Méthodologie de conception d'architectures reconfigurables dynamiquement pour des applications temps-réel

François Duhem

▶ To cite this version:

François Duhem. Méthodologie de conception d'architectures reconfigurables dynamiquement pour des applications temps-réel. Systèmes embarqués. Univ. Nice-Sophia Antipolis, 2012. Français. NNT : . tel-01573906

HAL Id: tel-01573906
https://theses.hal.science/tel-01573906

Submitted on 10 Aug 2017

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.

UNIVERSITE DE NICE - SOPHIA ANTIPOLIS
ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H E S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis
Mention : Electronique

Présentée et soutenue par

François DUHEM

Méthodologie de conception d'architectures reconfigurables dynamiquement pour des applications temps-réel

Thèse dirigée par Fabrice MULLER & Philippe LORENZINI

préparée dans l'équipe MCSOC
de l'unité de recherche UMR 7248 - LEAT/CNRS

soutenue le 23 novembre 2012

Jury :

M.	Serge Weber	Professeur, LIEN	Président du jury
M.	Bertrand Granado	Professeur, LIP6	Rapporteur
M.	Christophe Jégo	Professeur, IMS	Rapporteur
M.	François Verdier	Professeur, LEAT	Examinateur
M.	Fabrice Muller	Maître de conférences HDR, LEAT	Co-Directeur de thèse
M.	Philippe Lorenzini	Professeur, IM2NP	Directeur de thèse

Résumé

La reconfiguration dynamique des FPGA, malgré des caractéristiques intéressantes, peine à s'installer dans l'industrie principalement pour deux raisons. Tout d'abord, les performances du contrôleur natif développé par Xilinx sont faibles et pourront résulter en un rapport entre le temps de reconfiguration et la période de la tâche trop important pour une implémentation dynamique. Ensuite, le développement d'une application reconfigurable dynamiquement demande un effort plus conséquent, notamment concernant l'ordonnancement des tâches. Il est en effet impossible d'évaluer une architecture et/ou un algorithme d'ordonnancement pour vérifier si l'application respectera bien ses contraintes de temps avant la phase d'implémentation.

Cette thèse s'inscrit dans ce contexte et propose des solutions aux problématiques énoncées précédemment. Dans un premier temps, nous présenterons FaRM, un contrôleur de reconfiguration dynamique capable d'atteindre les limites théoriques de la technologie grâce à un algorithme de compression efficient et une architecture optimisée. Ensuite, nous présenterons RecoSim, un simulateur d'architectures reconfigurables en SystemC modélisant à un haut niveau d'abstraction un tel système. Basé sur un modèle de coût du temps de reconfiguration avec FaRM, RecoSim permet notamment le développement et l'évaluation d'algorithmes d'ordonnancement, qui sont des éléments clés des architectures temps-réel. Finalement, nous montrerons comment ces premières contributions sont utilisées au sein de FoRTReSS, un flot d'exploration d'architectures intégré avec les outils de développement Xilinx.

Ces travaux ont été effectués dans le cadre du projet ANR ARDMAHN.

Abstract

Despite promising capabilities, FPGAs partial reconfiguration feature is not anchored in the industry yet, mostly for two reasons. First of all, Xilinx controller shows low performance and might introduce a large time overhead compared to the task period, incompatible with the use of partial reconfiguration. Also, developing such a dynamic application requires an extra design effort compared to a static solution for developing scheduling strategies. Indeed, it is impossible to evaluate an architecture and/or a scheduling algorithm to verify that real-time constraints are met before the implementation step.

This thesis offers solutions to the issues previously mentioned. We will first introduce FaRM, a Fast Reconfiguration Manager reaching partial reconfiguration theoretical limits thanks to an efficient compression algorithm and an optimised architecture. Then, we present RecoSim, a high-level SystemC simulator for reconfigurable architectures. It makes use of FaRM reconfiguration overhead cost model to allow for developing and verifying scheduling algorithms. Finally, we will show how these contributions are used inside FoRTReSS, a Flow for Reconfigurable archiTectures in Real-time SystemS compliant with Xilinx partial reconfiguration design flow.

This work was carried out in the framework of project ARDMAHN, sponsored by the French National Research Agency.

Remerciements

Je tiens tout d'abord à remercier mes collègues de bureau, en particulier Clément Foucher, Sébastien Icart, Laurent Brossier, Umberto Cerasani et Yannick Vaiarello avec qui ces trois années sont passées très vite.

Merci à Alexandra Loisel et Jérémy Belarbi qui ont contribué respectivement au développement de FaRM et de l'outil graphique de FoRTReSS durant leur stage.

Je remercie également Philippe Lorenzini pour son suivi tout au long de la thèse et ses encouragements.

Enfin, je tiens à remercier tout particulièrement Fabrice Muller qui m'a aidé, soutenu et surtout guidé tout au long de la thèse. C'est une vraie chance de l'avoir en tant que co-directeur de thèse et je le remercie de m'avoir donné l'opportunité de faire cette thèse.

Table des figures

1.1 Comparaison des approches pour les systèmes embarqués 2
1.2 Passerelle résidentielle et réseau domestique 4

2.1 Architecture des FPGA 10
2.2 Architecture du contrôleur d'ICAP Xilinx [1] 13
2.3 Comparaison de différents contrôleurs de reconfiguration [2] 13
2.4 Comparaison d'algorithmes de compression appliqués à différentes IP [3] . . . 14
2.5 Comparaison d'algorithmes de compression appliqués à différentes IP [4] . . . 15
2.6 Architecture du contrôleur UPaRC [5] 16
2.7 Exemple de modélisation utilisant UML et MARTE [6] 18
2.8 La méthodologie MoPCoM [7] 19
2.9 Différents niveaux d'abstraction pour la modélisation SystemC [8] 19
2.10 Exemple de design utilisant ReChannel [9] 20
2.11 Processus de contrôle de ReChannel [9] 20
2.12 Contrôle de modules dynamiques [10] 22
2.13 Analyse structurelle de la reconfiguration dynamique [11] 24
2.14 Mesures de consommation d'UPaRC pendant la reconfiguration [5] 26
2.15 Représentation des unités reconfigurables RFUOP [12] 27
2.16 Regroupement de tâches dans une zone reconfigurable [13] 29
2.17 Influence du rapport hauteur/largeur des zones reconfigurables [14] 30

3.1 Architecture de l'IP FaRM 34
3.2 Sous-système de configuration AXI utilisant FaRM 35
3.3 Exemple de compression RLE 36
3.4 Principe de la compression O-RLE 37
3.5 Exemple de compression O-RLE 38
3.6 Modes de fonctionnement de FaRM 40
3.7 Adressage des frames de configuration [15] 41
3.8 Influence des tailles des accès rafales sur la reconfiguration 41
3.9 Exemple de trace obtenue avec Chipscope Pro 43

3.10 Trace VCD représentant une transaction sur le bus PLB 44
3.11 Architecture de l'application de test AES/FFT64 48
3.12 Influence de la taille de la FIFO sur le débit de l'ICAP 51
3.13 Nombre de mots écrits au débit maximal 52

4.1 Approche en Y de la modélisation avec RecoSim 56
4.2 Architecture d'un module reconfigurable 59
4.3 Diagramme de séquences pour la fin d'exécution d'un module 61
4.4 Automate fini pour les changements d'état des tâches 62
4.5 Gestion de la préemption au sein de RecoSim 64
4.6 Diagramme de séquence d'une transaction suivant le *Base protocol* [16] 65
4.7 Exemple de fonctionnement des canaux prioritaires 67
4.8 Application de transmissions vidéo sécurisées 68
4.9 Exemples de traces VCD 74
4.10 Schémas-blocs des cas d'utilisation ARDMAHN 76
4.11 Architecture du FPGA du démonstrateur ARDMAHN 78

5.1 L'approche FoRTReSS 82
5.2 Le flot FoRTReSS 83
5.3 Formes autorisées pour les zones reconfigurables 84
5.4 Principe du partitionnement des tâches avec FoRTReSS 87
5.5 Gestion des interfaces de la zone reconfigurable 90
5.6 Exemples d'architectures cibles 93
5.7 Interface graphique de FoRTReSS 96
5.8 Floorplan issu de FoRTReSS pour deux chaînes en parallèle 99

