\frac{217}{4096}(-D_1[n] - D_1[n-1]) + 0.5] \quad (3.11)$$

$$D[n] = D_1[n] + [\frac{113}{128}(D_1[n+1] + D_1[n]) + 0.5] \quad (3.12)$$

$$S[n] = S_1[n] + [\frac{1817}{4096}(D_1[n] + D_1[n-1]) + 0.5] \quad (3.13)$$

Il existe des similitudes entre les équations de filtre $5/3$ et de filtre $9/7$ sur le graphique flot de données. En considérant le tableau 3.3, nous pouvons voir que l'échange entre différents filtres peut être mis en oeuvre par la reconfiguration partielle de partie différente.

TABLE 3.3 – Autres exemples de filtres de la transformée en ondelette

Filtrage	Additions	Shifts	Multiplications
5/3	5	2	0
9/7-M	8	2	1

3.4 Description de l'architecture

Pour vérifier le compromis efficacité énergétique et performance dans le cas de l'adaptation aux variations de la quantité de données et dans le cas de la DWT inverse, nous avons utilisé l'architecture illustrée par la figure 3.10. Cette architecture est composée de quatre éléments de calcul reconfigurables, l'interface de communication, et un contrôleur mémoire. La mémoire est organisée de manière hiérarchique afin d'assurer une bande passante suffisante entre les ressources de stockage et de calcul. L'interface de communication permet d'assurer les différents échanges de données entre la mémoire globale et le PE. Par ailleurs, cette interface permet la communication entre les mémoires interne et externe.

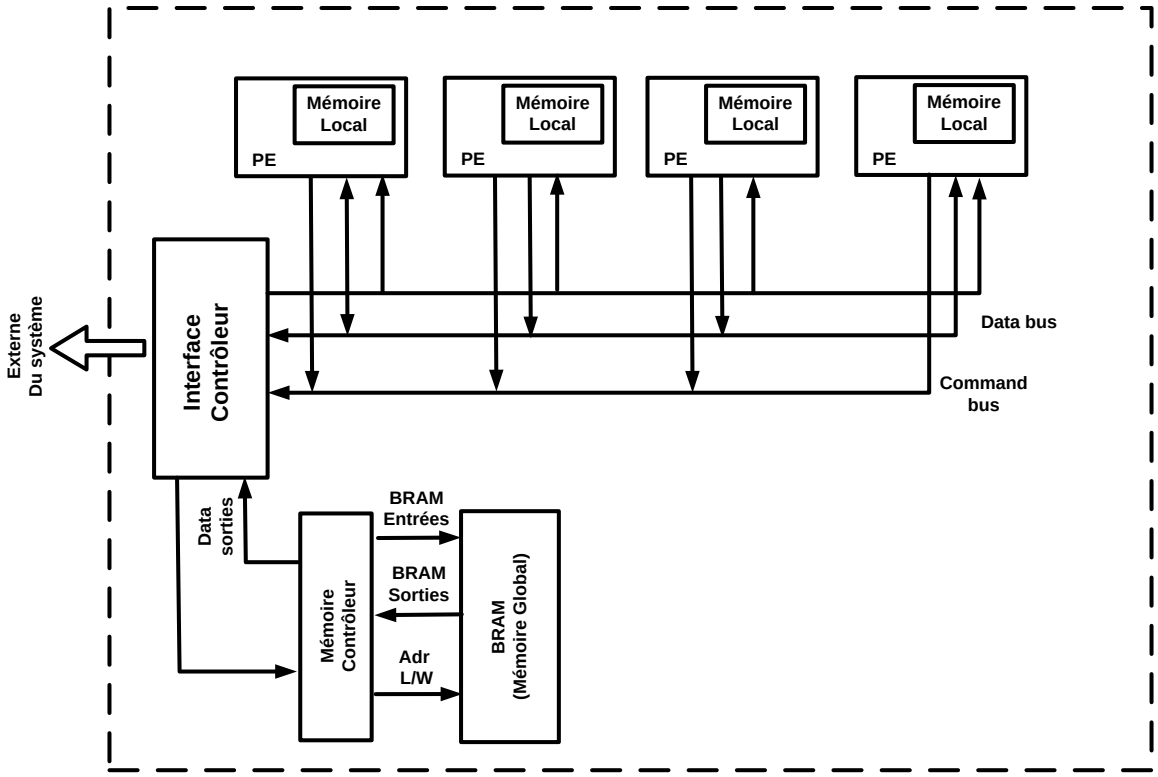

FIGURE 3.10 – Vue globale de l'architecture

3.4.1 Module de calcul

Le PE est le module de calcul principal. Il permet d'exécuter les différentes tâches matérielles telles que la IDWT en reconfiguration dynamique. Le nombre de PE pour une tâche peut être ajusté dynamiquement par la reconfiguration partielle (Résultat de placement et routage voit dans l'annexe B). La figure 3.11 illustre un exemple d'architecture d'un PE. Chaque PE contient une mémoire locale de taille 64 octets.

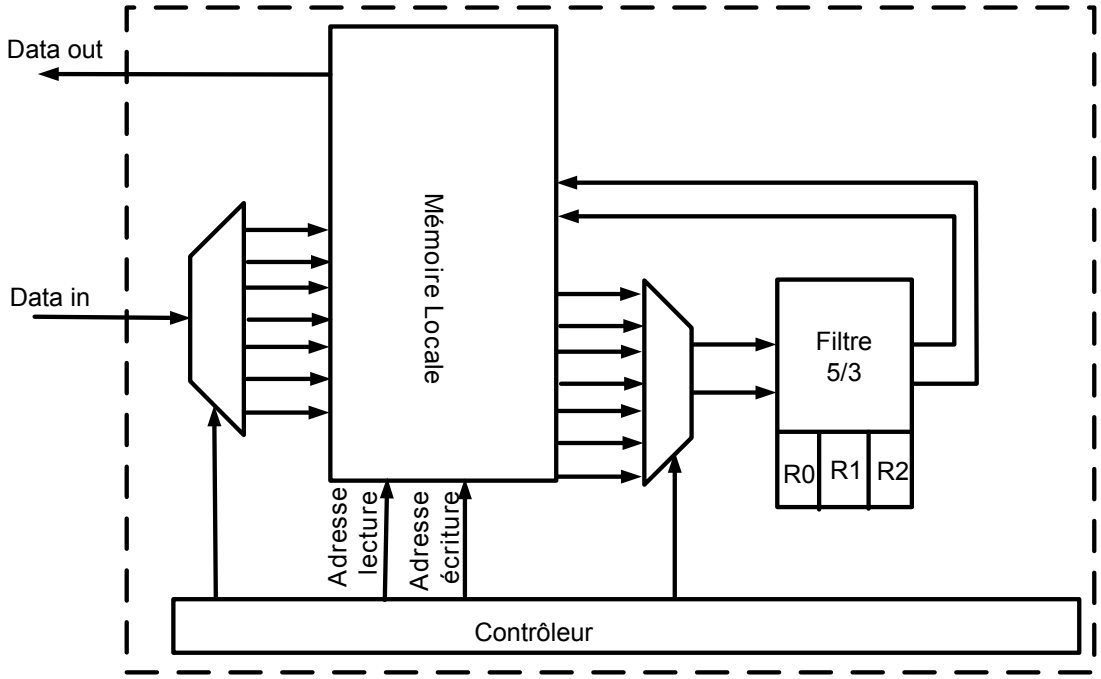

FIGURE 3.11 – Architecture d'un PE

L'élément de calcul de IDWT est conçu pour traiter un bloc de données de taille d'une mémoire locale. La période de travail d'un PE est composée de trois étapes séquentielles (illustrée dans la figure 3.14-(2)) : lecture de données à partir de la mémoire globale, exécution, écriture des résultats dans la mémoire globale. Les détails de l'exécution d'une tâche sont les suivants :

1. Lecture(R) : Les données à traiter sont d'abord envoyées à la mémoire locale à partir de la mémoire globale. La gestion d'accès à la mémoire locale est effectuée par le module de contrôle qui est intégré à l'interface de communication. Cela donne l'ordre de lecture et d'écriture pour l'accès de la ligne ou la colonne des

données à traiter. Les mémoires internes étant indépendantes, deux données de calcul peuvent être lues dans un cycle d'horloge.

2. Exécution(E) : Dans cette phase, le filtre 5/3 DWT inverse est installé dans le PE (illustrée dans l'équation 3.6). Chaque période de calcul du filtre est composée de trois étapes exécutées en pipeline :

   Lecture des données à traiter pour effectuer le calcul du filtre IDWT5/3, on doit accéder à deux coefficients. Cet accès est effectué en un cycle d'horloge grâce à une organisation spécifique de la mémoire locale.

   Calcul des filtres dans cette phase, les données disponibles dans le fichier de registres sont utilisées par le chemin de données du PE pour traiter en parallèle les deux parties du filtre.

   Écriture des résultats les résultats de traitement du filtre passe-bas et passe-haut sont écrits dans la mémoire locale. Les deux écritures sont exécutées en parallèle dans une période d'horloge.

   Le contrôleur du PE s'occupe de la gestion des différents accès mémoire par la génération adéquate des adresses de lecture et écriture et le contrôle du flux de donnée.

3. Écriture(W) : Après la fin du traitement d'un bloc de donnée,les résultats sont transférés vers la mémoire globale.

3.4.2 Hiérarchie mémoire

Les ressources de mémorisation sont distribuées sur deux niveaux de hiérarchie. Le premier est constitué d'une mémoire globale unique de 64 mots de 8 bits. Cette mémoire peut être assimilée à la mémoire cache d'un microprocesseur. Elle procure une solution intermédiaire, en terme de coût énergétique et de capacité de stockage,entre les mémoires locales et la mémoire globale du système à l'extérieur du cluster. L'ordre de ces accès et les déplacements de données entre cette mémoire sont assurés par le contrôleur mémoire décrit dans la section suivante.

Le second niveau de hiérarchie intègre 128 mots de 32 bits distribués équitablement entre les 4 RPMs. Cette mémoire, connectée à l'extérieur du cluster par un bus du système de 32 bits, stocke les données à traiter au sein des RPM. Elle est contrôlée par un générateur d'adresses local dans l'élément de communication.

3.4.3 Élement de communication

L'élément de communication de l'architecture permet de transmettre les données de la mémoire externe (DDR) vers la mémoire globale interne (BRAM), et de la mémoire globale vers la mémoire locale. La figure 3.12 donne les détails de cet élément. Il est composé d'un contrôleur de la mémoire globale qui s'occupe du transfert de données entre l'extérieur du circuit et l'intérieur et un séquenceur qui permet de réaliser les différents modes de communication entre la mémoire globale et les PEs. Le séquenceur permet en particulier d'adapter les niveaux de parallélisme (dans notre cas 1PE, 2PE ou 4PE) à la mémoire globale.

L'élément de communication accède à la mémoire globale par lecture ou écriture de mots de 32 bits. L'élément de communication effectue également le transfert sur 8 bits entre la mémoire globale et chaque PE en parallèle, en effet les données à traiter et les résultats de IDWT sont codés sur 8 bits. Le schéma de la distribution des données est illustré par la figure 3.13. Ici, nous supposons que les données à traiter sont organisées sous forme de blocs. Chaque bloc a une taille 64 octets.

3.4.4 Fonctionnement de l'architecture

3.4.4.1 Période de fonctionnement

L'architecture fonctionne en trois phases : lecture de données à traiter, calcul, et écriture des résultats. Ces trois phases constituent une période de fonctionnement. Dans cette période, une quantité de données correspondant à la taille de la mémoire globale est traitée. La figure 3.14 illustre ces trois phases.

Tout d'abord, une lecture des données à traiter qui se trouvent dans la mémoire externe est effectuée. Le temps de lecture est lié à la quantité de données et la vitesse