Liste des tableaux

2.1 Comparaison des principales approches de modélisation 24

3.1 Comparaison de taux de compression pour différents algorithmes 38
3.2 Ressources contraintes par les zones reconfigurables 48
3.3 Temps de reconfiguration et de relecture 49
3.4 Comparaison de FaRM avec l'HwICAP 53

4.1 Rapports de synthèse des tâches de l'application Secure Box 68
4.2 Taille des zones reconfigurables optimales pour un FPGA Virtex-6 70
4.3 Résultats en surface de l'application Secure Box pour un FPGA Virtex-6 . . . 71
4.4 Temps d'exécution des tâches sur les zones reconfigurables 72
4.5 Résultats en surface de l'application Secure Box après partitionnement 72
4.6 Temps d'exécution des tâches sur les zones reconfigurables après partitionnement 73
4.7 Cas d'utilisation du démonstrateur ARDMAHN 75
4.8 Rapports de synthèse des tâches de l'application ARDMAHN 76
4.9 Taille des zones reconfigurables optimales pour les tâches ARDMAHN 78
4.10 Résultats en surface de l'application ARDMAHN pour un FPGA Virtex-6 . . 78

5.1 Paramètres de la détermination des zones reconfigurables 84
5.2 Paramètres du tri des zones reconfigurables 85
5.3 Paramètres de la sélection des zones reconfigurables 88
5.4 Paramètres de la sélection des processeurs 88
5.5 Paramètres de l'allocation 90
5.6 Paramètres du modèle de coût 91
5.7 Paramètres de la simulation 92
5.8 Paramétrage de FoRTReSS pour l'application Secure Box 97
5.9 Résultats en surface issus de FoRTReSS pour un FPGA Virtex-6 LX240T . . . 98
5.10 Résultats en surface pour trois applications sur un FPGA Virtex-6 LX240T . . 101
5.11 Taux de compression réels (en pourcentage de la taille initiale) 102

Table des matières

1 Introduction **1**
- 1.1 Présentation des systèmes reconfigurables . . . 1
- 1.2 Le projet ARDMAHN . . . 3
- 1.3 Contributions . . . 5
  - 1.3.1 FaRM . . . 5
  - 1.3.2 Recosim . . . 5
  - 1.3.3 FoRTReSS . . . 6
- 1.4 Organisation du mémoire . . . 6

2 État de l'art **9**
- 2.1 Présentation des concepts de la reconfiguration dynamique . . . 9
  - 2.1.1 Architecture interne des FPGA . . . 9
  - 2.1.2 Contenu des fichiers de configuration . . . 10
  - 2.1.3 Sauvegarde de contexte et relecture . . . 11
  - 2.1.4 Flots de développement . . . 12
- 2.2 Optimisation des performances de la reconfiguration dynamique . . . 12
  - 2.2.1 Optimisation de l'architecture . . . 12
  - 2.2.2 Compression des bitstreams . . . 14
  - 2.2.3 Optimisation de l'ordonnancement . . . 16
- 2.3 Modélisation de la reconfiguration dynamique . . . 17
  - 2.3.1 Modélisation comportementale . . . 17
  - 2.3.2 Modèle de coût de la reconfiguration dynamique . . . 23
- 2.4 Exploration d'architectures reconfigurables . . . 26
- 2.5 Conclusion . . . 31

3 FaRM : un contrôleur de reconfiguration dynamique pour l'optimisation des performances **33**
- 3.1 Notre approche . . . 33
  - 3.1.1 Architecture de FaRM . . . 34

3.1.2 Compression des bitstreams 36
3.1.3 Modes de fonctionnement 39
3.2 Modèle de coût du temps de reconfiguration 42
3.2.1 Coût du mode d'écriture simple 42
3.2.2 Coût du mode de préchargement 45
3.2.3 Coût du mode de relecture 47
3.3 Application 47
3.3.1 Evaluation des performances 49
3.3.2 Overclocking de l'ICAP 50
3.3.3 Influence de la FIFO sur le préchargement des bitstreams 50
3.4 Positionnement de FaRM 53
3.5 Conclusion 54

4 RecoSim : un simulateur d'architectures reconfigurables dynamiquement pour des applications temps-réel 55
4.1 L'approche RecoSim 55
4.1.1 Approche en Y de la modélisation 56
4.1.2 Définition du module reconfigurable 58
4.1.3 Gestionnaire de reconfiguration 60
4.1.4 Communications et TLM 64
4.2 Applications 67
4.2.1 Application Secure Box 67
4.2.2 Application ARDMAHN 73
4.3 Perspectives 79
4.4 Conclusion 80

5 FoRTReSS : un flot d'exploration d'architectures reconfigurables pour des applications temps-réel 81
5.1 L'approche FoRTReSS 81
5.1.1 Le flot FoRTReSS 82
5.1.2 Architecture cible 92
5.1.3 Environnement graphique 95
5.2 Application 95
5.3 Résultats 97
5.3.1 Floorplanning et résultats de simulation 98
5.3.2 Exploration d'architectures pour trois applications 99
5.3.3 Validation finale 101
5.3.4 Temps d'exécution de FoRTReSS 102
5.4 Perspectives 103
5.5 Conclusion 104

6 Conclusion générale et perspectives 107

Bibliographie 111

CHAPITRE 1

Introduction

1.1 Présentation des systèmes reconfigurables

Au cours des dix dernières années, les systèmes embarqués ont connu un développement fulgurant au point qu'ils font aujourd'hui parti de notre vie quotidienne : téléphones portables, tablettes numériques, ordinateurs de bord, assistance médicale... Les applications ayant recours à ce type de systèmes sont nombreuses et ont besoin de toujours plus de puissance de calcul. Les circuits développés sont également de plus en plus complets et tendent à embarquer un système complet composé d'un ou plusieurs processeurs, de mémoire, et éventuellement d'accélérateurs matériels. On parle alors de système sur puce (en anglais *System-on-Chip*, SoC).

En particulier, le calcul reconfigurable a connu un essor considérable, faisant du FPGA (*Field Programmable Gate Array*) un composant essentiel des systèmes embarqués. En effet, le FPGA offre un bon compromis entre une solution logicielle, flexible et lente à cause de son exécution temporelle, et une solution matérielle à base d'ASIC (*Application Specific Integrated Circuit*, un circuit intégré spécialisé), très rapide grâce à une exécution spatiale et parallèle mais peu flexible (voir figure 1.1). Malgré un coût en grande série très supérieur à celui de son équivalent ASIC, un système basé sur un FPGA aura toutefois un temps de mise sur le marché bien inférieur, ce qui en fait un parfait candidat pour le prototypage rapide. Lorsqu'un système complet est implanté sur un composant programmable, on parle alors de système sur puce reprogrammable, ou en anglais SoPC (*System on Programmable Chip*).

Toutefois, les FPGA classiques ne peuvent pas modifier leur comportement en cours d'exécution. Prenons l'exemple d'un système composé d'un accélérateur matériel disponible en trois versions plus ou moins efficaces qui sont activées individuellement selon les performances requises par le système. Même si un seul accélérateur est actif à la fois, les trois doivent être présents sur le FPGA à tout instant. Partant de ce constat, Xilinx a développé le concept de reconfiguration dynamique partielle (dans le manuscrit, nous utiliserons indifféremment le terme reconfiguration dynamique).

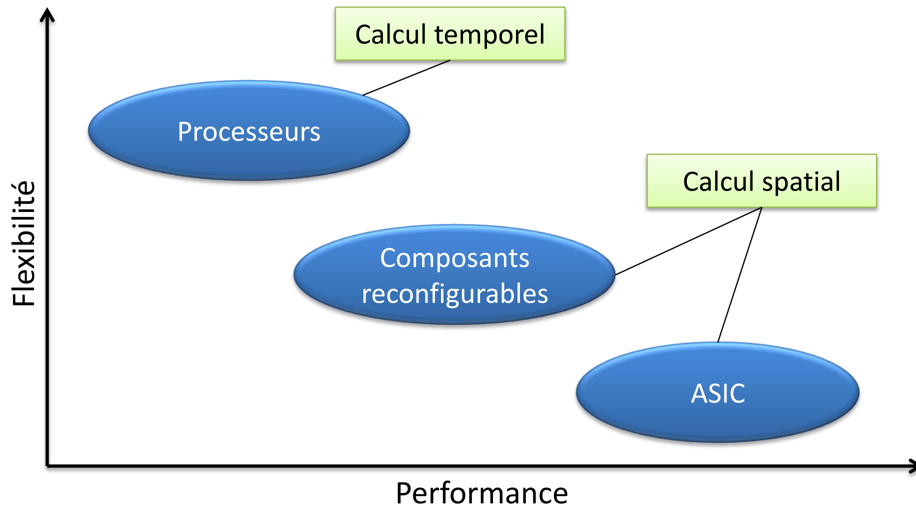


Figure 1.1 – *Comparaison des approches pour les systèmes embarqués*

Introduite à la fin des années 1990 avec la série de FPGA XC6200, la reconfiguration dynamique permet la modification d'une partie prédéfinie du FPGA pendant que le reste du design s'exécute normalement. Ainsi, il est possible de changer les fonctionnalités du système en cours d'exécution, permettant de réduire la surface nécessaire et/ou la consommation d'énergie du FPGA [17, 18]. En reprenant l'exemple précédent, il serait alors possible de n'avoir qu'un seul accélérateur instancié sur le FPGA à chaque instant.