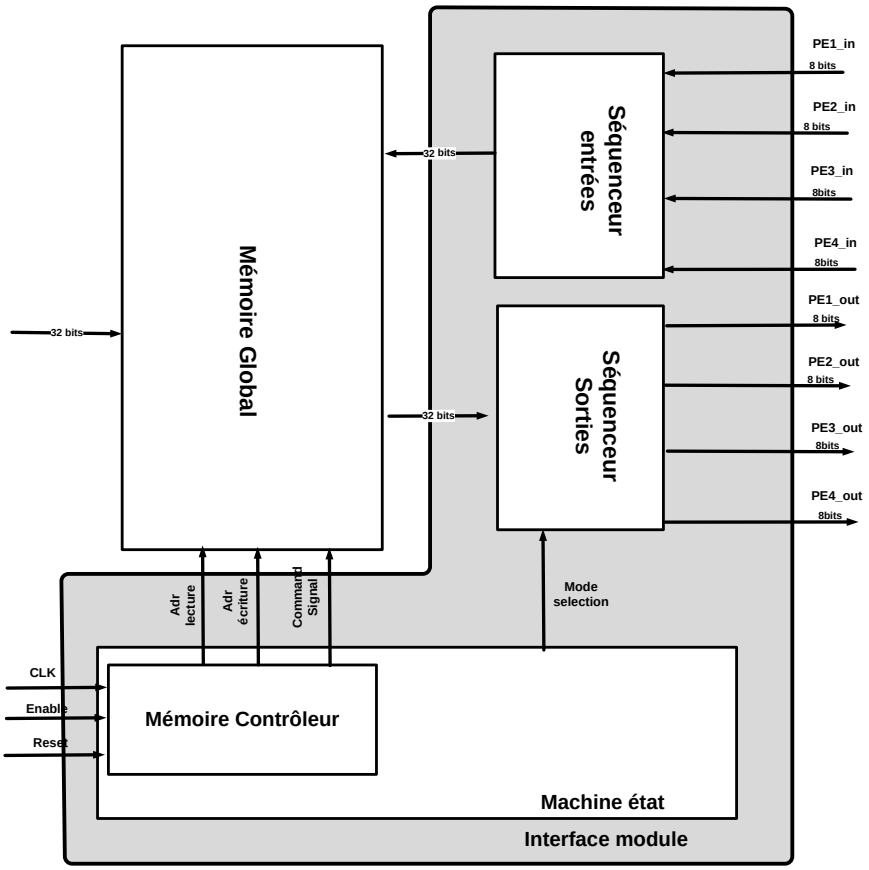

FIGURE 3.12 – Interface de communication

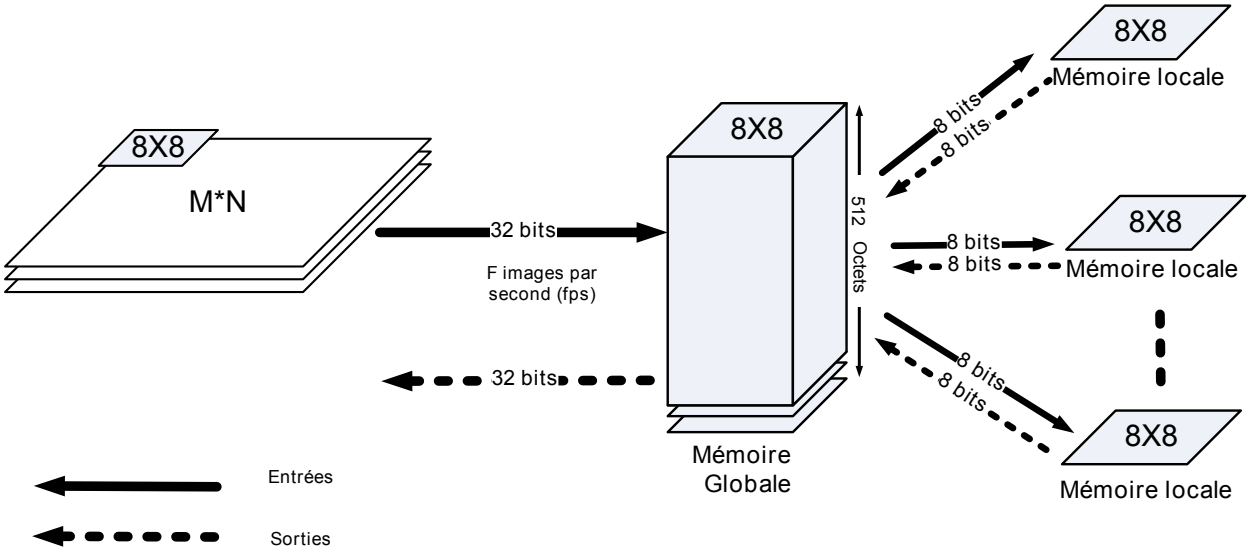

FIGURE 3.13 – Distribution des ressources dans le système

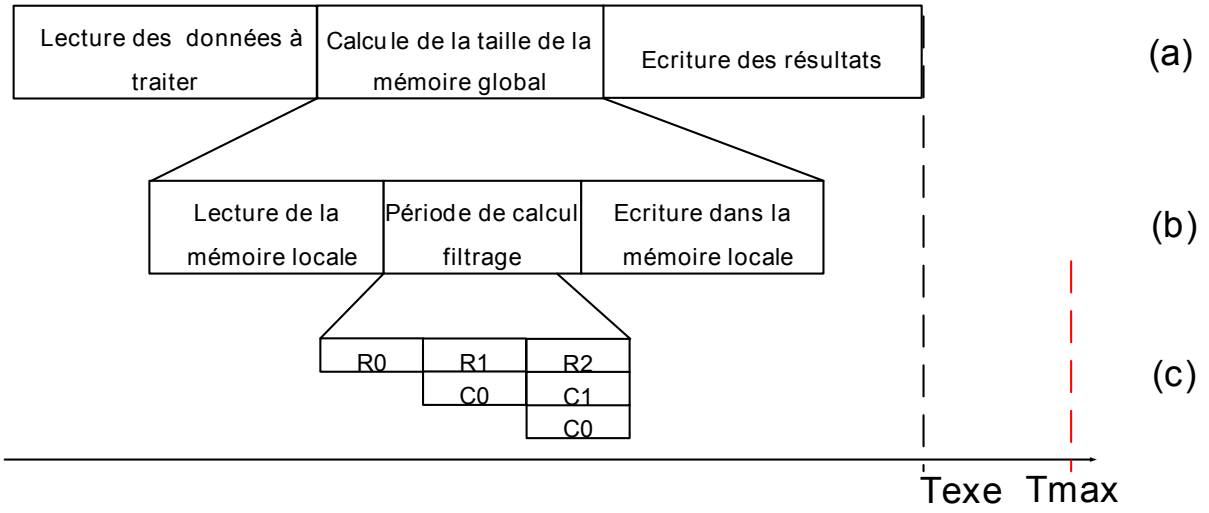

FIGURE 3.14 – Période de calcul du système

de leur transmission. Cette vitesse correspond au nombre de trame par second $F(fps)$.

La deuxième phase est la période de traitement pour un bloc mémoire globale (illustré dans la figure3.14-b). Elle est composée du temps de lecture des données à traiter de la mémoire globale, de la période de calcul de (des) PEs et de l'écriture des résultats dans la mémoire globale.

En fin, la dernière phase est l'écriture des résultats d'une taille de la mémoire globale dans la mémoire externe. Le temps d'écriture est identique au temps de lecture.

3.4.4.2 Temps d'exécution

Dans cette section, nous allons caractériser le temps d'exécution maximum et réel du système. Le temps d'exécution maximum est la contrainte de temps qui doit être respectée par le système pour garantir le bon fonctionnement. Le temps d'exécution réel est une période de fonctionnement du système qui dépend des paramètres de l'architecture. Ici, ces deux temps sont liés à la taille de la mémoire globale.

3.4.4.2.1 Temps d'exécution maximum Dans notre système, La taille d'une trame de données peut être différente de la taille de la mémoire globale. Le temps d'exécution

maximum est défini par l'équation suivant :

$$T_{max} = \frac{taille_{mmoireglobal}}{F * taille_{trame}}(s) \tag{3.14}$$

F est la vitesse de transmission (trames), $taille_{mmoireglobale}$ est fixé dans notre système. Donc, le temps d'exécution maximum est déterminé par la vitesse de transmission et la taille d'une trame.

Par exemple, Si on considère une séquence de données à traiter ayant une taille de trame $128x128$ *octets et qui arrive avec une vitesse de* 30 *trames par second (*$F = 30trame/s$*). La taille de la mémoire globale étant de* 512 *octets, le temps d'exécution maximum (contrainte de temps) est donc* $T_{max} = 1.04(ms)$

Le temps d'exécution réel Le temps d'exécution réel (T_{exe}) correspond à la période de traitement définie dans la figure 3.14. Celui-ci est composé de trois parties : le temps de lecture et écriture et le temps de calcul. Ce temps d'exécution réel dépend de la granularité de parallélisme et de la fréquence de fonctionnement.

3.5 Résultats expérimentaux

3.5.1 Distribution des ressources logiques

L'architecture illustrée dans la figure 3.2 est réalisée en ciblant le circuit FPGA Virtex-4SX35 de Xilinx. Le nombre de zones reconfigurables partiellement est défini en fonction du nombre maximum de PEs de l'application. Dans notre expérimentation, le nombre maximum de PE représentant une fonction de IDWT étant quatre, nous avons défini autant de zones reconfigurables. Le placement de chaque composant de l'architecture doit d'abord être effectué. La taille de chaque composant de l'architecture est obtenue après les résultats de synthèse (le table 3.5.1). Cependant, le choix de la taille des zones reconfigurables est défini en fonction du nombre de CLBs occupé par le PE et du nombre de trames. Les tailles et les données de configuration des quatre zones sont identiques. Les données de configurations sont stockées dans la mémoire

externe du système. La mémoire externe DDR est choisie pour stocker les données de configuration à cause de la grande quantité de données de configuration. La fréquence maximum de l'horloge du système obtenue par l'outil synthèse de Xilinx est $104MHz$.

TABLE 3.4 – L'implantation du système avec différents nombres de PE.

Composant de l'architecture	Taille(CLB)	Taille de données de configuration
$Total$	12644(sur 15360)	1673 Koctets
PE	1771(sur 15360)	206 koctets
élément de communciation	3856(sur 15360)	inclut dans le Total
BusMacro	$6CLB$	n/a

La partie *Total* correspond à la partie statique (le microblaze, les entrées/sorties et le bus du système,etc). La configuration du FPGA et le stockage de données de configuration partielle dans la mémoire sur puce sont faits par JTAG.

3.5.2 Placement de PE et de l'arbre de l'horloge

Le placement des PEs peut être effectué automatiquement ou manuellement par l'outil de Xilinx. Pour qu'un PE puisse fonctionner avec sa propre horloge, chaque zone reconfigurable est placée dans une zone d'horloge. 60% de la consommation dans le FPGA étant due aux interconnexions , le placement manuel des PEs dans une zone reconfigurable permet d'optimiser le routage de l'arbre de l'horloge et donc de réduire la consommation. De plus, l'interconnexion du réseau d'horloge occupe 40% [ANC03].

La figure 3.15 illustre la consommation de puissance dynamique utilisant trois différentes méthodes : le placement automatique, le placement manuellement dans une zone de l'horloge, et le placement manuellement avec la reconfiguration partielle.

Dans le premier cas, tous les quatre PEs sont pré-installés dans le circuit sans contrainte de placement. Leurs fonctions peuvent être activés ou désactivés par un simple signal

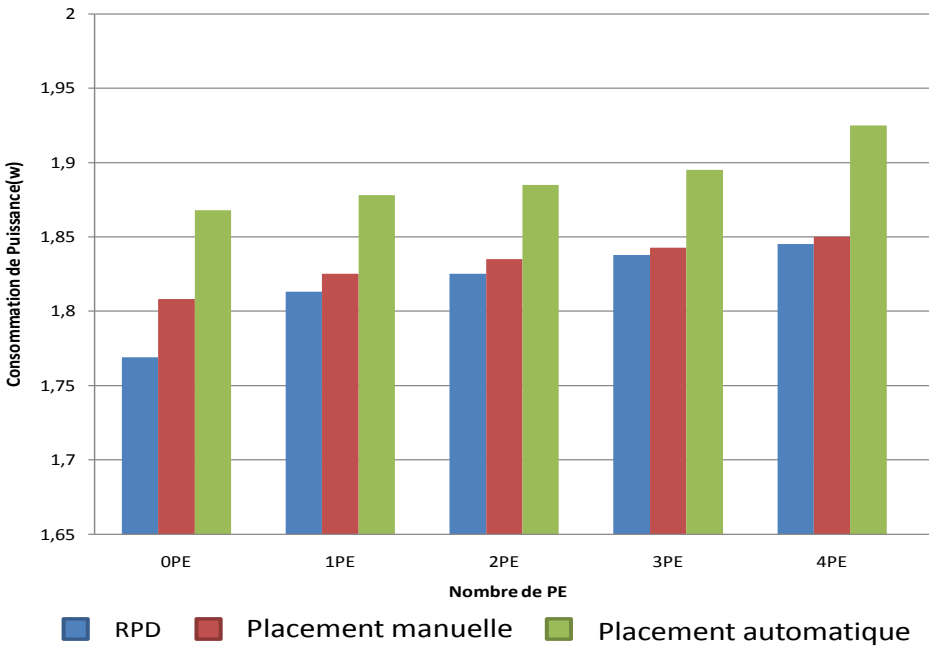

FIGURE 3.15 – Comparaison de la consommation de puissance en utilisant trois différentes méthodes de placement. Les placements de RPD et Placement manuellement sont mêmes. RPD active le nombre de PE par reconfiguration partielle. Placement manuel active le nombre de PE par une simple interrupteur comme le cas Placement automatique.

pour étudier la consommation dans le cas de fonctionnement de 1,2,3 ou 4 PEs. Dans le deuxième cas, Les PEs sont pré-installés dans le circuit avec une contrainte de placement dans une zone isochrone. Dans le troisième cas, quatre zones reconfigurable sont définie et les PEs sont placés par la reconfiguration partielle.

Dans cette figure, 0 PE dans le premier et le second cas, signifie tous les PEs sont désactivées par un signal. Alors que dans le troisième cas, cela signifie que seule la partie statique est configurée (les zones reconfigurables ne contiennent aucun PE). Nous remarquons que la consommation dans le premier cas est plus grande que les deux autres quelque soit le nombre de PEs actifs. La différence de consommation entre ces trois cas est essentiellement due à la manière de distribuer l'horloge pour chaque PE et la contrainte de placement. On constate en particulier que la suppression de modules est plus efficace que leur désactivions.

3.5.3 Variation du temps d'exécution réel et maximum

3.5.3.1 Temps d'exécution réel

Les temps d'exécution réels du traitement $2 - D$ IDWT en fonction du nombre de PEs et de la fréquence de fonctionnement sont donnés dans le tableau 3.5.3.1. Ces temps d'exécution sont obtenus par simulation après routage pour une quantité de données égale à la taille de la mémoire globale (512 octets). La flexibilité de notre architecture autorise ainsi le traitement d'un bloc de 512 pixels en moins de $103,1us$. Dans le cas le plus favorable, le temps de traitement d'un bloc est réduit à $7,5us$. La comparaison de la contrainte de temps d'exécution avec les valeurs de la table permet d'identifier les temps d'exécution raisonnables. A ce temps correspond une architecture d'efficacité énergétique optimale.

3.5.3.2 Contrainte de temps

La contrainte de temps est composée de deux parties : la contrainte de temps d'exécution et la contrainte de temps de configuration (RPD et DFS). La première dépend du débit et de la taille des données à traiter. Le tableau 3.5.3.2 donne un exemple de va-

TABLE 3.5 – Temps d'exécution réel (taille de 512 octets) en fonction des paramètres d'architecture(us)

	$25MHz$	$30MHz$	$35MHz$	$40MHz$	$50MHz$	$60MHz$	$70MHz$	$80MHz$
$1PE$	$103,1$	$85,9$	$73,5$	65	$51,5$	$43,7$	$38,7$	$32,8$
$2PE$	$50,7$	42	$34,9$	$31,3$	$25,8$	$21,2$	$18,5$	$16,1$
$4PE$	24	$19,9$	17	$14,8$	$11,9$	$9,8$	$8,5$	$7,5$

riation de cette contrainte de temps pour trois différentes tailles de données à traiter et pour deux débits ($30trame/s$ et $60trame/s$). Les temps d'exécution maximum du système sont calculés et sont présentés dans le tableau 3.5.3.2 en utilisant l'équation 3.14.

TABLE 3.6 – Contrainte de temps pour un bloc mémoire globale (512 octets) en fonction du débit et de la taille de données à traiter

Débit (trame/s)	taille de données à traiter (M*N)	Contrainte de temps $T_{max}(us)$
30	512*512	65
	800*600	$35,55$
	1024*768	$21,70$
60	512*512	$32,55$
	800*600	$17,78$
	1024*768	$10,85$

La comparaison de T_{max} avec le temps d'exécution réel représenté sur le tableau 3.5.3.1, montre que plusieurs solutions sont possibles permettant de respecter la contrainte de temps. Ces solutions sont représentées dans le tableau 3.5.3.2 sous forme du couple (nombre de PEs, fréquence) défini dans le chapitre 2. Le choix d'une solution va dépendre de son efficacité énergétique, pour cela une étude de consommation est nécessaire.

3.5.3.2.1 Consommation de puissance La consommation de puissance dynamique en fonction du nombre de PEs et de la fréquence est présentée dans la figure 3.16.