En dépit de caractéristiques intéressantes, la reconfiguration dynamique reste n'est toujours pas ancrée dans l'industrie [19]. Nous y voyons deux raisons principales. La première raison est un problème de performances pour la reconfiguration dynamique. En effet, le contrôleur de reconfiguration fourni par Xilinx ne permet pas d'atteindre des débits intéressants. Si l'on considère que le système doit ordonnancer des tâches matérielles, le temps de reconfiguration est crucial et si celui-ci est trop grand devant le temps d'exécution de la tâche au point d'en devenir inacceptable, le concepteur privilégiera une solution statique classique [20, 21].

La seconde raison de l'absence de la reconfiguration dynamique dans l'industrie aujourd'hui est selon nous la difficulté de conception de telles architectures. Tout d'abord, il est impossible de simuler un système avec des outils classiques comme par exemple ModelSim, qui ne permet pas de modifier les modules en cours de simulation. La validation s'effectue donc directement sur carte, pendant la dernière phase du processus de conception : l'implémentation. L'identification d'un problème lors des dernières phases de conception d'un système est très coûteuse, notamment en temps de développement pour le résoudre. C'est pour cela que durant ces dernières années, la tendance est à l'élévation du niveau d'abstraction lors de la conception afin d'identifier de potentiels problèmes au plus tôt dans le cycle de développement. Enfin, la définition de l'architecture du système reconfigurable est un problème très vaste qu'il faut aujourd'hui résoudre en explorant manuellement l'espace de conception, qui grandit exponentiellement avec le nombre de tâches considérées.

Pour ces raisons, la reconfiguration dynamique des FPGA reste un thème de recherche majeur qu'il est compliqué d'intégrer à une approche industrielle de la conception de systèmes. Le

projet ARDMAHN (Architectures Reconfigurables Dynamiquement pour l'Auto-adaptation en Home Networking) [22], financé par l'Agence Nationale de la Recherche (ANR) a pour objectif de répondre à cette problématique.

1.2 Le projet ARDMAHN

Ce projet se place dans le contexte des applications multimédia qui prennent une part importante dans le quotidien des utilisateurs, notamment pour la diffusion de flux. Mais, avec l'explosion des terminaux et des types de réseaux d'accès, l'hétérogénéité devient aussi omniprésente. Malheureusement, pour le moment, les éléments de la chaîne de distribution de services multimédia n'intègrent pas cette nouvelle contrainte. La gestion et le traitement du flux multimédia doivent être réalisés tout au long de la chaîne de distribution : de sa génération dans les "Têtes de réseaux", pendant son acheminement à travers les réseaux de distribution et jusqu'aux utilisateurs finaux. L'adaptation dynamique et transparente de ce flux multimédia selon les caractéristiques du ou des consommateurs demandeurs et de leur(s) environnement(s) (hétérogènes) est la clé du succès des services réseaux de demain.

L'adaptation d'un flux multimédia aux caractéristiques des réseaux de distribution et aux besoins de l'utilisateur est un processus complexe demandant d'importantes ressources de calcul. Cette adaptation repose sur des techniques de transcodage, transrating, transizing, ajout de données, etc. La complexité de ces techniques, d'une part, et les contraintes sur les temps de réponses, d'autre part, génèrent des problèmes d'implémentation, d'intégration et de validation très difficiles à maîtriser. Une nouvelle approche des phases de définition, spécification et implémentation s'impose. Dans ce projet, nous résoudrons ces problèmes de performances et de timing en transférant une partie des fonctions d'adaptation du domaine logiciel vers le domaine matériel, principalement via des technologies de type FPGA.

Dans ce domaine, si l'adaptabilité à des évènements connus peut être maîtrisée sur des fonctions purement logicielles, elle est quasi inexistante sur des fonctions matérielles. Or, les architectures matérielles auto-adaptatives sont un enjeu essentiel pour les prochaines générations d'équipements de diffusion et d'exploitation de services. En effet, la vitesse d'exécution des fonctions et l'énergie consommée sont deux facteurs particulièrement limitant et de surcroît antagonistes dans les applications embarquées. Le coût et l'encombrement du système de refroidissement sont parfois un obstacle aux développements. Bien gérée, la reconfiguration dynamique peut permettre d'allier performances, faible consommation et adaptabilité en utilisant les technologies récentes. Même si elle est perfectible, la technologie est disponible, en revanche la mise en œuvre optimale, sa vérification et validation architecturale restent à imaginer.

L'innovation que nous proposons porte donc sur trois aspects intimement liés et qui visent la gestion et l'adaptation optimale du flux multimédia :

- La définition d'une architecture reconfigurable dynamiquement répondant à des exigences de performances et d'adaptabilité de flux multimédias aux besoins des utilisateurs (préférences, terminaux, contenus, ...) et en fonction des caractéristiques des réseaux de distribution.

- L'adaptation et la validation des fonctions logicielles d'un système existant pour migrer vers une architecture reconfigurable dynamiquement.
- La définition d'une méthodologie permettant de vérifier et de valider les architectures à reconfiguration dynamique pour faciliter le développement aisé des applications.

Le projet réalisera et validera la fonction d'adaptation intégrée dans une passerelle résidentielle mais la même fonction peut aussi bien être intégrée dans la "Tête de réseau" ou dans un nœud du réseau de distribution. Dans le contexte des réseaux à domicile (Home Network), les passerelles résidentielles (Home Gateway) vont être amenées dans l'avenir à gérer de multiples flux d'information d'entrée sur des technologies de réseaux variées (3G/3G+/LTE, DVB-T/-S, xDSL, WiFi, WiMax, bluetooth, technologies de capteurs, etc.) vers de multiples usages sur terminaux hétérogènes (TV, PDA, PC, Phone, etc.) en les adaptant de manière dynamique et transparente selon la demande du ou des consommateurs au sein de la maison (figure 1.2).

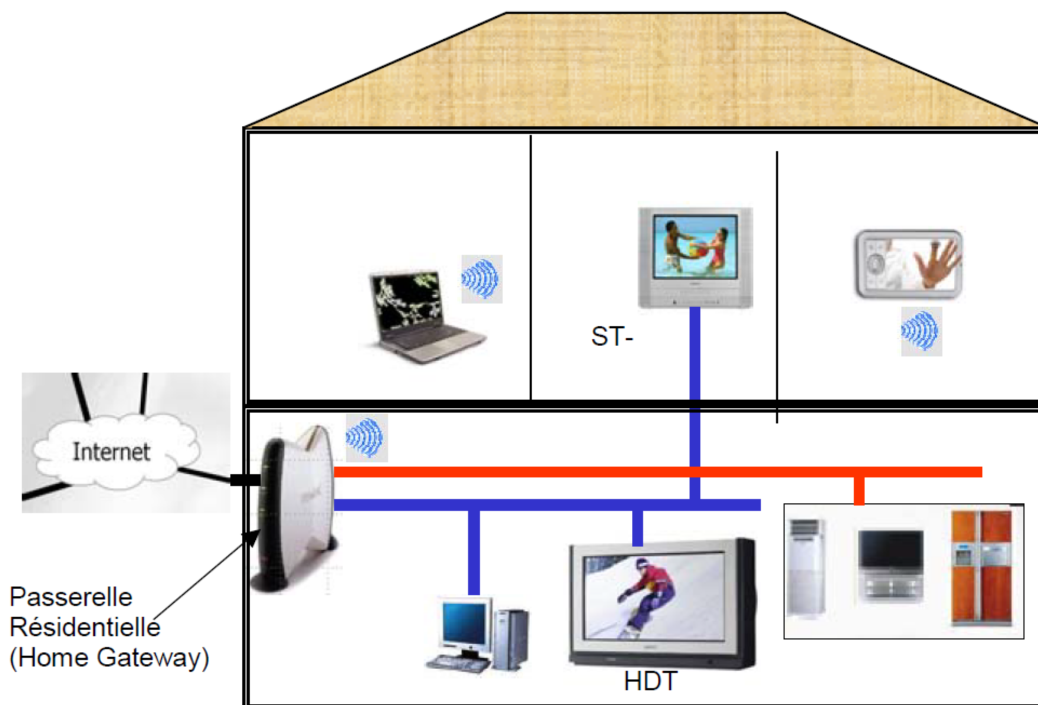


Figure 1.2 – *Passerelle résidentielle et réseau domestique*

Sur le plan de la recherche, le projet porte donc sur des aspects d'architecture matérielle et logicielle, au travers d'un système embarqué, capable de répondre aux défis doubles de performances et d'adaptation dynamique à des contextes et des besoins différents. Il mettra en œuvre des solutions dédiées reconfigurables et Multiprocesseurs Programmables sur puces (MPSoC). L'objectif est également de concevoir des modèles de validation de la reconfiguration dynamique afin de simuler le comportement de ce type de conception. L'application sur la passerelle résidentielle (Home Gateway) revêt un caractère industriel, à vocation de démonstration.

1.3 Contributions

Nous présentons ici nos trois contributions proposant une approche pour la conception de systèmes reconfigurables dynamiquement. Ainsi, nous proposons un contrôleur de reconfiguration pour l'optimisation des performances de la reconfiguration dynamique, un simulateur adapté aux problématiques des systèmes reconfigurables dynamiquement ainsi qu'un flot d'exploration et de validation de telles architectures. Un outil graphique permet de réunir toutes ces contributions.

1.3.1 FaRM

Notre première contribution est FaRM (pour *Fast Reconfiguration Manager*), un contrôleur de reconfiguration dynamique dont les performances ont été optimisées afin de rendre le processus le plus transparent possible par rapport à l'exécution des tâches matérielles. L'architecture du contrôleur est basée sur une interface maître/esclave, respectivement pour la récupération autonome des bitstreams et pour la configuration de FaRM, ainsi que sur une FIFO permettant le préchargement des bitstreams de configuration en mémoire. Ces bitstreams peuvent être compressés afin de réduire l'empreinte mémoire d'une solution et surtout réduire les temps de transferts depuis la mémoire de stockage vers le contrôleur. L'algorithme de compression utilisé est dérivé du *Run Length Encoding* (RLE) qui assure une compression efficace tout en conservant un débit d'un mot par cycle d'horloge lors de la décompression (qui est faite en ligne). Enfin, FaRM permet de relire les données de configuration du FPGA qui peuvent être utilisées pour la sauvegarde de contexte, par exemple.

La connaissance du temps de reconfiguration étant cruciale pour modéliser précisément le comportement d'un système reconfigurable dynamiquement, nous avons également déterminé un modèle de coût pour FaRM qui permet d'estimer très précisément le temps de reconfiguration nécessaire en fonction du bitstream considéré, du taux de compression réalisé et de certaines caractéristiques du sous-système de configuration (par exemple la fréquence de fonctionnement du contrôleur de reconfiguration ou du bus de données).

1.3.2 Recosim

RecoSim, pour *Reconfiguration Simulator*, constitue notre seconde contribution. Ce simulateur d'architectures reconfigurables dynamiquement écrit en SystemC permet de vérifier la faisabilité d'une solution pour des applications ayant des contraintes temps-réel. Pour cela, RecoSim essaye de placer les tâches de l'application, décrites à l'aide d'un diagramme orienté acyclique (en anglais DAG, *Directed Acyclic Graph*), sur les zones reconfigurables définies par le concepteur en se conformant à la politique d'ordonnancement utilisée. Le simulateur prend en compte les temps d'exécution des tâches ainsi que les temps de reconfiguration qui peuvent être estimés au moyen du modèle de coût de FaRM.

RecoSim utilise les threads dynamiques afin de permettre la modification du comportement d'une tâche en cours de simulation et ainsi modifier le niveau de précision de l'algorithme simulé (par exemple passer d'un code SystemC à un code VHDL). RecoSim utilise également la modélisation transactionnelle (TLM, pour *Transaction-Level Modeling*) qui per-

met de s'abstraire des détails d'implémentation lors de la modélisation des communications entre modules. Le simulateur peut ainsi s'adapter très facilement à différentes architectures cibles à peu de frais. L'utilisation de TLM permet également de réduire drastiquement le temps de simulation par rapport à une simulation au niveau RT (*Register Transfer*).

1.3.3 FoRTReSS

Enfin, notre dernière contribution s'intitule FoRTReSS, pour *Flow for Reconfigurable archiTectures in Real-time SystemS*, une méthodologie hautement paramétrable de conception d'architectures reconfigurables dynamiquement ayant des contraintes temps-réel qui est entièrement intégrée dans les flots de conception existants comme celui de Xilinx. Basé sur FaRM et RecoSim, FoRTReSS utilise les résultats de la synthèse des tâches reconfigurables pour inférer un ensemble de zones reconfigurables compatibles avec l'application. Les meilleures zones sont alors sélectionnées pour être simulées en fonction de critères comme la fragmentation externe ou la forme de la zone reconfigurable.

FoRTReSS cherche dans un premier temps à déterminer le nombre de zones reconfigurables nécessaires au système pour satisfaire les contraintes de temps de l'application. La première simulation est lancée avec une zone reconfigurable. Le nombre de zones lors de la simulation est incrémenté jusqu'à obtenir une première solution viable. Dans un second temps, FoRTReSS essaye d'améliorer cette solution en optimisant l'allocation des tâches sur les zones reconfigurables disponibles. L'application est également partitionnée en fonction des ressources nécessaires à chaque tâche afin d'obtenir une solution dynamique qui soit plus intéressante qu'une solution statique en termes de ressources consommées. Une fois la solution optimisée, FoRTReSS génère un fichier de contraintes utilisable par les outils d'implémentation comme Xilinx PlanAhead.

1.4 Organisation du mémoire

Le mémoire est organisé de la manière suivante : nous effectuerons tout d'abord une analyse de l'existant sur les architectures reconfigurables dynamiquement dans le chapitre suivant. Nous traiterons à la fois des performances de la reconfiguration dynamique ainsi que des approches utilisées pour la modélisation et l'exploration d'architectures reconfigurables dynamiquement. Nous étudierons particulièrement les problèmes de placement des zones reconfigurables lors de la phase de conception du système. Dans le chapitre 3, nous présenterons notre première contribution, FaRM, une IP qui maximise les performances de la reconfiguration dynamique. Nous présenterons l'architecture de ce contrôleur ainsi que ses différents modes de fonctionnement. Ensuite, nous décrirons RecoSim, notre simulateur d'architectures reconfigurables dynamiquement dans le chapitre 4. Nous montrerons que l'utilisation conjointe des threads dynamiques et de la modélisation transactionnelle permet de modéliser à un haut niveau d'abstraction un système reconfigurable afin d'estimer la faisabilité d'une telle solution. Pour compléter notre approche, le chapitre 5 présentera FoRTReSS, notre flot de conception d'architectures reconfigurables dynamiquement, ainsi que les différents para-

mètres pris en compte pour la génération de l'architecture, spécifiée à partir de l'application ciblée. Enfin, les conclusions et perspectives de ces travaux seront énoncées dans le chapitre 6.

CHAPITRE 2

État de l'art

La problématique de la thèse définit clairement deux axes majeurs. Tout d'abord, l'aspect matériel et bas-niveau de l'étude concernant l'optimisation des performances de la reconfiguration dynamique. Ensuite, l'aspect modélisation du mécanisme de reconfiguration dynamique et par extension des architectures reconfigurables dynamiquement afin de permettre de faciliter le développement de telles solutions. Dans ce chapitre, nous présenterons tout d'abord quelques concepts sur la reconfiguration dynamique. Ensuite, nous développerons l'état de l'art concernant l'optimisation des performances et la modélisation de la reconfiguration dynamique. Enfin, nous parlerons des travaux menés sur l'exploration d'architectures reconfigurables pour des applications auto-adaptatives.