TABLE 3.7 – Les architectures valables en fonction du débit et de la taille de données à traiter

Débit (trame/s)	taille d'une trame(M*N)	Les possibilité des architectures
30	512*512	$(1PE, 40\ 80MHz), (2PE, 25\ 80MHz), (4PE, 25\ 80MHz)$
	800*600	$(1PE, 80MHz), (2PE, 35\ 80MHz), (4PE, 25\ 80MHz)$
	1024*768	$(2PE, 60\ 80MHz), (4PE, 25\ 80MHz)$
60	512*512	$(2PE, 40\ 80MHz), (4PE, 25\ 80MHz)$
	800*600	$(2PE, 80MHz), (4PE, 35\ 80MHz)$
	1024*768	$(4PE, 60\ 80MHz)$

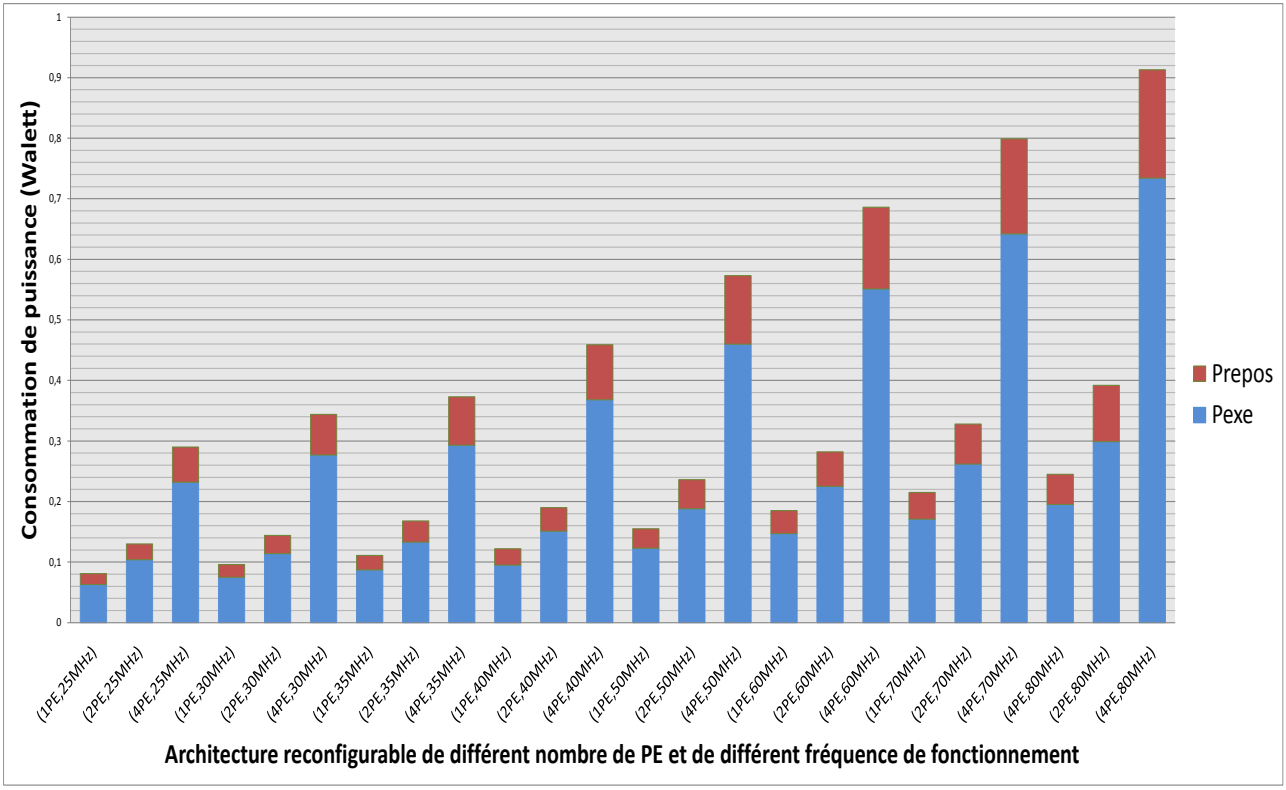

FIGURE 3.16 – Consommation de puissance en fonction du nombre de PE et des différentes fréquences de fonctionnement

Dans cette figure, la consommation de puissance est composée de deux parties : la puissance P_{exe} et P_{repos}. Le P_{exe} est la puissance consommée pendant le temps de fonctionnement du système. P_{repos} est la puissance consommée pendant le temps de repos. Ces résultats sont obtenus par l'outil d'estimation de puissance de Xilinx. On constate que la puissance dynamique pendant le fonctionnement est proportionnelle à la fréquence de fonctionnement et au nombre de PEs. La puissance consommée pendant le temps de repos (due essentiellement à la consommation de l'horloge) représente en moyen 27% de la puissance consommée pendant le fonctionnement. La durée de repos va avoir un impact direct sur la consommation d'énergie hors fonctionnement ; et donc sur l'efficacité énergétique.

3.5.3.3 Définition de l'efficacité énergétique

La vérification du choix de l'architecture optimale passe par l'efficacité énergétique. Cette métrique utilisée dans pour évaluer cette efficacité énergétique s'exprime en $MOPS/W$. Elle est définie par l'équation 3.15. Dans cette équation, N_{opt} représente le nombre d'opération par période ($T_e + T_r$). Le produit $(T_{exe} * P_{exe} + T_{repot} * P_{repot})$ représente la consommation d'énergie à l'exécution de N_{opt} opérations dans la même période.

$$e_E = \frac{N_{opt}}{(T_{exe} * P_{exe} + T_{repot} * P_{repot})} \quad (MOPS/W) \tag{3.15}$$

Le nombre des opérations N_{opt} pour le traitement 2D-DWT inverse à n'importe quel niveau de calcul est donné par l'équation [LL06] :

$$N_{opt} = \frac{8 * F * M * N}{3} \tag{3.16}$$

F est le nombre de trames de données à traiter transmit par second. Chaque trame a une taille $M * N$ sur 8 bits.

*Par exemple, pour une série de trames avec une vitesse 10 trame/s la taille d'image est 8*8, le nombre total d'opérations pour cette série d'images est :* $N_{opt} = (8 * 10 * 8 * 8)/3 = 1701$.

3.5.3.4 Calcul de l'efficacité énergétique

le tableau 3.8 illustre l'efficacité énergétique pour les trois exemples de contrainte de temps ($35,54us$, $21,69us$ et $17,7us$). Les efficacités énergétiques valables sont obtenues par l'équation 3.15. L'architecture optimale avec critère d'efficacité existe sous la condition de contrainte de temps. Pour ces trois contraintes de temps,les architectures optimales sont $(2PE, 35MHz)$, $(2PE, 60MHz)$ et $(4PE, 35MHz)$.

TABLE 3.8 – Efficacité énergétique($\frac{MOPS}{W}$) pour différents temps d'exécution maximum

35,5us	25MHz	30MHz	35MHz	40Mhz	50MHz	60MHz	70MHz	80MHz
1 PE	/	/	/	/	/	/	/	2234,79
2 PE	/	/	**2445,28**	2417,55	2367,27	2336,55	2294,09	2261,97
4 PE	2265,64	2231,98	2190,26	2155,27	2080,27	2017,178	1950,07	1889,75
21,7us	25MHz	30MHz	35MHz	40Mhz	50MHz	60MHz	70MHz	80MHz
1 PE	/	/	/	/	/	/	/	/
2 PE	/	/	/	/	/	**2522,53**	2476,80	2446,79
4 PE	2413,37	2381,25	2344,41	2312,73	2238,73	2180,27	2112,37	2052,53
17,7us	25MHz	30MHz	35MHz	40Mhz	50MHz	60MHz	70MHz	80MHz
1 PE	/	/	/	/	/	/	/	/
2 PE	/	/	/	/	/	/	/	1436,31
4 PE	/	/	**1444,78**	1387,11	1339,45	1299,72	1256,91	1218,54

3.5.3.5 Description du processus d'adaptation

La méthode d'adaptation pour identifier et réaliser l'architecture optimale est implémentée dans un microprocesseur. Le choix d'un contrôleur d'adaptation logiciel offre une mise à jour facile par rapport une version matériel dans la phase de développement. En outres, il permet de piloter facilement le périphérique HWICAP de Xilinx et le contrôleur de DFS.

L'identification de l'état optimal est effectué par la comparaison de la valeur de contraint de temps et les valeurs dans le base de données de temps d'exécution. Afin de trouver l'état optimal, une séries de relations de transitions $X_{E_a->E_o}(x, y)$ (voir l'équation **??**) sont utilisées. Chaque relation de transitions $X_{E_p->E_o}(x, y)$ est définie par deux conditions x et y : le **respect de la contrainte de temps** d'exécution et le **maximum d'efficacité énergétique**.

Le processus d'adaptation est activé par le changement de contrainte de temps d'exécution. Une fois la contrainte de temps est modifiée, les états potentiels qui respectent la nouvelle contrainte de temps sont identifiés. Une exemple d'identification des états potentiels avec une contrainte de temps x=T_D : $35,5us$ est illustré ci-dessous :

$$X_{E_a->E_o}(T_D) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

où, afin de respecter une contrainte de temps ($35,5us$), les états potentiels de la tâche IDWT de l'application JPEG2000 sont indiqués par les relations de transition $X_{E_p->E_o}(35,5\ us)$ avec les combinaisons de deux paramètres (la fréquence de fonctionnement et le niveau de parallélisme). En cherchant dans la base de données, les $X_{E_a->E_o}(35,5\ us)$ (définir dans l'équation **??**) du système sont égales à 1 quand leurs temps d'exécution respectent la contrainte de temps $35,5us$. Ensuite, parmi les états potentiels, la relation de transition optimale (définir dans l'équation **??**) est indiqué par la deuxième condition d'adaptation : le maximum de l'efficacité énergétique. Un exemple pour identifier l'état optimal est montrée dans la matrice ci-dessous :

$$X_{E_p->E_o}(T_D, Max(T)) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

En se basant sur la première condition (Respecter la contrainte de temps : $35,5us$), l'état optimal qui est choisi parmi ces états potentiels (indiqués par les valeurs 1 dans la matrice 3.5.3.5), associe un temps d'exécution maximal : $Max(T)$ par rapport aux temps d'exécution des autres états potentiels. Dans cette exemple, l'état optimal est indiqué par 1. Une fois que l'état optimal (associé à l'architecture $(2PE, 35MHz)$) est trouvé, la processus de configuration est lancé.

Architecture optimale maximisant l'efficacité énergétique Pour une contrainte de temps donnée, une seule architecture optimale existe pour obtenir le maximum d'efficacité énergétique. Par exemple, pour une contrainte de temps $35,5us$ (c.f 3.2), l'architecture garantissant la meilleure efficacité énergétique doit être constituée de deux PE et cadencé à une fréquence de fonctionnement de $35MHz$. Pour le même exemple, on peut voir qu'une architecture constituée de $1PE$ et une fréquence de fonctionnement inférieur à $80MHz$ ne peut pas satisfaire la contrainte de temps. Évidement, l'architecture qui contient le plus de PE($4PE$) et travaillant à une fréquence de fonctionnement maximale $80MHz$ offre la meilleure performance. Cependant, son efficacité énergétique n'est que de 39% de celle de l'architecture optimale ($2PE$,$35MHz$).

Adaptation de l'architecture optimale La contrainte de temps peut varier dynamiquement en fonction de la variation du débit de données à traiter. Quand la contrainte de temps varie, la méthode d'adaptation détaillée dans le chapitre précédent est appliquée dans notre expérimentation. Elle consiste, dans un premier temps, à choisir tous les temps d'exécution réels inférieurs à la contrainte de temps dans le tableau3.5.3.1, puis à choisir parmi ces temps celui qui est le plus proche de la contrainte de temps. Ce temps correspond à une architecture optimale en terme de nombre de PE et de fréquence de fonctionnement (voir le tableau3.5.3.1) qui maximise l'efficacité énergétique.

On peut distinguer deux cas d'activité d'adaptation : le cas anticipé et le cas non anticipé.

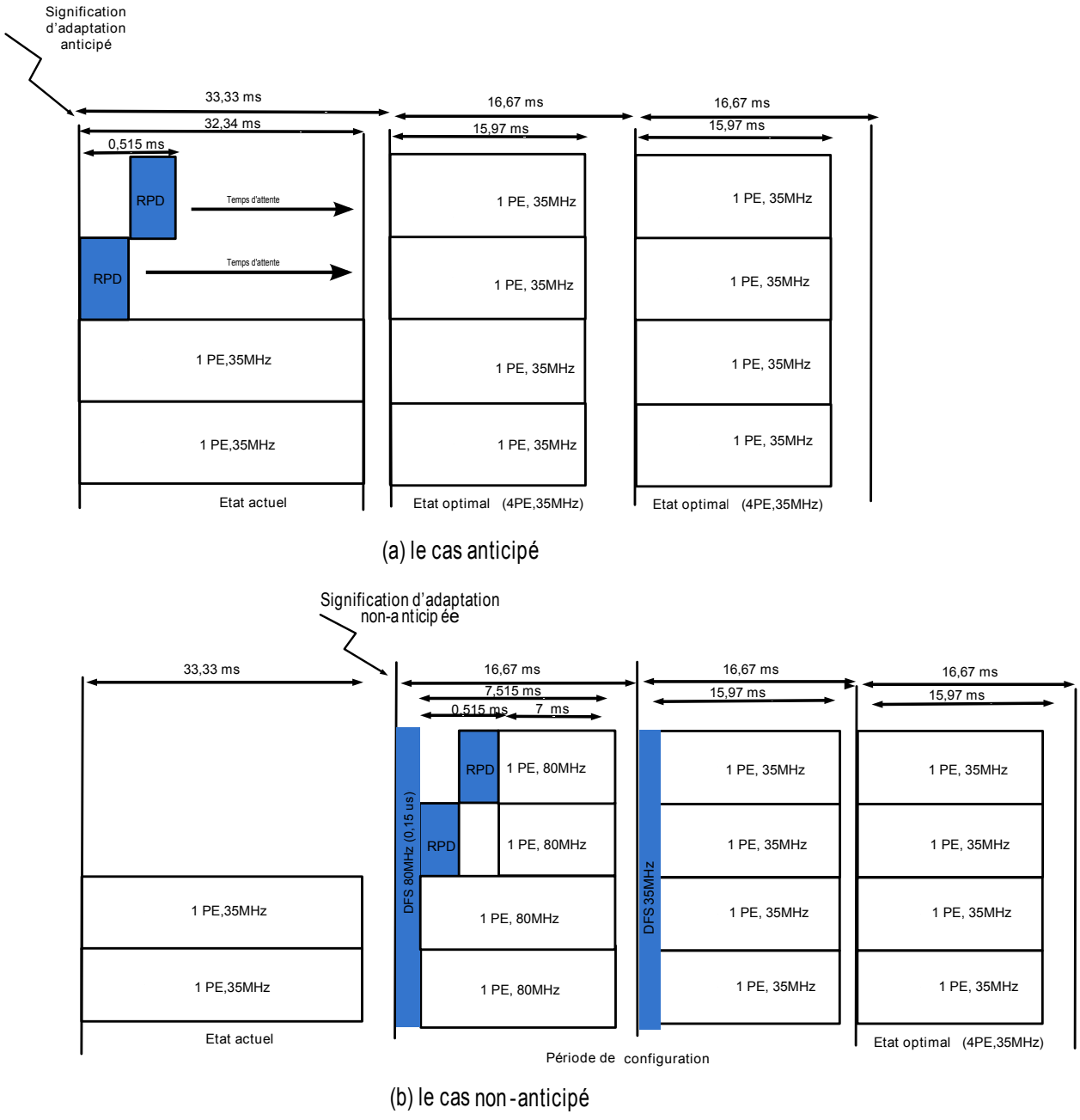

FIGURE 3.17 – Étapes de configuration anticipé(a) et non-anticipé(b)

Cas où le changement est anticipé : La méthode d'adaptation présentée dans le chapitre précédent est utilisée pour trouver une architecture optimale en fonction de

la nouvelle contrainte de temps (l'algorithme 2.1). Dans le cas où l'adaptation est anticipée, un contrôleur à l'entrée du système peut surveiller le changement du débit et prévoir une modification dans la période future. Ceci est illustré dans la figure 3.17-a. Dans ce cas, l'adaptation peut être correctement terminée avant le changement de débit. Le temps d'exécution réel est de $32,34ms$ pour une trame de données à traiter. Le temps de reconfiguration des deux PEs est au total de $0,515ms$. Les PEs configurés dans cette période ne fonctionnent pas immédiatement. L'architecture optimale commence à fonctionner à partir de la prochaine période. Le temps d'exécution de la nouvelle architecture est de $15,97ms$. Ce temps, de la même architecture (4PE,35MHz), est la somme des temps d'exécution de mémoire globale ($17us$ pour une mémoire 3.5.3.1). La contrainte de temps pour traiter une trame est de $16,67ms$. L'efficacité d'énergétique de cette architecture est $1444,78(MOPS/w)$.

Cas où le changement est non-anticipé Notre méthode d'adaptation proposée supporte le cas où le changement n'est pas anticipé. Dans ce cas, la reconfiguration du système n'est pas prévue. Pour garantir le fonctionnement du système en fonction de la nouvelle période d'exécution, une configuration transitoire (période d'adaptation) est nécessairement ajoutée. Ceci est visible dans la figure 3.17-b.

Dans cette période d'adaptation, la stratégie de configuration est décidée par l'algorithme 2.2 dans le chapitre précédent. Dans cet exemple, le nombre de PE est le nombre maximal. Donc, la troisième règle de cette méthode est appliquée : "*si j est déjà le niveau de parallélisme maximum, on augmente la fréquence de fonctionnement au maximum*"

L'architecture la plus performante $(4PE, 80MHz)$ est configuré dans le système. Le temps de configuration est divisé en deux (le temps de configuration de la fréquence et le temps de configuration des modules parallèles), est d'environ de $0,515ms$. Le temps d'exécution réel dans cette période est de $7,515ms$. Les deux PEs configurées ici fonctionnent ensemble d'après la configuration du deuxième PE.

La reconfiguration de la fréquence de fonctionnement est effectuée après une période. L'architecture optimale est réalisée après une période d'adaptation. L'efficacité énergétique dans l'état optimale est augmenté 19% par rapport l'état le plus perfor-

mance $(4PE, 80MHz)$

Limitations du contrôleur d'adaptation Outre cette première limitation, relative aux temps de configuration intégré à la période de calcul, une seconde limitation est liée à la consommation d'énergie durant la période de configuration. En effet, le temps d'auto-configuration par le port reconfiguration ICAP contient deux partis de temps détaillées dans la partie 3.2.1. Ici, le temps de transmission de données de la mémoire externe au composant HWICAP n'est pas pris en compte. Ce temps supplémentaire est causé par l'utilisation de MicroBlaze qui ralentit le temps total de configuration. Ce problème peut être bien réglé en utilisant un contrôleur matériel. Par ailleurs, si le processus d'adaptation n'est pas répétitif, ce qui est le cas dans le domaine d'applications concerné, la consommation dans la phase de configuration représenterait une par négligeable dans toute la durée de fonctionnement du système.

3.6 Conclusion

Dans ce chapitre, nous avons présenté les résultats d'implémentation d'une des fonctions principales (IDWT) de l'application JPEG2000. Une analyse de la fonction est présentée d'abord. Ensuite, une description d'architecture est détaillée avec des résultats préliminaires tels que le temps de reconfiguration partielle et dynamique et le temps de gestion dynamique de fréquence. Ces contraintes de temps sont prévues dans notre processus d'adaptation. Enfin, la partie des résultats expérimentaux montre des possibilités d'optimisation d'efficacité énergétique avec différents combinaisons des paramètres d'architecture. Les composants d'architecture sont détaillés ici avec les temps d'exécution réels estimés. La méthode d'adaptation proposée dans le troisième chapitre est réalisée de manière logicielle. Nous avons distingué deux cas d'adaptation et montré comment les gérer.

Conclusion et Perspectives

Les travaux présentés dans ce rapport de thèse portent sur la définition d'une architecture répondant dynamiquement à des contraintes inhérentes aux applications multimédia. Ces contraintes de flexibilité, de performance et aussi de consommation d'énergie ont été illustrées dans cette étude par l'analyse des méthodes d'adaptation et l'état de l'art des architectures reconfigurables. Cette étude nous a conduits à nous intéresser aux architectures reconfigurables à grain fin, actuel (les FPGAs), et sur l'impact de la reconfiguration partiel et dynamique en terme d'efficacité énergétique.

Par l'analyse du domaine applicatif, en particulier à parallélisme de données, ainsi que l'exploration de l'état de l'art, nous avons abouti à la définition d'une architecture reconfigurable partielle et dynamique en combinant deux méthodes : la reconfigurable partielle et dynamique et la gestion dynamique de la fréquence. Nous avons ainsi découvert qu'il n'existait aucune définition formelle ni méthode sur cette manière d'optimisation de l'efficacité énergétique dans les architectures reconfigurables.

Afin de garantir l'efficacité énergétique du système, une méthode d'adaptation est présentée. Celle-ci est capable de déterminer une architecture optimale pour avoir le maximum d'efficacité énergétique tout en respectant les contraintes de temps.

On distingue les caractéristiques de notre proposition sur la flexibilité, la performance et le coût de reconfiguration. Pour répondre aux besoins en flexibilité, on utilise les méthodes d'adaptation proposées. Pour répondre aux besoins en performance, nous avons en particulier développé des modules de calcul capables de supporter des traitements en parallèle. Pour réduire le coût de configuration, un choix de stratégie de configuration est développé dans notre méthode.

Suite à la proposition de la méthode d'adaptation, une architecture adaptative ba-

sée sur cette méthode est définie. Cette architecture contient plusieurs modules reconfigurables partiellement(RPM). Le nombre de RPM est lié à des contraintes du système telles que la bande passante, le temps de reconfiguration, le temps d'exécution réel et la contrainte de temps. Afin d'assurer une bande passante suffisante entre les ressources de stockage et de calcul de RPM, une hiérarchique mémoire à deux niveaux a été définie. Le premier niveau correspond à la mémoire globale. Elle est accessible par tous les RPMs via une interface commune. Ensuite, le second niveau de la hiérarchie correspond aux mémoires locales intégrées au sein des RPMs. Les RPMs n'accèdent qu'aux données stockées dans ce dernier niveau de hiérarchie.

Dans ce travail de thèse, afin d'illustrer ces concepts par un cas concret, l'exploitation de techniques de gestion dynamique de la fréquence(DFS) et la reconfiguration partielle pour notre architecture ont été proposées et vérifiées dans une étude de cas d'implantation de l'algorithme IDWT. Ces études menées pour caractériser l'efficacité énergétique, suivant différentes fréquences d'exécution et différents nombres de RPM, ont permis d'optimiser la consommation d'énergie selon le débit de données à traiter. Le maximum de l'efficacité énergétique est toujours garanti par cette méthode.

Bien sûr, notre proposition peut être optimisée. Comme perspectives de ce travail, nous allons discuter les évolutions possibles ou souhaitables de notre architecture et de sa chaîne de développement.

Discussion et Perspective

Optimisations architecturals Dans ce mémoire de thèse, nous nous sommes principalement concentrés sur l'optimisation de l'architecture reconfigurable en se basant sur l'étude de la niveau de parallélism et la fréquence de fonctionnement. Les études menées pour caratériser les architectures en consommation, suivant différentes fréquence de fonctionnment, différentes niveaux de paramlélisme, ont permis de constater une réduction moyenne 48% de la puissance consommée lorsque l'architecture est resecte les deux conditions d'adaptation. La gestion dynamique de tension(DVS) est une des méthodes les plus efficaces. La mise en oeuvre des techniques DVS produis

cependnat une caratérisation plus instable de la mémoire BRAM. Celle-ci n'est pas utilisée dans notre méthode à cause de la limitation d'utilisation de circuit FPGA commerciaux. Actuellement, la modification de tensions d'alimentations d'une partie de FPGA est inaccessible.

L'auto-reconfiguration dans notre expérimentation est réalisée en utilisant les circuits FPGAs de Xilinx. Celle-ci est détaille avec une exemple contient un certain numbre de module de cacul limité. Le contrôleur de reconfiguration est un processeur MicroBlaze. La majeure partie de temps de configuration est liée à la transmission des données de configuration par ce processeur. Pour réduire cette contrainte de temps de reconfiguration, une perspective principale est le développement d'un contrôleur matériel connectant directement le port ICAP et la mémoire où on enregistre les données de configuration.

Évaluations de la méthode d'adaptation Dans ce mémoire, nous avons démontré la faisabilité d'une méthode d'adaptation pour un architecture reconfigurable partielle et présentée les principaux étapes permettant de détecter l'état optimal de l'architecture et de le réaliser en temps réel. La vérification de cette méthode a été détaillée dans le chapitre 3. Celle-ci est distinqué par une application contient un numbre limité de module de calcul. Le type des modules de calcul sont identiques. Une perspective importante à ce travail consiste donc à compléter notre méthode afin d'adapter aux architecture hybride qui contient des différentes IP matériels et/ou logicielle. Acutellement, il n'y a que la puissance consommée par la calcul est pris en compte. La puissance consommée pendant la processus d'adaptation n'est pas prise en compte dans notre méthode. Pour cela, l'évoluation de la méthode mesure de la puissance consommée sur circuit permet d'identifier la puissance consommée pendant la processus d'adaptation.

Annexe A

Flot de conception de la reconfiguration partielle

A.1 Description du flot de conception utilisé

Le flot complet de conception de la reconfiguration partielle retenu est illustré dans la figure A.1. Elle est basée sur le flux de l'implémentation du système de Xilinx. Les étapes du flot de conception depuis la description d'un circuit logique à la configuration d'un circuit FPGA sont détaillées ci-dessous.

A.1.1 Description et synthèse

La première étape est la définition de la description de la conception en HDL(VHDL est utilisé dans notre travail). L'application au parallélisme de données est présenté par une série de tâches et de sous-tâches de traitement. De plus, les tâches sont séparées en deux parties : une partie commune et une partie spécifique. Un exemple est illustré dans la figure A.2, avec deux applications A_1 et A_2 présentant des tâches communes (ligne continue) et en des tâches spécifiques (ligne pointillée). Le passage de l'application A_1 à l'application A_2 nécessite le remplacement les tâches spécifiques et de la communication entre les tâches chargées et les tâches communes.

La synthèse est supportée par l'outil ISE de Xilinx. La reconfiguration partielle de-

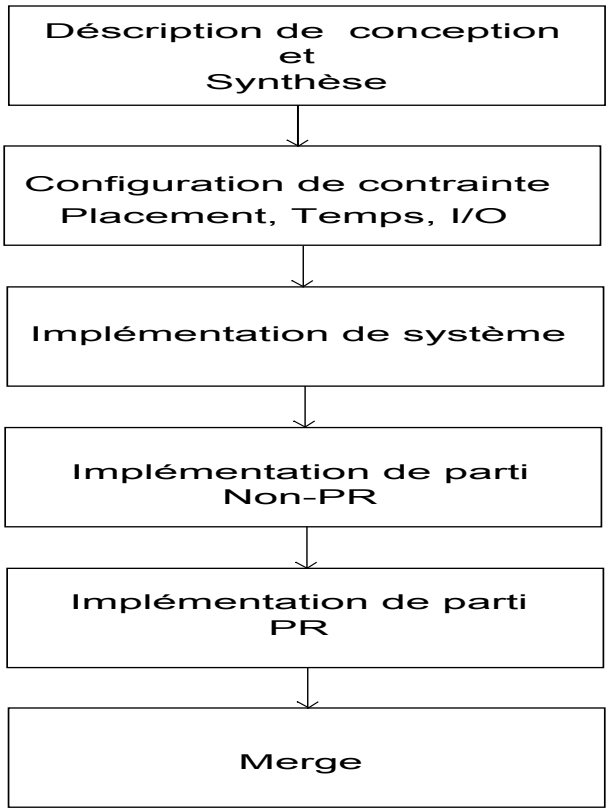

FIGURE A.1 – le flot de conception expérimenté

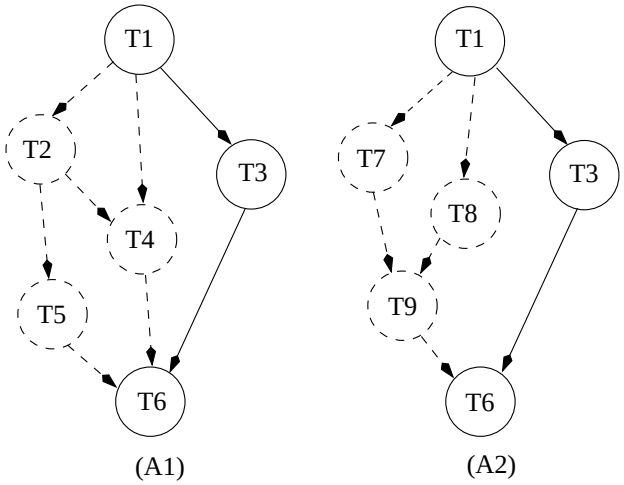

FIGURE A.2 – Description de l'application adaptative

mande une conception hiérarchique dans laquelle il est obligatoire de mettre les modules reconfigurables comme des sous-modules. Les netlists des modules reconfigurables vont être générées indépendamment (tableau A.1.1).

TABLE A.1 – Description de la conception HDL

Utilisation	Pour l'outil synthèse
entrée	system.vhd(niveau top) RPM1.vhd(module reconfigurable partiellement) RPM2.vhd base.vhd(statique parti)
sortie(XST)	system.ngc RPM1.ngc RPM2.ngc base.ngc

A.1.2 Modification de contrainte du système

D'après la description de la conception, VHDL est synthétisé, la prochaine étape est de placer des contraintes pour le placement et le routage (tableauA.1.2).

TABLE A.2 – Description de la conception HDL

Utilisation	Placement dans PlanAhead,Floorplanner ou text éditeur
entrée	system.ucf(niveau top)
sortie	system.ucf
commande	**ngdbuild -p** *type_composant* **-modular initial** *system.ugc*

Règles de la conception VHDL pour le module reconfigurable Chaque module reconfigurable est placé dans une région reconfigurable. Par ailleurs, les entrées et sorties du module reconfigurable connectent obligatoirement sur une interface à sens unique : BusMacro A.3.

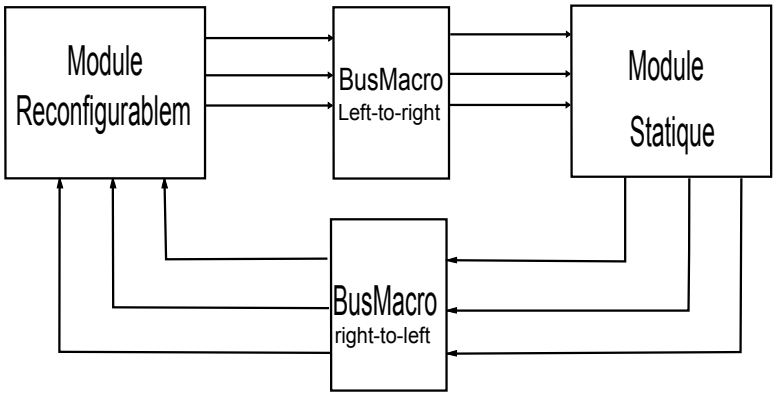

FIGURE A.3 – Interface pour la reconfiguration partielle : Slice Macro

La dernière version de BusMacro, illustrée dans la figure A.4, utilise deux CLBs dans le circuit FPGA au lieu de composants à trois états. Il supporte une interface à bande passant 8 octets avec ou sans signal *enable* (un exemple est illustré dans la figure A.4. Ce type de BusMacro a été développé par Hubener et al. [HBB04].