2.1 Présentation des concepts de la reconfiguration dynamique

Dans la suite du manuscrit, nous décrirons principalement les FPGA Xilinx puisqu'ils étaient les seuls capables de reconfiguration dynamique fin 2008. Depuis, Altera s'est également engagé dans cette technologie [23]. Toutefois, les concepts énoncés ici sont parfaitement adaptés à d'autres FPGA partageant une architecture similaire. Nous nous intéresserons dans un premier temps à l'architecture interne des FPGA ainsi qu'à la mise en œuvre d'une architecture reconfigurable dynamiquement.

2.1.1 Architecture interne des FPGA

La figure 2.1 représente l'architecture de type matricielle des FPGA actuels. Les ressources disponibles sont hétérogènes, avec par exemple des cellules logiques configurables (CLB, pour *Configurable Logic Block*), de la mémoire (BRAM, pour *Block Random-Access Memory*) ainsi que des processeurs de traitement de signal (DSP, pour *Digital Signal Pro-*

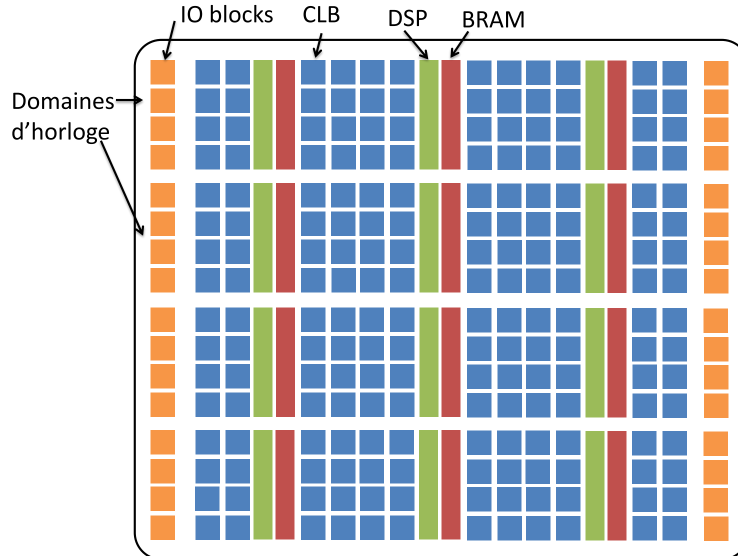


Figure 2.1 – *Architecture des FPGA*

cessors). Ces ressources sont arrangées en colonnes dont la hauteur représente un domaine d'horloge du FPGA. La configuration des ressources et de leur routage est contenue dans la mémoire de configuration. C'est donc cette mémoire qui est modifiée lors de la reconfiguration dynamique. Parmi les technologies utilisées pour cette mémoire, on trouve aujourd'hui la technologie anti-fusible (programmable une fois) et la technologie SRAM (*Standard Random-Access Memory*). Cette dernière utilise une mémoire volatile pour la configuration du FPGA, à la fois programmable et re-programmable *in situ*. Pour conserver la configuration en dehors des mises sous tension, il est nécessaire de coupler cette SRAM avec une mémoire non-volatile externe, typiquement une mémoire Flash.

Il existe plusieurs moyens d'accéder à la mémoire de configuration. On peut distinguer deux classes d'interfaces [15] :

- Les interfaces externes au FPGA comme le SelectMap ou le JTAG, utilisées notamment pour la configuration "classique" du FPGA. On parle alors d'exo-reconfiguration.
- Les interfaces internes au FPGA comme l'ICAP (*Internal Configuration Access Port*). La reconfiguration est alors possible depuis l'intérieur du FPGA et permet de bâtir un système autonome : on parle alors d'endo-reconfiguration.

2.1.2 Contenu des fichiers de configuration

La reconfiguration dynamique est effectuée en écrivant les données de configuration via l'interface interne ICAP. Ces données sont contenues dans un fichier de configuration appelé *bitstream* qui est généré par les outils de développement Xilinx. En particulier, dans le cas de la reconfiguration dynamique, les bitstreams sont générés par l'outil de floorplanning PlanAhead. Un bitstream est composé de deux parties :

- Un en-tête contenant des informations sur le design dont est issu le bitstream ainsi que des informations d'initialisation et de synchronisation. Cet en-tête ne consiste qu'en une cinquantaine de mots de 32 bits.

- Un corps qui contient les données de configuration du FPGA ou de la zone reconfigurable. La taille du corps est variable et dépend du nombre et du type de ressources contraintes par la zone reconfigurable.

Pour chaque type de ressource, la taille des données de configuration diffère. En effet, la mémoire de configuration est découpée en *frames* qui sont les plus petits éléments reconfigurables, de même hauteur qu'un domaine d'horloge et de taille fixe (41 mots de 32 bits pour la technologie Virtex-5 [15]) et chaque type de ressources requiert un nombre différent de frames pour la configuration. Ainsi, pour un FPGA Virtex-5, une colonne de CLB nécessite 36 frames de configuration contre 28 frames pour les DSP48 et 158 frames pour les BRAM. L'utilisation d'une ou plusieurs BRAM dans une zone reconfigurable introduit donc un coût supplémentaire en termes de stockage et de temps de reconfiguration. Toutefois, parmi les 158 frames de configuration, 128 sont dédiées au contenu de la mémoire, souvent initialisée à zéro. L'utilisation d'algorithmes de compression sans pertes sur ces bitstreams peut donc s'avérer très intéressante. Enfin, la taille des domaines d'horloge varie également d'une famille de FPGA à l'autre. Pour les FPGA de la famille Virtex-5, une colonne de CLB contient 20 CLB alors qu'une colonne en contient 40 pour les FPGA Virtex-6.

2.1.3 Sauvegarde de contexte et relecture

Les tâches matérielles ont un comportement similaire aux tâches logicielles. Ainsi, lorsqu'une tâche est préemptée en cours d'exécution au bénéfice d'une autre tâche, il est souvent nécessaire de sauvegarder le contexte afin de pouvoir le restaurer lorsque la tâche aura de nouveau le droit de s'exécuter. Pour des tâches logicielles, il faut par exemple sauvegarder l'état de la tâche ainsi que les variables utilisées. Pour les tâches matérielles, la gestion du contexte est plus compliquée. Deux solutions sont envisageables :

- Soit le concepteur a prévu ce cas lors du développement de l'IP, et il suffit alors de lire un ensemble de registres de mémoires prévus à cet effet.
- Soit la tâche n'a pas été prévu pour être préemptée, auquel cas il faut utiliser la capacité de relecture des FPGA Xilinx.

La capacité de relecture (en anglais *readback*) des FPGA Xilinx consiste à avoir accès en lecture à la mémoire de configuration du FPGA. Il est ainsi possible d'accéder à toutes les données de configuration du FPGA et donc en particulier aux données de configuration correspondant aux zones reconfigurables. En pratique, ces données sont effectivement accessibles mais ne représentent pas vraiment l'état courant du FPGA. Par exemple, le contenu des mémoires BRAM ou LUTRAM (LUT utilisée comme mémoire) n'est pas représentatif du contenu lors de la relecture. De plus, cette méthode nécessite de nombreux accès à la mémoire de configuration puis à une mémoire de stockage relativement importante, représentant également une pénalité temporelle. En effet, les données de configuration font sensiblement la même taille que les bitstreams de configuration, ce qui augmente énormément les besoins en mémoire du système.

En conclusion, nous pouvons donc dire que même si la fonctionnalité de relecture parait intéressante à première vue, celle-ci n'est pas encore tout à fait au point pour la technologie

actuelle (Virtex-5 et Virtex-6) et elle est de toute manière difficilement utilisable dans le cas de systèmes ayant de fortes contraintes de temps.

2.1.4 Flots de développement

Le but de cette thèse n'est pas de redéfinir complètement des outils, très complexes, de développement d'architectures reconfigurables mais bien de venir compléter les approches existantes. Dans ce mémoire, nous parlerons essentiellement du flot de développement de Xilinx qui a été utilisé tout au long de la thèse. Des approches alternatives sont sorties depuis, notamment OpenPR [24], le flot open-source de Virginia Tech basé sur le flot Xilinx.

Le choix du placement et du nombre de ressources contraintes par une zone reconfigurable est effectué durant la phase de floorplanning sous PlanAhead. Les zones reconfigurables doivent être rectangulaires et leur hauteur doit être un multiple de la taille d'un domaine d'horloge du FPGA. Dans le cas inverse, le bitstream généré contiendra également les données de configuration relatives à la partie non contrainte dans le domaine d'horloge, résultant en un bitstream plus important que nécessaire.

Certaines ressources ne peuvent pas être contraintes dans une zone reconfigurable. C'est par exemple le cas des buffers d'horloge ou des boucles à verrouillage de phase (PLL, *Phase Lock Loops*). Si une zone reconfigurable inclus de telles ressources, elles seront conservées statiques.

Une fois les zones reconfigurables définies, la phase d'implémentation peut commencer. Dans un design reconfigurable, une implémentation représente les couples IP/zone reconfigurable choisis. Par exemple, on peut choisir d'implémenter l'IP 1 sur la zone 1 et l'IP 2 sur la zone 2. Dans ce cas, trois bitstreams seront générés : un bitstream complet implémentant les IP 1 et 2 ainsi que deux bitstreams partiels, un pour chaque zone reconfigurable. Il y a donc potentiellement autant de bitstreams partiels que de paires couples IP/zone reconfigurable possibles dans le design.

2.2 Optimisation des performances de la reconfiguration dynamique

Dans cette partie, nous nous intéressons aux différents moyens utilisés pour améliorer les performances de la reconfiguration dynamique. Nous verrons d'abord comment l'architecture du contrôleur peut-être optimisée, puis comment la compression des bitstreams réduit le temps de reconfiguration et enfin l'impact de l'ordonnancement.

2.2.1 Optimisation de l'architecture

La figure 2.2 présente l'architecture du contrôleur d'ICAP de Xilinx [1], dans sa version AXI, le nouveau standard de bus ARM [25]. On remarque que le contrôleur est simplement un périphérique esclave qui doit être commandé par un microprocesseur, par exemple un MicroBlaze ou un PowerPC. Les données sont alors stockées dans une FIFO contrôlée par une machine à états finis. La fréquence d'horloge recommandée pour l'ICAP étant 100 MHz

avec un bus de 32 bits de données, le débit théorique de cette macro est de 400 Mo/s. Le fait de devoir transférer les données une à une via des écritures de registres sur le bus constitue un premier goulot d'étranglement au niveau des performances ce qui mène à des performances très éloignées du débit théorique.

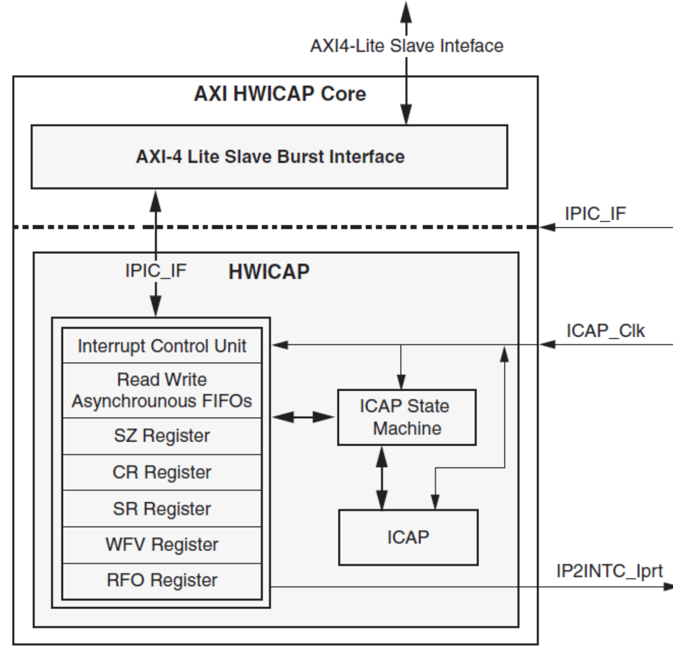


FIGURE 2.2 – *Architecture du contrôleur d'ICAP Xilinx [1]*

Les auteurs de [26] présentent un dépôt de bitstreams hiérarchique fonctionnant sur le principe de la mémoire cache. Au niveau le plus proche de l'ICAP, lorsque les bitstreams sont stockés localement, les auteurs ont développé une architecture basée sur DMA (*Direct Memory Access*, en français accès direct à la mémoire) couplé à une RAM afin de répondre au problème mentionné ci-dessus. Une approche similaire est proposée en [2], où les auteurs comparent différentes architectures, utilisant ou non le cache du processeur ou un DMA (voir figure 2.3).

ICAP design	Test 1 (Bit. Size/Reconf. Time)	Test 2 (Bit. Size/Reconf. Time)	Test 3 (Bit. Size/Reconf. Time)	Test 4 (Bit. Size/Reconf. Time)	Avg. reconfig. speed	Max. reconfig. speed
OPB_HWICAP (PowerPC cache_disabled)	7.7 KB/12.1 ms	23.2 KB/36.5 ms	44.5 KB/75.6 ms	79.9 KB/135.6 ms	0.61 MB/s	0.64 MB/s
XPS_HWICAP (PowerPC cache_disabled)	7.7 KB/9.2 ms	23.2 KB/27.9 ms	46.5 KB/57.9 ms	80.0 KB/99.7 ms	0.82 MB/s	0.84 MB/s
OPB_HWICAP (PowerPC cache_enabled)	7.7 KB/694.8 μs	22.7 KB/2.3 ms	43.9 KB/4.5 ms	75.9 KB/7.8 ms	10.1 MB/s	11.1 MB/s
XPS_HWICAP (PowerPC cache_enabled)	7.7 KB/336.9 μs	23.2 KB/1.3 ms	44.5 KB/2.5 ms	74.6 KB/4.2 ms	19.1 MB/s	22.9 MB/s
OPB_HWICAP (Microblaze cache_enabled)	7.7 KB/1.3 ms	23.2 KB/3.9 ms	47.1 KB/7.9 ms	77.7 KB/13.0 ms	6.0 MB/s	6.0 MB/s
XPS_HWICAP (Microblaze cache_enabled)	7.7 KB/532.6 μs	23.2 KB/1.6 ms	47.2 KB/3.3 ms	79.1 KB/5.4 ms	14.5 MB/s	14.6 MB/s
DMA_HWICAP	7.7 KB/95.1 μs	23.2 KB/282.3 μs	46.8 KB/566.3 μs	81.9 KB/991.1 μs	82.1 MB/s	82.6 MB/s
MST_HWICAP	7.7 KB/33.0 μs	23.2 KB/98.9 μs	44.8 KB/190.7 μs	76.0 KB/323.1 μs	234.5 MB/s	235.2 MB/s
BRAM_HWICAP	7.7 KB/28.0 μs	23.2 KB/66.3 μs	45.2 KB/121.7 μs	none	332.1 MB/s	371.4 MB/s

FIGURE 2.3 – *Comparaison de différents contrôleurs de reconfiguration [2]*

Les auteurs ont également développé une version maître du contrôleur d'ICAP, qui est

alors indépendant du processeur pour les transferts de données. Cette version est la plus efficace en termes de transfert de données à l'exception de la version BRAM_HWICAP, atteignant presque le débit théorique de l'ICAP de 400 Mo/s qui utilise une BRAM pour stocker intégralement le bitstream. Cette version atteint ses limites avec le test 4 qui demande le transfert d'un bitstream d'environ 80 ko qu'il n'a pas été possible de stocker intégralement en BRAM sur un FPGA Virtex-4 FX20. Même si les FPGA embarquent de plus en plus de mémoire, la granularité de reconfiguration tend à grandir également et stocker l'intégralité d'un bitstream n'est donc pas la solution la plus sobre pour un contrôleur de reconfiguration.