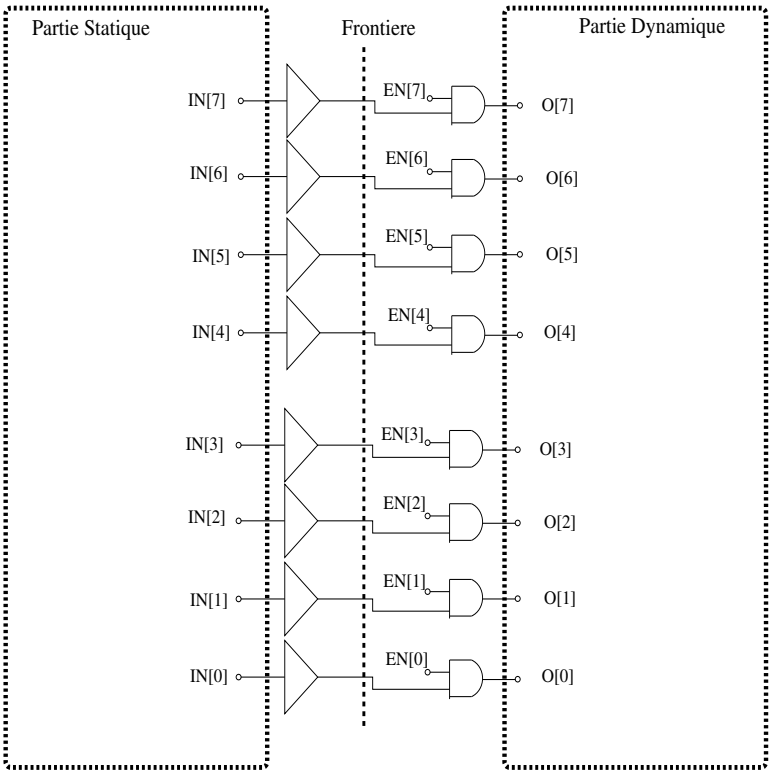

FIGURE A.4 – Interface pour la reconfiguration partielle : Slice Macro

Il faut que le nombre de cellules logiques dans une zone reconfigurable soit égal ou

supérieur au nombre requis par le module fonctionnel implémenté dans cette région. L'outil Floorplanner, PlanAhead peut-être utilisé pour estimer la taille du module reconfigurable. Le placement de BusMacro est illustré dans la figureA.6.

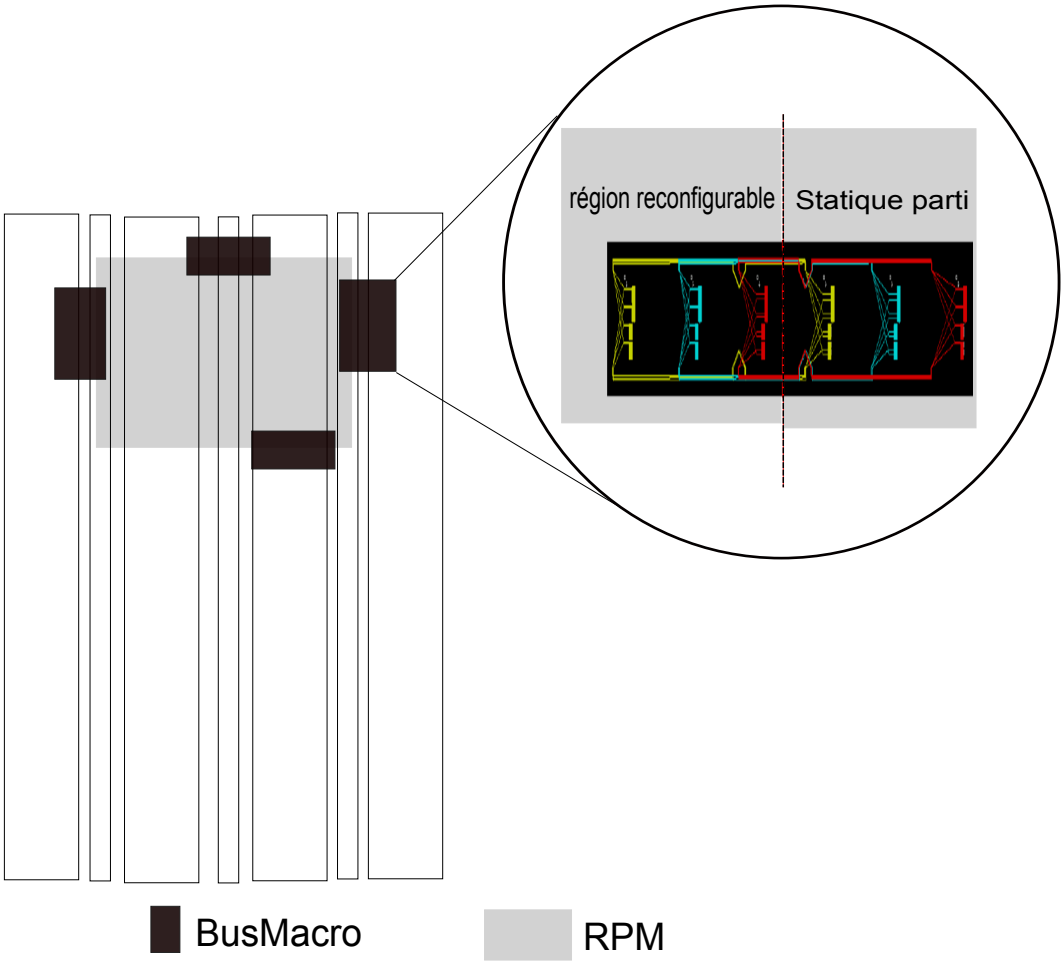

FIGURE A.5 – Placement de BusMacro

Il y a différentes versions de BusMacro permettant de choisir :

– Connexion directe :
  left-to-right et right-to-left pour Virtex-II/Pro ;
  left-to-right, right-to-left , top-to-bottom, or bottom-to-top pour Virtex-4,5 ;

– Différentes tailles : *Wide* - large de 4 CLBs ou *narrow* - large de 2 CLBs

– synchrones ou asynchrones

Le tableauA.3 montre les BusMacros du circuit FPGA Virtex-4

TABLE A.3 – BusMacro de Virtex-4

busmacro_xc4v_b2t_async_enable_narrow.nmc	busmacro_xc4v_r2l_async_enable_narrow.nmc
busmacro_xc4v_b2t_async_enable_wide.nmc	busmacro_xc4v_r2l_async_enable_wide.nmc
busmacro_xc4v_b2t_async_narrow.nmc	busmacro_xc4v_r2l_async_narrow.nmc
busmacro_xc4v_b2t_async_wide.nmc	busmacro_xc4v_r2l_async_wide.nmc
busmacro_xc4v_b2t_sync_enable_narrow.nmc	busmacro_xc4v_r2l_sync_enable_narrow.nmc
busmacro_xc4v_b2t_sync_enable_wide.nmc	busmacro_xc4v_r2l_sync_enable_wide.nmc
busmacro_xc4v_b2t_sync_narrow.nmc	busmacro_xc4v_r2l_sync_narrow.nmc
busmacro_xc4v_b2t_sync_wide.nmc	busmacro_xc4v_r2l_sync_wide.nmc
busmacro_xc4v_l2r_async_enable_narrow.nmc	busmacro_xc4v_t2b_async_enable_narrow.nmc
busmacro_xc4v_l2r_async_enable_wide.nmc	busmacro_xc4v_t2b_async_enable_wide.nmc
busmacro_xc4v_l2r_async_narrow.nmc	busmacro_xc4v_t2b_async_narrow.nmc
busmacro_xc4v_l2r_async_wide.nmc	busmacro_xc4v_t2b_async_wide.nmc
busmacro_xc4v_l2r_sync_enable_narrow.nmc	busmacro_xc4v_t2b_sync_enable_narrow.nmc
busmacro_xc4v_l2r_sync_enable_wide.nmc	busmacro_xc4v_t2b_sync_enable_wide.nmc
busmacro_xc4v_l2r_sync_narrow.nmc	busmacro_xc4v_t2b_sync_narrow.nmc
busmacro_xc4v_l2r_sync_wide.nmc	busmacro_xc4v_t2b_sync_wide.nmc

A.1.3 Implémentation de la partie Statique

L'implémentation de la partie statique est effectuée indépendamment de la partie reconfigurable. Elle est illustrée dans le tableau A.1.3.

TABLE A.4 – Description de l'implémentation de la partie statique

Utilisation	Placement dans PlanAhead,Floorplanner ou text éditeur
entrée	system.ucf(niveau top) netlists des module statique busmacro_nom.nmc
sortie	system_sansRPM_routed.ncd
Commands	MAP,PAR,et Bitgen
Notes	le(s) module(s) reconfigurable(s) n'est(sont) pas placé ici. La(es) région(s) reconfigurée(s) sont comme une boîte noire.

A.1.4 Implémentation de la partie reconfigurable partiellement

Après l'implémentation de la partie statique, tous les modules reconfigurables(RPM) nécessaires sont implémentés, comme illustré dans le tableauA.1.4.

TABLE A.5 – Description de l'implémentation de Module Reconfigurable partiellement

Utilisation	Placement dans PlanAhead,Floorplanner ou text éditeur
entrée	system.ucf(niveau top) system.ngc RPM1.ngc static.used busmacro_nom.nmc
sortie	system_sansRPM_routed.ncd RPM_routed.ncd
Commands	MAP,PAR,et Bitgen
Notes	le fichier static.used indique les informations de la placement des parties statiques

A.1.5 Merge

La dernière étape du flot de conception de la reconfiguration partielle est de merger les parties statiques, reconfigurables, et le top design ensemble. pour générer les

données de configuration de la partie statique (statique.bit) et de la partie reconfigurable(RPM.bit). Le détail de cette étape est illustré dans la tableauA.1.5.

TABLE A.6 – Description de l'implémentation de Module reconfigurable partiellement

Utilisation	génération de donnée de configuration de parti statique et de partie reconfigurable
entrée	system_sansRPM_routed.ncd RPM1.ncd
sortie	system_complet.bit RPM1.bit RPM_blank.bit
Commands	MAP,PAR,et Bitgen
Notes	Trois bitstreams sont générés dans cette étape RPM_blank.bit est le bistream pour effacer la zone reconfigurable.

A.2 Carte d'expérimentation -Virtex-4SX35

Les composants principaux :

- Virtex-4 sx35
- RS-232
- Ethernet MAC/PHY à 10/100 baseT
- LED Bleus
- Afficheurs 7-segments
- Port USB
- CPLD de gestion de la configuration/reconfiguration
- Connecteur JTAG
- 32M DDR RAM

A.2.1 MicroBlaze

Le microprocesseur MicroBlaze est utilisé dans notre expérimentation comme un contrôleur d'adaptation. Il possède :

- 32 registres internes de 32 bits de large
- un pipeline à 3 niveaux
- Fréquence de fonctionnement maximale à 133MHz
- un bus de données interne (DLMB)
- un bus d'instructions externe(IOPB)
- un bus de données externe (DOPB)
- un bus d'instruction interne(ILMB)

Le processeur est facilement configurable et permet à l'utilisateur de sélectionner ou de paramétrer les composants internes selon ses besoins :

- Utilisation des multiplieurs câblés du FPGA
- Opérateur de décalage(Barrel Shifter)
- Mémoire cache d'instructions et de données
- Logique de debug(interface XMD)

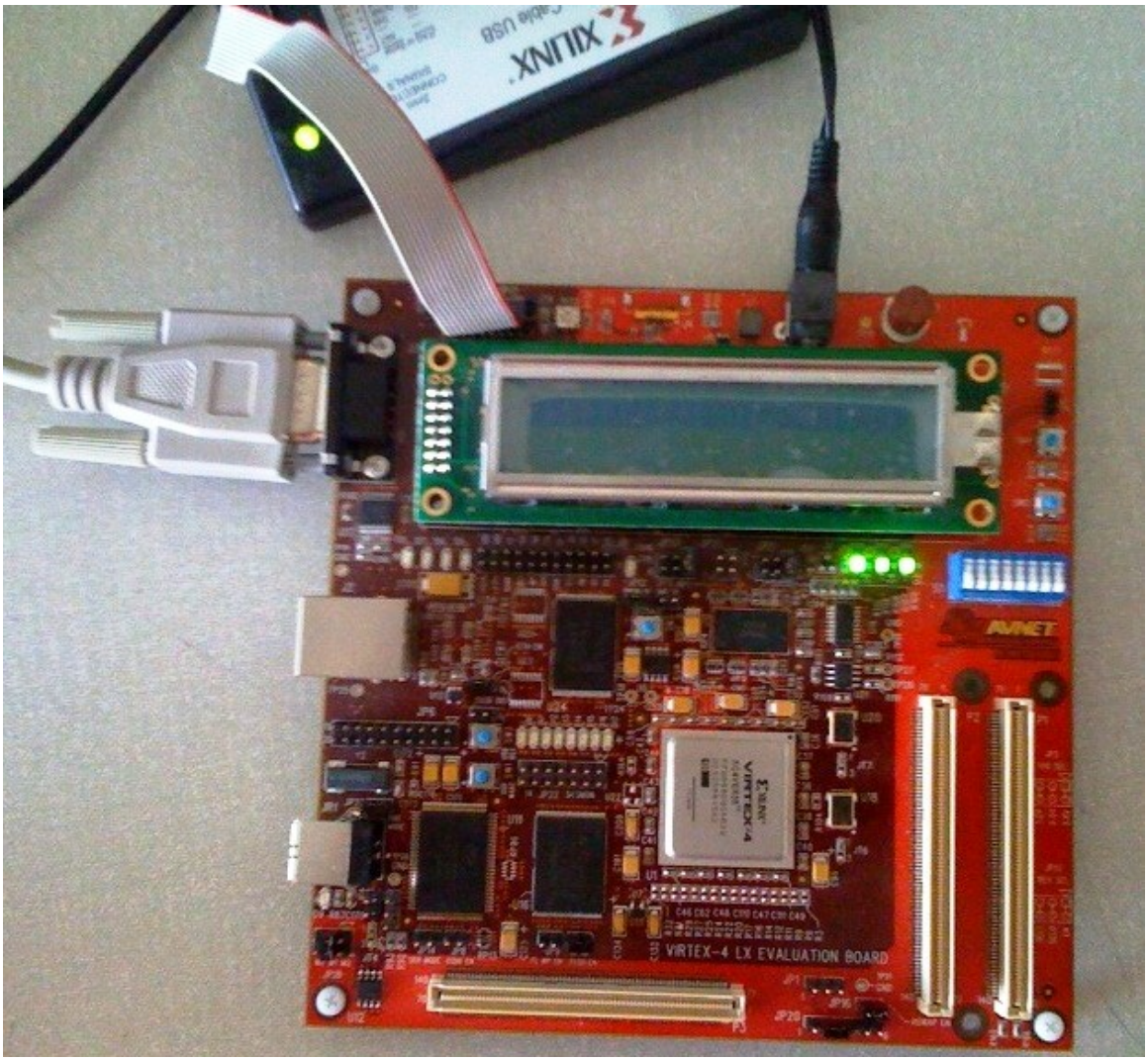

FIGURE A.6 – Plate-forme *Avnet* du circuit FPGA Virtex-4SX35 Xilinx

Il occupe entre 900 et 2600 cellules logiques et atteint une fréquence de 80 à 180MHz selon la plateforme et les options souhaitées.