Les mêmes auteurs présentent dans [27] leur utilisation de configurations virtuelles pour réduire le temps de reconfiguration. Cette méthode consiste à implémenter plusieurs contextes : un contexte actif contenant les données de configuration courantes du FPGA et un ou plusieurs contextes latents qui peuvent être modifiés à volonté sans interrompre le fonctionnement du contexte actif. Cette solution est parfaitement adaptée aux FPGA multi-contextes, qui bénéficient nativement de contextes de configuration séparés [28, 29, 30]. Néanmoins, dans le cas de FPGA mono-contexte comme c'est le cas des FPGA Xilinx, l'implémentation nécessite de mémoriser autant de données de configuration qu'il y a de contextes latents, ce qui peut rapidement faire exploser le coût mémoire de la solution.

2.2.2 Compression des bitstreams

Comme précisé dans le préambule, compresser les bitstreams permet d'une part de diminuer les besoins en stockage mémoire du système mais également de réduire la taille des données à transférer à l'ICAP. Les auteurs à l'origine de DAGGER [3] comparent différents algorithmes de compression reconnus comme ZIP, GNU Zip (GZIP) et RLE (*Run Length Encoding*, en français codage par plages) à leur propre algorithme basé sur RLE et Lempel-Ziv-Welch (LZW) (voir la comparaison des tailles de bitstream en octets obtenus après compression en figure 2.4).

Benchmark	Compression Algorithms					
	DAGGER	RLE	ZIP	LZW	GZIP	BZIP2
adder5and2bit	2025	12485	754	2821	625	427
fft_256_points	2784	14089	2278	3187	2149	1604
mux4	1857	11975	668	2737	539	328
mux32	2752	13133	2065	3063	1936	1430
mux48	3154	14290	2728	3222	2599	1973
Subtractor_4bit	1771	12142	759	2682	630	416
b01	1894	12169	1037	2706	908	587
b02	1713	11955	713	2640	584	336
b03	2631	13625	2083	3088	1954	1426
b04	4356	18632	5146	3975	5017	3853
b06	2106	12551	921	2835	792	586
b07	3609	16212	3382	3589	3253	2484
b09	2490	13318	1904	2977	1775	1266
b10	3082	14608	2741	3322	2612	1941
b11	4204	17771	4932	3852	4803	3710
b13	3283	14749	3045	3372	2913	2202

Figure 2.4 – *Comparaison d'algorithmes de compression appliqués à différentes IP [3]*

Dans cette étude, les auteurs donnent uniquement les tailles des bitstreams après compression et non les tailles des fichiers originaux. On ne peut donc pas connaitre le taux de compression réalisé par leur algorithme. Toutefois, on peut le comparer aux autres algorithmes évalués : celui-ci semble donc être meilleur que le RLE (qui donne étonnamment de très mauvais résultats) mais donne des résultats très éloignés des algorithmes comme ZIP. Il faut toutefois garder à l'esprit que même si le taux de compression s'avère excellent (réduisant ainsi les transferts depuis la mémoire), il faut pouvoir garantie une chaîne de décompression efficace puisque la décompression doit être effectuée en ligne pour envoyer les données décompressées à l'ICAP. Les algorithmes comme ZIP perdent donc de leur intérêt. Il faut également noter que ces bitstreams programment une couche virtuelle du FPGA et peuvent donc ne pas être représentatifs des résultats qui seraient obtenus pour des bitstreams natifs.

Les travaux présentés en [4] sont plus exhaustifs vis-à-vis de l'évaluation des algorithmes de compression. En effet, les auteurs n'ont pas uniquement observé le taux de compression obtenu mais aussi des métriques comme le débit obtenu et la quantité de ressources nécessaires à la décompression.

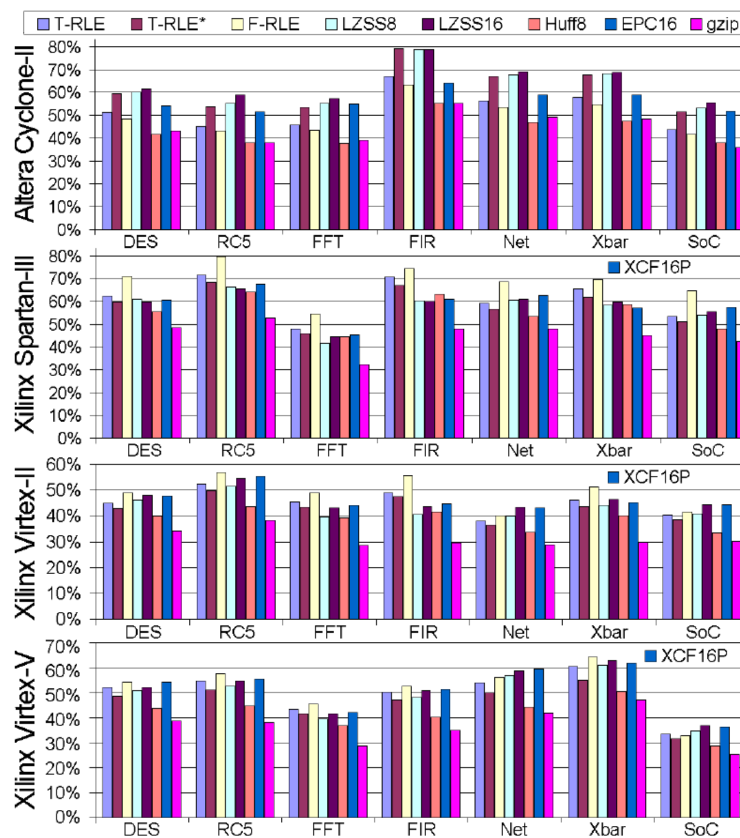


FIGURE 2.5 – *Comparaison d'algorithmes de compression appliqués à différentes IP [4]*

Les auteurs de [5] présentent UPaRC (*Ultra-Fast Power-aware Reconfiguration Controller*, un contrôleur de reconfiguration ultra-rapide utilisant à la fois une architecture optimisée à base de BRAM et un algorithme de compression de bitstreams (voir figure 2.6). Utilisant une solution très similaire à la nôtre, les auteurs vont toutefois plus loin dans l'optimisation des performances. Ils utilisent ainsi un algorithme de compression open source très efficace, X-MatchPRO [31], réalisant des taux de compression de près de 75% en moyenne. Toutefois,

ces performances se font à l'encontre de la complexité de l'algorithme, dont la partie décompression doit être implantée sur le FPGA nécessite un nombre non négligeable de ressources (environ 1000 slices) pour obtenir un débit correct (1 Go/s, soit un mot de 32 bits par cycle à 250 MHz). Concernant la partie architecture, les auteurs ont réussi à élever la fréquence d'horloge de l'ICAP, initialement prévue pour fonctionner à 100 MHz, jusqu'à 362.5 MHz ce qui leur permet d'annoncer un débit maximal de 1433 Mo/s pendant la reconfiguration.

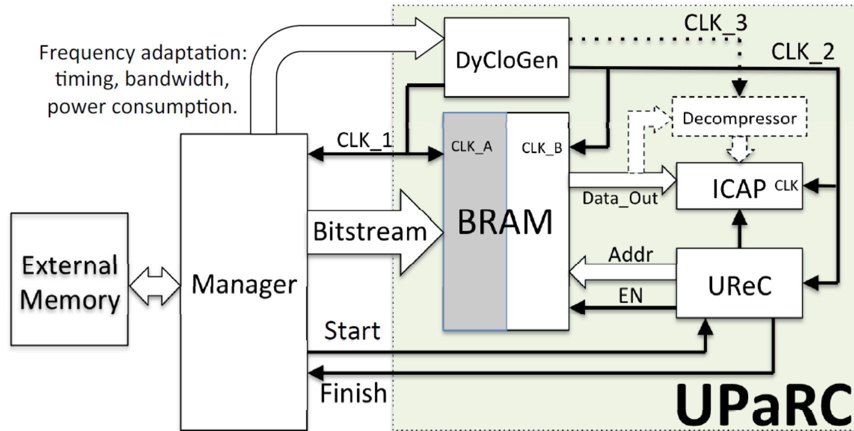


FIGURE 2.6 – *Architecture du contrôleur UPaRC [5]*

2.2.3 Optimisation de l'ordonnancement

Jusqu'à présent, nous avons uniquement mentionné des moyens matériels d'améliorer le temps de reconfiguration. Une autre solution consiste à simplement réduire le nombre de reconfigurations effectuées en jouant sur l'ordonnancement des tâches matérielles. De nombreux travaux ont été menés sur le sujet, par exemple sur le placement des tâches pour réduire la quantité de ressources inutilisées sur une zone reconfigurable (on parle de fragmentation interne) [32, 33]. Nous ne détaillerons pas ces travaux ici car nous considérons que l'ordonnancement des tâches matérielles représente une problématique à part entière. Il est toutefois intéressant de noter qu'à notre connaissance, aucun outil n'est adapté au développement et/ou à la comparaison d'architectures ou d'algorithmes d'ordonnancement pour le reconfigurable. En effet, nous pensons que l'écriture de l'algorithme d'ordonnancement doit se faire conjointement avec la définition de l'architecture du système. Dans le cas contraire, il faudra potentiellement modifier l'architecture pendant les dernières phases du développement (où l'ordonnancement est défini), entraînant un effort de design supplémentaire non négligeable.