A.2.2 Fast Simplex Link - FSL

Un bus asynchrone FSL est un moyenne de communication entre le microprocesseur et une autre entité. Chaque lien FSL 32bits est unidirectionnel et met en oeuvre une FIFO(pour stocker les données) et des signaux de contrôle(FULL,EMPTY, WRITE, READ,...). Les communications sur le lien FSL se font très simplement grâce à des instructions prédéfinies. Elle peuvent travailler aux différentes fréquences de fonctionnement des deux côtés. Ceci permet de réaliser facilement la gestion dynamique de fréquence partielle du circuit.

A.2.3 Xilinx matériel ICAP - HWICAP

La reconfiguration partielle dynamique de dispositifs FPGA avant à la famille VirtexII/Pro utilisent l'interface SelectMap ou JTAG. Cependant, ces modes de reconfiguration demandent un contrôleur externe pour envoyer les données de reconfiguration. Avec l'arrivée des familles VirtexII/Pro le fabricant a inclu une interface interne de reconfiguration nommée ICAP. L'interface ICAP inclut un ensemble réduit de l'interface SelectMap est qui ne supporte pas différents modes de reconfiguration.

Le périphérique qui est intégré dans EDK permet le contrôle de ce port de configuration par le processeur MicroBlaze/PowerPC. Ce périphérique PLB est composé du module ICAP, quelque logiques de contrôle et un cache de configuration en BRAM.

La fonction principale pour réaliser la reconfiguration partielle et la gestion dynamique de fréquence est fourni :

```
1 /* $Id: xhwicap_testapp.c,v 1.1 2007/11/01 13:58:56 svemula Exp $ */
2       /*****************************************************************************
3 *
4 *   Progamme de la reconfiguration Partiel via ICAP et la gestion de la fréquence
5 *   dyanamique. (exemple de code )
6 *   Trois fonction principals dans cette programme
```

```
7  *                1) initialisation de ICAP
8  *                2) téléchargement de bitstreams dans le ICAP
9  *                3) configuration de DCM via le contrôleur DFS
10       ******************************************************************************/
11 /*
12       /******************************************************************************/
13 /**
14 * @file icap_testapp.c
15 *
16 * This file contains a design example using the HwIcap device driver and
17 * hardware device.
18 *
19 * @note
20 *
21 * Ce fonction est testé sur le circuit FPGA Virtex-4 et 5.
22 * L'outil ISE 9.2 et EDK 9.2 est nécessaire
23 *
24 * MODIFICATION HISTORY:
25 *
26 * Ver   Who Date     Changes
27 * ----- ---- --------
       ----------------------------------------------------------------------
28 * 2.00a sv  10/04/07 Initial release.
29 * </pre>
30 *
31       ******************************************************************************/
32
33   /*************************** Include Files ********************************/
34
35 #include "dfs_enable_ipif.h" /* Pilot de la DFS contrôleur */
36 #include "xparameters.h"
37 #include "xhwicap.h"    /* Pilot de HWICAP */
38 #include "xhwicap_i.h"
39 #include "xhwicap_parse.h"
40   /************************** Constant Definitions **************************/
41
```

```
42 #ifndef TESTAPP_GEN
43 #define HWICAP_DEVICE_ID XPAR_HWICAP_0_DEVICE_ID //:
       XPAR_XPS_HWICAP_0_DEVICE_ID
44 #endif
45
46 #define TEST_WRITE_BUFFER_SIZE 37177
47   /************************** Type Definitions ****************************/
48   int Status;
49   XHwIcap_Config *ConfigPtr;
50   u32 ConfigRegData;
51   Xuint32 word;
52   Xuint8 data3,data2,data1,data0;
53   /* exemple de bitstreams sont testés
54   u32 *FrameBuffer=0xcd000000; /*l'adresse de la bitstreams un dans le mémoire DDR
55
56   /************************** Function définit ****************************/
57
58 int HwIcapTestAppExample(u16 DeviceId);
59
60   /************************** Variable Definitions ************************/
61
62 static XHwIcap HwIcap; /* The instance of the HWICAP device */
63
64       /*****************************************************************/
65
66       /*****************************************************************/
67 //#ifndef TESTAPP_GEN
68 /* lire l'entêt de la fichier bitstreams dans le mémoire DDR*/
69
70 XHwIcap_Bit_Header XHwIcap_ReadHeader(Xuint8 *Data, Xuint32 Size)
71 {
72     Xuint32 I;
73     Xuint32 Len;
74     Xuint32 Tmp;
75     XHwIcap_Bit_Header Header;
76     Xuint32 Index;
```

```
77
78      /* Start Index at start of bitstream */
79      Index=0;
80
81      /* Initialize HeaderLength. If header returned early inidicates
82       * failure .
83       */
84      Header.HeaderLength = XHI_BIT_HEADER_FAILURE;
85
86   /* une partie de code est ignoré dans cette annexe*/
87
88      /* Get byte length of bitstream */
89      Header.BitstreamLength = Data[Index++];
90      Header.BitstreamLength = (Header.BitstreamLength << 8) | Data[Index++];
91      Header.BitstreamLength = (Header.BitstreamLength << 8) | Data[Index++];
92      Header.BitstreamLength = (Header.BitstreamLength << 8) | Data[Index++];
93
94      Header.HeaderLength = Index;
95
96      return Header;
97
98  }
99
100 /* une fonction définit pour piloter la composant HWICAP */
101
102 int reconfiguration()
103 {
104 XHwIcap_Bit_Header bit_header;
105 bit_header = XHwIcap_ReadHeader((Xuint8*)FrameBuffer,0);
106
107  xil_printf ("the size of file is %ld\n\r",bit_header.BitstreamLength);
108    ConfigPtr = XHwIcap_LookupConfig(XPAR_HWICAP_0_DEVICE_ID);
109    if (ConfigPtr == NULL) {
110      return XST_FAILURE;
111       xil_printf (" failed \n\r ");
112    }
```

```
113 xil_printf ("XHwIcap_LookupConfig ok\n\r ");
114   Status = XHwIcap_CfgInitialize(&HwIcap, ConfigPtr,
115           ConfigPtr->BaseAddress);
116    if (Status != XST_SUCCESS) {
117      return XST_FAILURE;
118    }
119    /* initialiser  HWICAP*/
120 xil_printf ("XHwIcap_CfgInitialize ok\n\r ");
121 int i;
122 xil_printf (" start RP\n\r ");
123   /*télécharger bitstreams dans le HWICAP*/
124 for(i=0;i<bit_header.BitstreamLength;i+=4)
125 {
126           data3 = *(FrameBuffer1++);
127           data2 = *(FrameBuffer1++);
128           data1 = *(FrameBuffer1++);
129           data0 = *(FrameBuffer1++);
130           word = ((data3 << 24) | (data2 << 16) | (data1 << 8) | (data0));
131 Status = XHwIcap_DeviceWrite(&HwIcap,&word,1);
132
133 if (Status != XST_SUCCESS)
134 {
135 xil_printf ("error writing to ICAP (%d)\r\n", Status);return -1;
136 }
137 }
138
139 }
140
141 void menu(void)
142 {
143 xil_printf ("--------o------o------ooooo------o------o----\r\n");
144 xil_printf ("--------o-------------o----------o-o----o----\r\n");
145 xil_printf ("--------o------o------oooo-------o--o---o----\r\n");
146 xil_printf ("--------o------o------o----------o----o-o----\r\n");
147 xil_printf ("--------ooooo--o------ooooo------o------o----\r\n");
148 xil_printf ("-----------------------------------------------\r\n");
```

```
149
150 }
151
152 /*Une démonstration pour tester la fonctionnalité de l'adaptation*/
153 /*  DFS_ENABLE_IPIF_mWriteSlaveReg1 est la fonction pour modifier*/
154 /* les parametrers de DCM et puis génerer une nouvelle fréquence */
155 /*  (0,2) -> 33MHz, (0,3) -> 40MHz, (0,4) -> 50MHz, (0,5)-> 60MHz */
156     /************************************************************/
157
158 int main (void) {
159  char key1,key2;
160  menu();
161 key1=getchar();
162   xil_printf ("  \n\r key1 is %c  \n\r",key1);
163 while ((key1 != 'q') ) {
164    switch (key1)
165        {
166        case 'a':
167            DFS_ENABLE_IPIF_mWriteSlaveReg1(XPAR_DFS_ENABLE_IPIF_0_BASEADDR,0,2);
168            menu();
169      break;
170       case 'b':
171           DFS_ENABLE_IPIF_mWriteSlaveReg1(XPAR_DFS_ENABLE_IPIF_0_BASEADDR,0,3);
172        menu();
173            break;
174    case 'c':
175        DFS_ENABLE_IPIF_mWriteSlaveReg1(XPAR_DFS_ENABLE_IPIF_0_BASEADDR,0,4);
176         menu();
177               break;
178  case 'd':
179           DFS_ENABLE_IPIF_mWriteSlaveReg1(XPAR_DFS_ENABLE_IPIF_0_BASEADDR,0,5);
180             menu();
181                 break;
182         case 'e':
183 reconfiguration();
184 //menu();
```

```
185 break;
186         }
187  key1=getchar();
188         //  key1=scanf("%d",&key2);//        /* Wait for  another keystroke. */
189
190  } // end of while
191       return  0;
192 } // end main
```

Annexe B

Implémentation de l'exemple IDWT

Ce paragraphe donne un exemple d'utilisation du flot de conception détaillé dans l'annexe précédente. L'environnement de développement et les outils spécifiques utilisés dans le flot de la conception sont :

- ISE 9.2.sp_2
- EDK 9.2
- PlandAhead 10.1

Les Détails des outils ISE, EDK et PlanAhead dans l'environnement de développement sont donnés dans [Inc05c].

Un exemple de conception de décodeur IDWT, est illustré dans la figureB.1. Ce décodeur est testé sur la plate-forme de Avnet [7] avec le circuit FPGA Virtex-4sx35 xilinx.

Chaque PE est un module reconfigurable et est connecté à une interface commune via le BusMacro. La fonction de PE dans chaque zone reconfigurable peut être ajoutée ou écrasée avec le bitstreams(PE.bit et PE_blank.bit).

La figureB.2 illustre un exemple de conception d'architecture reconfigurable dans le circuit Virtex-4SX35 Xilinx.

Les données de configuration (système.bit) sont téléchargées une seule fois par JTAG. Les PEs sont configurées via le port de configuration ICAP. La figure B.3 illustre un exemple du placement d'un PE. La différence avant et après de la configuration partielle est montrée par la connexion de SliceMaco. Nous voyons clairement que la

7. www.avnet.com/fr

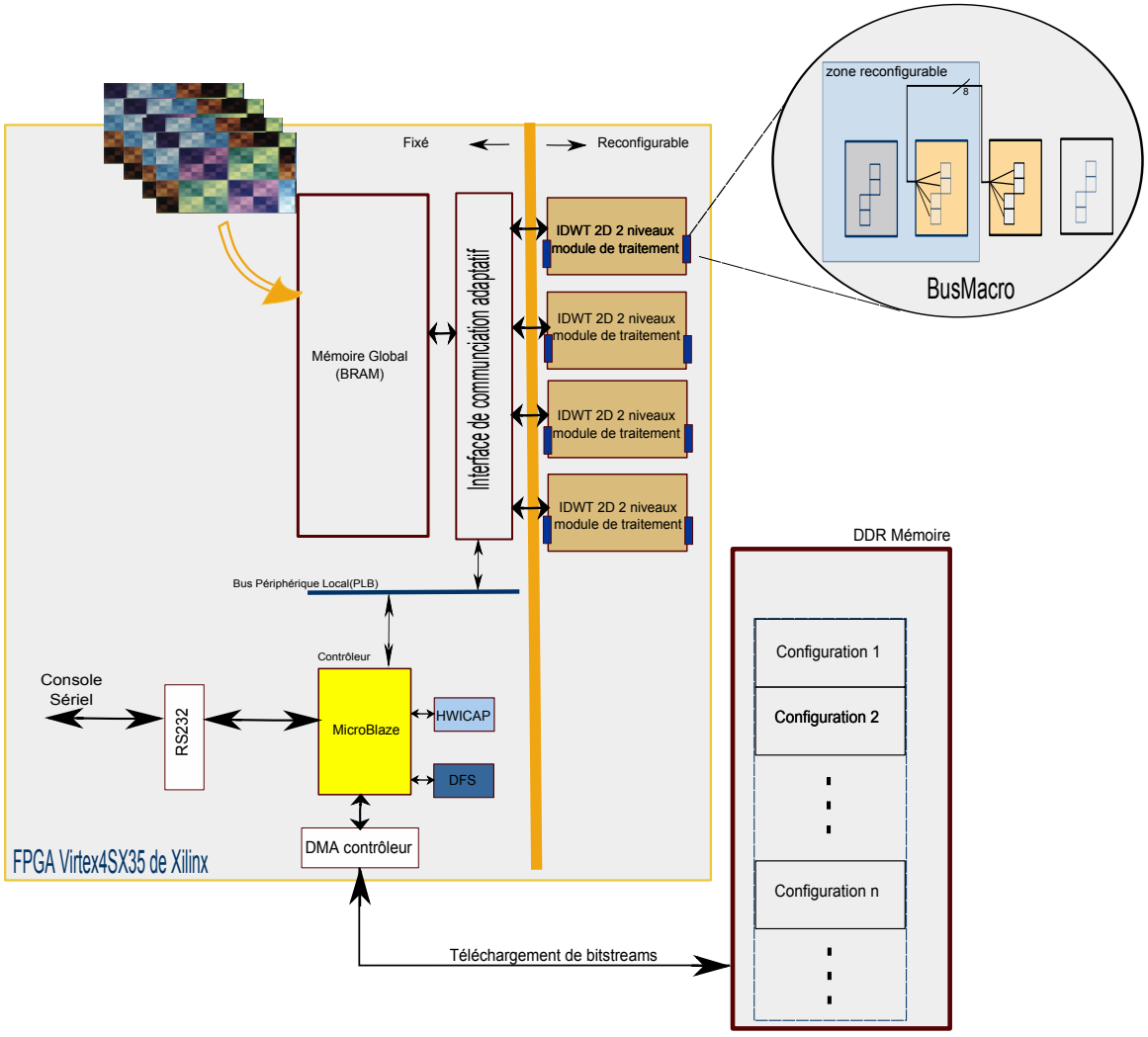

FIGURE B.1 – Vu global de l'architecture expérimentale

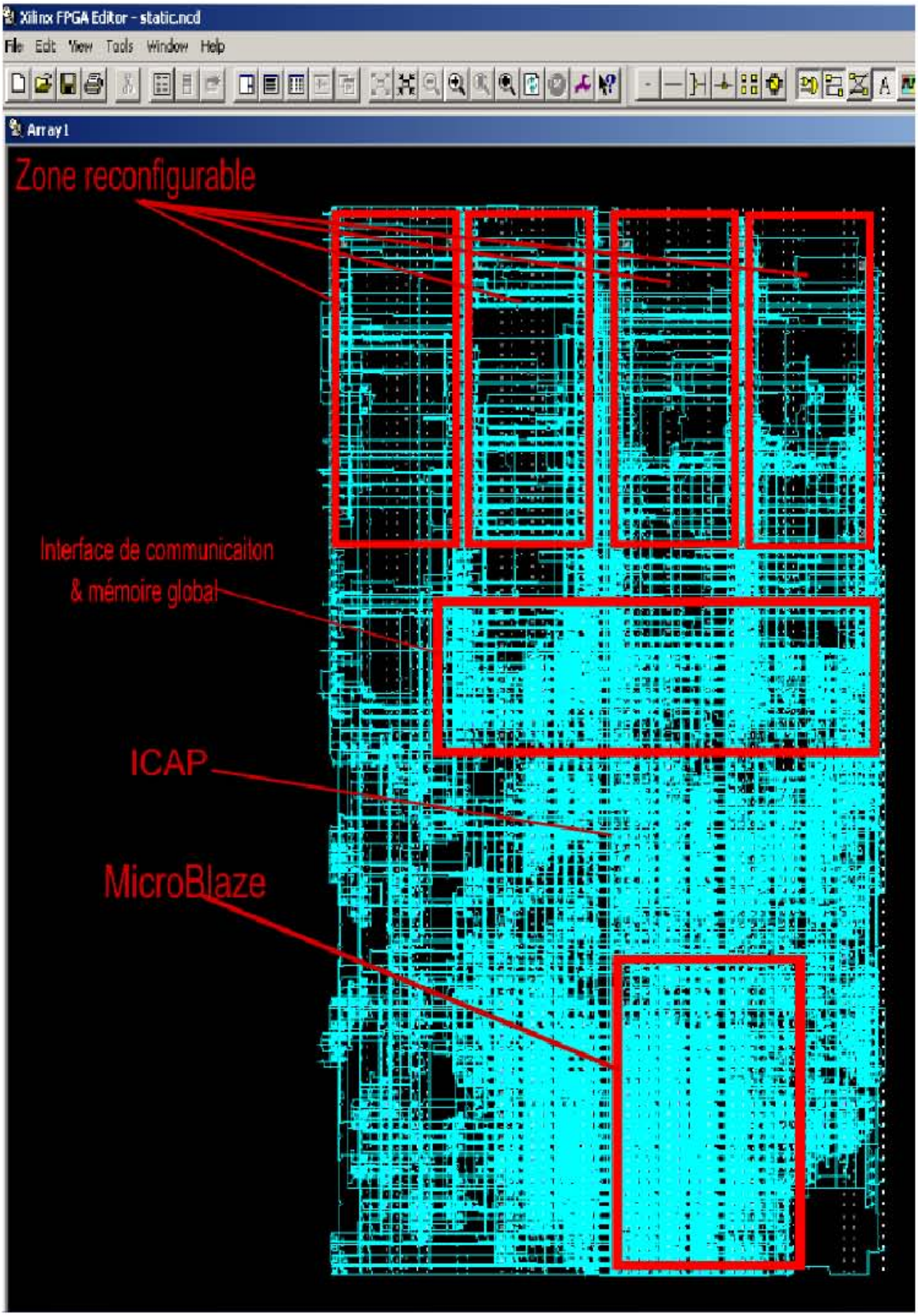

FIGURE B.2 – Placement de la partie statique du système avec la défintion des zones reconfigurables

SliceMacro ne connecte aucune cellule logique avant la configuration par rapport au schéma qui montre le résultat de la reconfiguration.

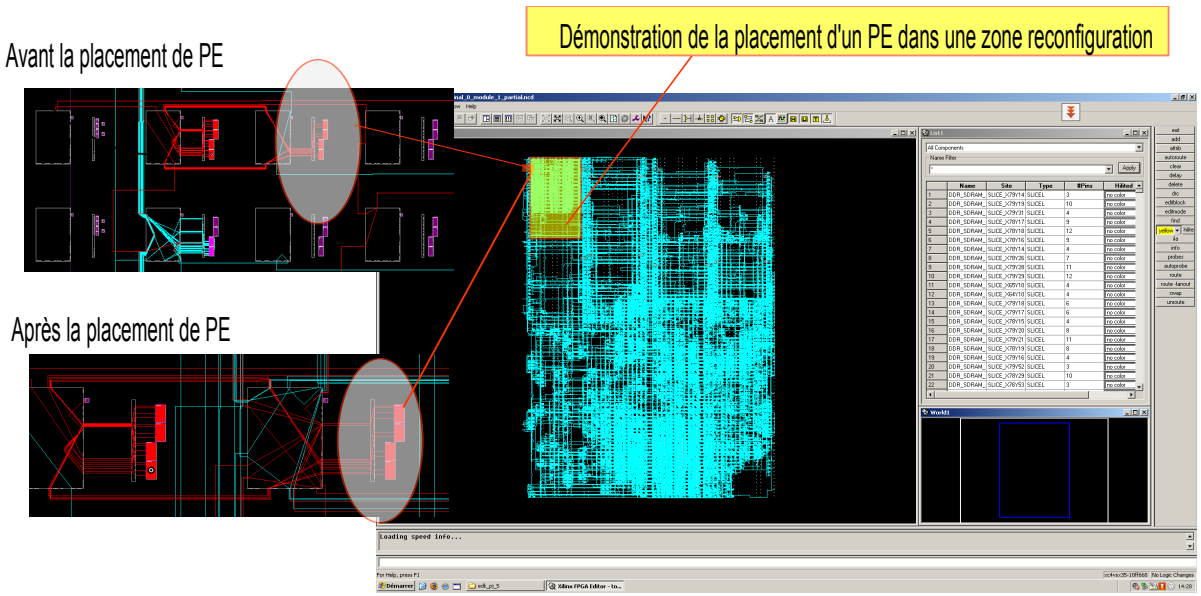

FIGURE B.3 – Implémentation de l'architecture de multi PE IDWT

Chaque PE a la même fonction, mais a son propre bitstreams :

- PE_1_fonction_partiel.bit
- PE_1_fonction_partiel_blank.bit
- PE_2_fonction_partiel.bit
- PE_2_fonction_partiel_blank.bit
- PE_3_fonction_partiel.bit
- PE_3_fonction_partiel_blank.bit
- PE_4_fonction_partiel.bit
- PE_4_fonction_partiel_blank.bit

Le bitstream $PE\_1\_fonction\_partiel.bit$ contient la configuration des cellules logiques dans une zone reconfigurable fixée. Le bitstreams $PE\_1\_fonction\_partiel\_blank.bit$ est utilisé pour effacer les configurations de cette zone reconfigurable. Cette dernière permet de désactiver la fonction du PE via la reconfiguration.

On enregistre tous les bitstreams dans la mémoire externe DDR. Ceci permettant de réduire l'utilisation de ressources sur puce et donc de rendre plus précise notre

mesure de la consommation de puissance. Un exemple d'organisation de mémoire est illustré dans la figure B.4. Le bitstreams débute par un fichier en tête, qui contient des informations telles que la date de création, la taille de ce bitstream, la position dans le circuit, etc. La fonction illustrée dans l'annexe-A, $XHwIcap\_ReadHeader$ ($70ime$ ligne de code),permet de récupérer toutes ces informations.

L'entête du bitstreams de configuration s'achève toujours par un mote de 32bits : $FFFFFFFF$ marqué dans la figureB.4. La fonction, $reconfigure()$, permet de télécharger les bitstreams en 8 octets dans le circuit par ICAP.

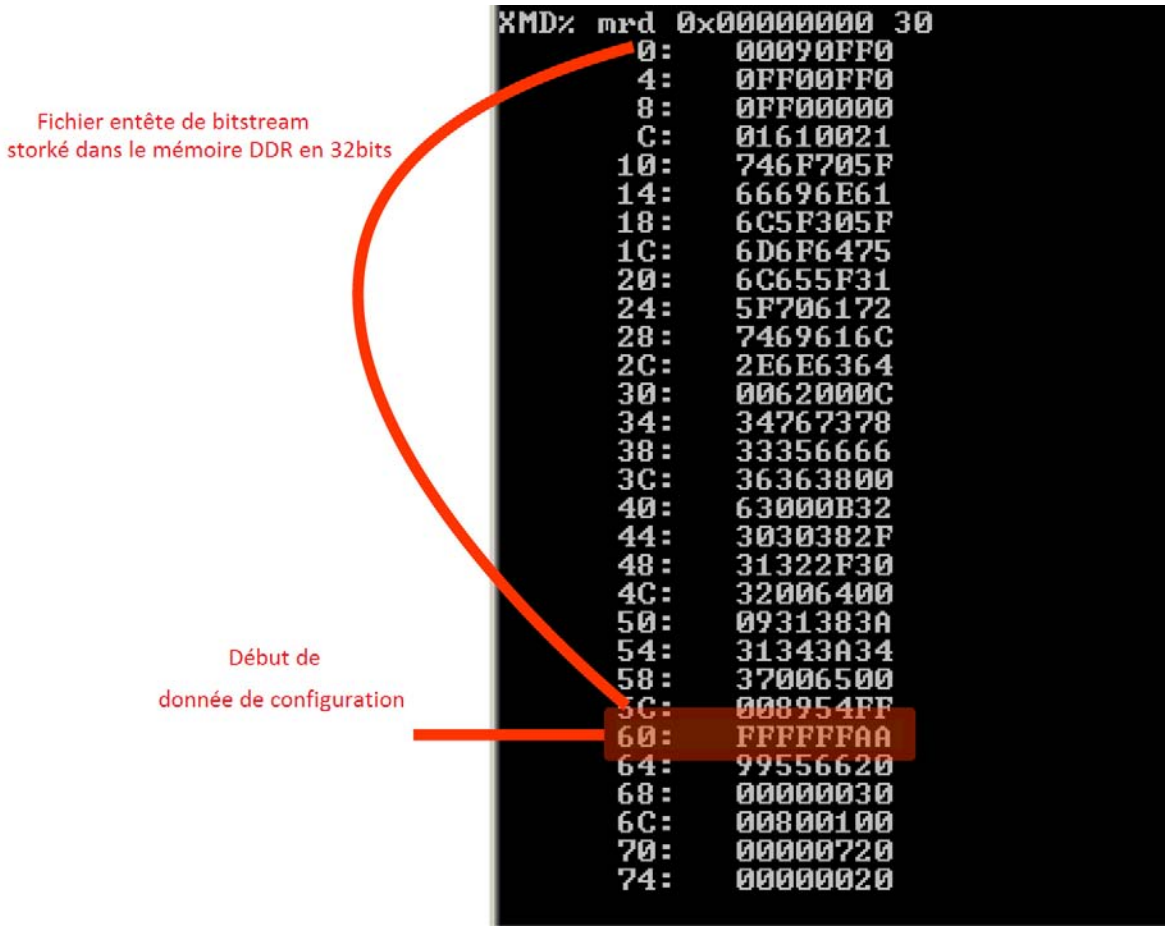

FIGURE B.4 – Exemple de stockage du bitstreams dans la mémoire DDR

Bibliographie

[Abe04] AZZEDINE ABEDENOUR : *Outil d'analyse et de partitionnement/ordonnancement Pour les système temps réel embarqués*. Thèse de doctorat, UFR Sciences et sciences de l'engenieur, 2004.

[Abn01] Arthur ABNOUS : *Low-Power Domain Specific Processors for Digital Signal Processing*. Thèse de doctorat, EECS Department, University of California, Berkeley, 2001.

[ACT01] A.SKODRAS, C.CHRISTOPOULOS et T.EBRAHIMI : The jpeg2000 still image compression standard. *IEEE Signal Processing Magazine*, pages 36–58, 2001.

[A.D00] A.D.GARCIA : Etude sur l'estimation et l'optimisation de la consommation de puissance des circuits logiques programmable du type fpga. *PHD these de Ecole National superieure des Telecommunications*, 2000.

[Ada01] M.D ADAMS : The jpeg2000 still image compression standard. Rapport technique, ISO/IEC JTC1/SC29/WG1, 2001.

[AIWB98] A.R.CALDERBANK, I.DAUBECHIES, W.SWELDENS et B.L.YEO : Wavelet tranforms that map integers to integers. *Applied and Computation*, 5(3):332–369, 1998.

[AKM06] Carsten ALBRECHT, Roman KOCH et Erik MAEHLE : Dynacore : A dynamically reconfigurable coprocessor architecture for network processors. *In PDP '06 : Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 101–108, Washington, DC, USA, 2006. IEEE Computer Society.

[A.M95] A.MORSE : Control using logic-based switching. *Trands in Control, Springer, London*, 1995.

[AN04] J.H. ANDERSON et F.N. NAJM : Power estimation techniques for fpgas. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(10):1015–1027, Oct. 2004.

[ANC03] François ANCEAU : Une technique de réduction de la puissance dissipée par l'horlogerie des circuit complexes rapides. *4ème journée francophones d'étude Faible Tension Faible Consommation(FTFC'03)*, 2003.

[AZB+07] Armando ASTARLOA, Aitzol ZULOAGA, Unai BIDARTE, José Luis MARTÍN, Jesús LÁZARO et Jaime JIMÉNEZ : Tornado : A self-reconfiguration control system for core-based multiprocessor csopcs. *J. Syst. Archit.*, 53(9):629–643, 2007.

[BB95] T.D. BURD et R.W. BRODERSEN : Energy efficient cmos microprocessor design. *28th Annual Hawaii Internaional Conference on System Sciences*, 1:288–297, 1995.

[BBD05] Sudarshan BANERJEE, Elaheh BOZORGZADEH et Nikil D. DUTT : Physically-aware hw-sw partitioning for reconfigurable architectures with partial dynamic reconfiguration. *In* William H. Joyner JR., Grant MARTIN et Andrew B. KAHNG, éditeurs : *DAC*, pages 335–340. ACM, 2005.

[BBD06] S. BANERJEE, E. BOZORGZADEH et N. D. DUTT : Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(11):1189–1202, Nov. 2006.

[BBD09] S. BANERJEE, E. BOZORGZADEH et N. DUTT : Exploiting application data-parallelism on dynamically reconfigurable architectures : Placement and architectural considerations. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(2):234–247, Feb. 2009.

[BBMP04] Sophie BOUCHOUX, El-Bay BOURENNANE, Johel MITERAN et Michel

PAINDAVOINE : Implementation of jpeg2000 arithmetic decoder on a dynamically reconfigurable atmel fpga. *isvlsi*, 00:237, 2004.

[BBW+01] Reinaldo A. BERGAMASCHI, Subhrajit BHATTACHARYA, Ronoldo WAGNER, Colleen FELLENZ, Michael MUHLADA, William R. LEE, Foster WHITE et Jean-Marc DAVEAU : Automating the design of socs using cores. *IEEE Des. Test*, 18(5):32–45, 2001.

[BDHM05] Roman BARTOSINSKI, Martin DANĚK, Petr HONZÍK et Rudolf MATOUŠEK : Dynamic reconfiguration in fpga-based soc designs (abstract only). pages 274–274, 2005.

[BFY+02] M BALEANI., Gennari F., Jiang Y., Patel Y., Brayton R. et Sangiovanni-Vincentelli A. : Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. *10th international workshop on hardware/software codesign(CODES/CASHE 2002)*, pages 156–6, 2002.

[BL00] Francisco BARAT et Rudy LAUWEREINS : Reconfigurable instruction set processors : A survey. *In RSP '00 : Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, page 168, Washington, DC, USA, 2000. IEEE Computer Society.

[BLD02] Francisco BARAT, Rudy LAUWEREINS et Geert DECONINCK : Reconfigurable instruction set processors from a hardware/software perspective. *IEEE Trans. Softw. Eng.*, 28(9):847–862, 2002.

[Bob05] Christophe BOBDA : Building up a course in reconfigurable computing. *Microelectronics Systems Education, IEEE International Conference on/Multimedia Software Engineering, International Symposium on*, 0:7–8, 2005.

[Bol07] Ivo BOLSENS : Energy-efficient system design : Power optimization techniques in silicon and software have become mainstream considerations in fpga system design. Rapport technique, 2007.

[CA05] S. CHANDRASEKARAN et A. AMIRA : High speed/low power architectures for the finite radon transform. pages 450–455, Aug. 2005.

[CH02] Katherine COMPTON et Scott HAUCK : Reconfigurable computing : a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[Cro99] D. CRONQUIST : Architecture design of reconfigurable pipelined datapaths, 1999.

[CSS01] C. CHEN, A. SRIVASTAVA et M. SARRAFZADEH : On gate level power optimization using dual supply voltages, 2001.

[Cur07] Derek CURD : White paper : Virtex-5 fpgas –power consumption in 65nm fpgas. Rapport technique, Xilinx, 2007.

[DCPS02] R. DAVID, D. CHILLET, S. PILLEMENT et O. SENTIEYS : Dart : A dynamically reconfigurable architecture dealing with next generation telecommunications constraints. *9th IEEE Reconfigurable Architecture Workshop RAW*, 2002.

[DeH96] Andre' DEHON : Reconfigurable architectures for general-purpose computing. (AITR-1586):368, 1996.

[DLRN00] D.DEMINGNY, L.KESSAL, R.BOURGUIBA et N.BOUDOUANI : How to use high speed reconfigurable fpga for real time image processing ? *IEEE Conf. on Computer Architecture for Machine Perception, IEEE Circuit and Systems,Padova*, pages 240–246, Septembre,2000.

[DPF+00] D.GRUNWALD, P.LEVIS, K.I. FARKAS, C.B.Morrey III et M.NEUFELD : Policies for dynamic clock scheduling. *USENIX Symposium on Operating System Design and Implementation*, pages 73–86, 2000.

[DRP] Nec, http ://www.nec.com.

[Ess07] Juanjo Noguera and Robert ESSER : Application-driven rsearch in partial reconfiguration. Rapport technique, 2007.

[G. 02] L. Torres P. Benoit T. Gil C. Diou G. Cambon J. Galy G. SASSATELLI : Highly scalable dynamically reconfigurable systolic ring-architecture for dsp applications. *Automation and Test in Europe Conference and Exhibition (DATE'02)*, page 0553, 2002.

[GSM+99] Seth Copen GOLDSTEIN, Herman SCHMIT, Matthew MOE, Mihai BUDIU, Srihari CADAMBI, R. Reed TAYLOR et Ronald LAUFER : Piperench : A co-processor for streaming multimedia acceleration. *In ISCA*, pages 28–39, 1999.

[Har06] Reiner HARTENSTEIN : Why we need reconfigurable computing education. *In Opening session of the 1st International Workshop on Reconfigurable Computing Education*, 2006.

[HBB04] M. HEUBNER, T. BECKER et J. BECKER : Real-time lut-based network topologies for dynamic and partial fpga self-reconfiguration. *Integrated Circuits and Systems Design, 2004. SBCCI 2004. 17th Symposium on*, pages 28–32, Sept. 2004.

[HG07] Thomas HOLLSTEIN et Manfred GLESNER : Advanced hardware/software co-design on reconfigurable network-on-chip based hyper-platforms. *Computers and Electrical Engineering*, 2007.

[HK95] Reiner W. HARTENSTEIN et Rainer KRESS : A datapath synthesis system for the reconfigurable datapath architecture. *In ASP-DAC '95 : Proceedings of the 1995 conference on Asia Pacific design automation (CD-ROM)*, page 77, New York, NY, USA, 1995. ACM Press.

[Inc04a] Xilinx INC. : Xilinx inc : Virtex-4 family overview. Rapport technique, Xilinx, San Jose, Calif, USA, 2004.

[Inc04b] Xilinx INC. : two flows for partial reconfiguration : mode based or difference based. Rapport technique, Xilinx inc, sep. 2004.

[Inc05a] Altera INC. : Stratix ii devic handbook,. Rapport technique, Altera,San Jos,Calif, USA, 2005.

[Inc05b] Xilinx INC. : Virtex-ii pro and virtex-ii pro platform fpgas : Complete datasheet. Rapport technique, In Xilinx, San Jose,Calif USA, 2005.

[Inc05c] Xilinx INC. : using partial reconfiguration to time-share device resources in virtex-ii and virtex-ii pro(early-access partial reconfiguration documenta-

tion for virtex-ii and virtex-ii pro devices). Rapport technique, Xilinx, May, 2005.

[Inc06] Xilinx INC : Virtex-5 family overview. Rapport technique, Xilinx, San Jose, Calif, USA, 2006.

[jpe] Iso/iec international standard 15444-1 jpeg2000 image coding system. Rapport technique.

[JPGCP04] J-PH.DELAHAYE, G.GOGNIAT, C.ROLAND et P.BOMEL : Software radio and dynamic reconfiguration on a dsp/fpga platform. *Frequenze, Journal of Telecommunications*, 58:152–159, 2004.

[KBW03] Noha KAFAFI, Kimberly BOZMAN et Steven J. E. WILTON : Architectures and algorithms for synthesizable embedded programmable logic cores. *In FPGA '03 : Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 3–11, New York, NY, USA, 2003. ACM Press.

[KST01] K.FLAUTNER, S.REINHARDT et T.MUDGE : Automatic performance-setting for dynamic voltage scaling. *the 7th Conference on Mobile Computing and Networking MOBICOM'01(Rome,Italy)*, July 2001.

[KVGO99] Meenakshi KAUL, Ranga VEMURI, Sriram GOVINDARAJAN et Iyad OUAISS : An automated temporal partitioning and loop fission approach for fpga based reconfigurable synthesis of dsp applications. pages 616–622, 1999.

[LAK02] L.SHANG, A.S.KAVIANI et K.BATHALAB : Dynamic power consumption in virtex-ii fpga familly. *International Symposium on Field Programmable Gate Arrays*, pages 157–164, 2002.

[LKS00] Jae Seung LEE, Sang-Choon KIM et Seung Won SOHN : A design of the security evaluation system for decision support in the enterprise network security management. *In Information Security and Cryptology*, pages 246–260, 2000.

[LL05] Philip LEONG et Wayne LUK : Dynamic Voltage Scaling for Commercial FPGAs. *In International Conference on Field Programmable Technology (FPT)*, pages 215–222, December 2005.

[LL06] Sze-Wei LEE et Soon-Chieh LIM : Vlsi design of a wavelet processing core. *IEEE Transations on circuits and systems for video technology*, 16, 2006.

[LLHC04] F. LI, Y. LIN, L. HE et J. CONG : Low-power fpga using pre-defined dual-vdd/dual-vt fabrics, 2004.

[LLTB] Andrew LAFFELY, Jian LIANG, Russell TESSIER et Wayne BURLESON : Adaptive system on a chip (asoc) : A backbone for power-aware signal processing cores.

[LM05] Sebastian LANGE et Martin MIDDENDORF : On the design of two-level reconfigurable architectures. *reconfig*, 0:9, 2005.

[MB98] M.J.WIRTHLIN et B.L.HUTCHINGS : Improving functional density using run-time circuit reconfiguration. *IEEE tran. VLSI Syst.*, 6:247–256, 1998.

[MCM+04] Leandro MÖLLER, Ney Laert Vilar CALAZANS, Fernando Gehm MORAES, Eduardo Wenzel BRIÃO, Ewerson CARVALHO et Daniel CAMOZZATO : Fipre : An implementation model to enable self-reconfigurable applications. 3203:1042–1046, 2004.

[MLw98] Malcolm MCILHAGGA, Ann LIGHT et Ian WAKEMAN : Towards a design methodology for adaptive applications. *Mobile computing and networking*, pages 133–144, 1998.

[MMPI92] M.ANTONINI, M.BARLAUD, P.MATHIEU et I.DAUBECHIES : Image coding using wavelet transform. *IEEE Trans. on Image Processing*, 1:205–220, 1992.

[MNC+03] J. MIGNOLET, V. NOLLET, P. COENE, D. VERKEST et V. LAUWREINS : Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip, 2003.

[MVV+03] Bingfeng MEI, Serge VERNALDE, Diederik VERKEST, Hugo De MAN et Rudy LAUWEREINS : ADRES : An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. 2778:61–70, 2003.

[NB02] J NOGUERA et R BADIA : Hw/sw co-design techniques for dynamically reconfigurable architecture. *IEEE Trans VLSI System*, pages 399–415, 2002.

[NMB+03] - NOLLET, J. MIGNOLET, T. BARTIC, D. VERKEST, S. VERNALDE et R. LAUWEREINS : Hierarchical run-time reconfiguration managed by a operating system for reconfigurable systems, 2003.

[PBB98] T. PERING, T. BURD et R. BRODERSEN : Dynamic voltage scaling and the design of a low-power microprocessor system, 1998.

[PHB09] Katarina PAULSSON, Michael HÜBNER et Jürgen BECKER : Dynamic power optimization by exploiting self-reconfiguration in xilinx spartan 3-based systems. *Microprocess. Microsyst.*, 33(1):46–52, 2009.

[PHBB07] Katarina PAULSSON, Michael HÜBNER, Salih BAYAR et Jürgen BECKER : Exploitation of run-time partial reconfiguration for dynamic power managment in xilinx spartan iii-based systems. *In ReCoSoC'7 : Reconfigurable Communication-centric SoCs, 3rd International Workshop on Reconfigurable Communication-centric Systems-on-Chip, Montpellier, France*, 2007.

[PTDK08] Ilias POLITIS, Michail TSAGKAROPOULOS, Tasos DAGIUKLAS et Stavros KOTSOPOULOS : Power efficient video multipath transmission over wireless multimedia sensor networks. *Mob. Netw. Appl.*, 13(3-4):274–284, 2008.

[QSN07] Yang QU, Juha-Pekka SOININEN et Jari NURMI : Static scheduling techniques for dependent tasks on dynamically reconfigurable devices. *J. Syst. Archit.*, 53(11):861–876, 2007.

[Rab97] J. RABAEY : Reconfigurable processing : The solution to low-power programmable dsp. *In ICASSP '97 : Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '97) -Volume 1*, page 275, Washington, DC, USA, 1997. IEEE Computer Society.

[RIWB98] R.C.CALDERBANK, I.DAUBECHIES, W.SWELDENS et B.L.YEO : Wavelet trans that map integers to integers. *Applied and Computational Harmonic Analysis*, 5:332–369, 1998.

[RLAR05] R.KHASGIWALE, L.KRNAN, A.PERINKULAM et R.TESSIER : Reconfigurable data acquisition system for weather radar application. *In 48th Midwest Symposium on Circuits and Systems(MWSCAS'05),Cincinnati,Ohio,USA*, 2005.

[RMVC05] Javier RESANO, Daniel MOZOS, Diederik VERKEST et Francky CATTHOOR : A reconfiguration manager for dynamically reconfigurable hardware. *IEEE Design and Test of Computers*, 22(5):452–460, 2005.

[SB06] Love SINGHAL et Elaheh BOZORGZADEH : Physically-aware exploitation of component reuse in a partially reconfigurable architecture. *In IPDPS*. IEEE, 2006.

[SD99] S.A.GUCCIONE et D.LEVI : The advantage of run-time reconfiguration. *Reconfigurable Technology : FPGAs for Computing and Applications, SPIE -The International Society for Optical Engineering*, pages 87–92, 1999.

[SEJN06] S.BANERJEE, E.BOZORGZADEH, J.NOGUERA et N.DUTT : Minimizing peak power for applicaiton chains on architectures wih partial dynamic reconfiguration. *IEEE International Conference on Field Programmable Technology*, pages 273–276, 2006.

[SN05] S.P.MOHANTY et N.RANGANATHAN : Energy-efficient datapath scheduling using multiple voltages and dynamic clocking. *ACM Transactions on Design Automation of Electronic Systems*, 10:330–353, 2005.

[TBB98] T.PERING, T. BURD et R. BRODERESEN : Dynamic voltage scaling and the design of a low-power microprocessor system. *Power-Driven Microarchitecture Workshop in conjunction with Intl. Symposium on Computer Architecture,Barcelona, Spain*, 1998.

[TBWB03] Camel TANOUGAST, Yves BERVILLER, Serge WEBER et Philippe BRUNET : A partitioning methodology that optimises the area on reconfigurable real-time embedded systems. *EURASIP J. Appl. Signal Process.*, 2003:494–501, 2003.

[TGM07] T.NIKOLIC, G.DJORDEVIC et M.STOJCEV : Micro-power simple porcessing element. *In VIII National conference ETAI*, 2007.

[TP04] T.ACHARYA et P.S.TSAI : Jpeg2000 standard for image compression concepts, algorithms and vlsi architecture. 2004.

[TSMM06] Jesús TABERO, Julio SEPTIÉN, Hortensia MECHA et Daniel MOZOS : Task placement heuristic based on 3d-adjacency and look-ahead in reconfigurable systems. pages 396–401, 2006.

[TTR00] T.PERING, T.BURD et R.BORDERSEN : Voltage scheduling in the lparm microprocessor system. *ISLPED'00 Rapallo, Italy*, pages 96–101, 2000.

[Ua04] Michael ULLMANN et A.L : On-demand fpga run-time system for dynamical reconfiguration with adaptive priorities. *Field Programmable Logic and Application*, 3203/2004:454–463, 2004.

[UH95] Kimiyoshi USAMI et Mark HOROWITZ : Clustered voltage scaling technique for low-power design. *In ISLPED '95 : Proceedings of the 1995 international symposium on Low power design*, pages 3–8, New York, NY, USA, 1995. ACM.

[UHGB04] Michael ULLMANN, Michael HUBNER, Grimm GRIMM et Jurgen BECKER : An fpga run-time system for dynamical on-demand reconfiguration. *ipdps*, 04:135a, 2004.

[Wol00] Wayne WOLF : *Computers as Components : Principles of Embedded Computer Systems Design*. Morgan Kaufmann, 2000.

[ZBR00] Ning ZHANG, BRODERSEN et R.W. : Architectural evaluation of flexible digital signal processing forwireless receivers. *Signals, Systems and Computers*, 1(1):78–83, 2000.

[ZRW07] Xun ZHANG, Hassan RABAH et Serge WEBER : Auto-adaptive reconfigurable architecture for scalable multimedia applications. *Adaptive Hardware and Systems, NASA/ESA Conference on*, 0:139–145, 2007.

[ZRW08] Xun ZHANG, Hassan RABAH et Serge WEBER : Dynamic slowdown and partial reconfiguration to optimize energy in fpga based auto-adaptive

sopc. *Fourth IEEE International Symposium on Electronic Design, Test and Applications, DELTA 2008*, 2008.

[ZWGR99] Hui ZHANG, Marlene WAN, Varghese GEORGE et Jan RABAEY : Interconnect architecture exploration for low-energy reconfigurable single-chip dsps. *In WVLSI '99 : Proceedings of the IEEE Computer Society Workshop on VLSI'99*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.