Un des résultats importants issus de ces études est que certains algorithme d'ordonnancement sont capables de prédire les futures tâches à exécuter sur le FPGA [34, 35, 36]. Dans de telles conditions, le bitstream peut alors être chargé en avance depuis sa position initiale (en général une mémoire externe ou même un dépôt de bitstream décentralisé) vers une mémoire plus rapide, au plus proche de l'ICAP (par exemple une BRAM locale comme nous l'avons vu dans la partie 2.2.1.

2.3 Modélisation de la reconfiguration dynamique

Nous allons maintenant étudier différentes approches concernant la modélisation à haut niveau d'abstraction de la reconfiguration dynamique. Dans un premier temps, nous nous concentrerons sur la partie comportementale, à savoir comment modéliser le comportement d'une application sur une architecture reconfigurable. Ensuite, nous étudierons plus en détail les caractéristiques de la reconfiguration dynamique au travers de modèles de coût en termes de temps de reconfiguration ou de consommation énergétique.

2.3.1 Modélisation comportementale

De nombreux efforts ont été fournis afin de proposer des outils de modélisation des architectures reconfigurables. Ces travaux consistent très souvent à étendre les capacités d'outils existant pour des systèmes statiques. On distingue dès lors deux grandes familles d'outils : ceux basés sur UML et ceux utilisant SystemC.

2.3.1.a Modélisation avec UML

UML (*Unified Modeling Language*) [37] est un langage de modélisation graphique et orienté objet, initialement dédié à la modélisation logicielle. Néanmoins, UML est désormais largement utilisé pour modéliser des systèmes matériels ou mixtes, en grande partie grâce aux nombreux outils supportant le langage ainsi que son mécanisme d'extension qui le rend très flexible pour personnalisation des modèles. En particulier, la modélisation de systèmes embarqués est facilitée par l'extension MARTE (*Modeling and Analysis of Real-Time Embedded systems*) [38].

MARTE étends les capacités de modélisation d'UML pour les systèmes embarqués et temps-réel mais ne supporte pas nativement la reconfiguration dynamique. Les travaux présentés en [6, 39] étendent les capacités de MARTE en utilisant le framework GASPARD2, un environnement de développement intégré pour la co-modélisation de SoC qui permet notamment la génération de code en différents langages à des fins de validation (Lustre), de simulation (SystemC) ou de synthèse (RTL). La figure 2.7 montre un exemple de modélisation avec MARTE d'un système reconfigurable dynamiquement. On remarque que certains paramètres ont été ajoutés comme le champ *areatype* qui définit une entité comme statique ou reconfigurable dynamiquement. À notre connaissance, cette approche ne permet pas de développer et valider d'algorithme d'ordonnancement. Ces travaux sont actuellement poursuivis dans le cadre du projet ANR FAMOUS (Flot de modélisation et de conception rapide pour les systèmes dynamiquement reconfigurables) dans le but de livrer l'extension RecoMARTE.

Toujours dans le cadre du projet ANR FAMOUS, les auteurs de [40] présentent un contrôle événementiel de la reconfiguration dynamique. La modification d'une zone reconfigurable n'est plus décidée par un algorithme d'ordonnancement mais par une machine à états finis définie lors de la description du modèle qui donne lieu à la génération automatique d'un contrôleur adapté. La reconfiguration dynamique est utilisée pour modifier le comportement d'un module ayant une fonctionnalité prédéfinie : il n'est pas question ici de mutualisation de ressources entre deux tâches différentes sur une même zone reconfigurable. C'est le point faible

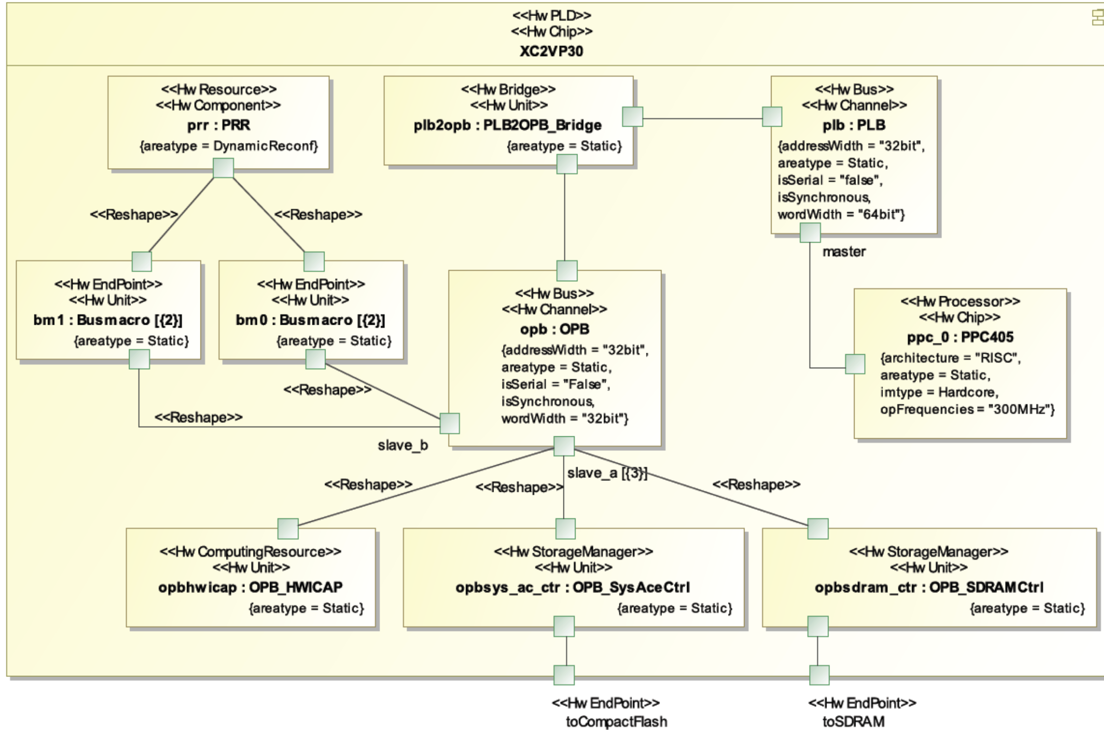


FIGURE 2.7 – *Exemple de modélisation utilisant UML et MARTE [6]*

de cette approche, qui ne prend en compte qu'une partie des aspects de la reconfiguration dynamique.

Une autre approche utilisant MARTE est présentée dans [41, 7], issue du projet MoPCoM (*Modeling and specialization of platform and components MDA*) qui définit une méthodologie de co-design basée sur l'architecture dirigée par les modèles (en anglais MDA, *Model Driven Architecture*). La modélisation est séparée en trois niveaux, depuis un modèle indépendant de la cible jusqu'à un modèle spécifique à l'architecture (voir figure 2.8). Le processus de développement part d'un modèle abstrait qui définit le modèle de calcul (en anglais MoC, *Model of Computation*) pour y intégrer au fur et à mesure les contraintes liées à l'architecture cible. Finalement, le modèle alloué du dernier niveau doit être suffisamment détaillé pour fournir du code d'implémentation des parties matérielles et logicielles de la plate-forme. À chaque niveau, on retrouve une approche en Y qui sépare le modèle de l'application, le modèle de l'architecture cible et les fusionne pour former un modèle alloué. La prise en compte de la reconfiguration dynamique intervient au niveau du modèle alloué. Dans ce cas, plusieurs composants du modèle applicatif sont assignés à un seul composant du modèle architectural.

2.3.1.b Modélisation avec SystemC

Le second grand courant de modélisation du reconfigurable utilise SystemC [42], une librairie C++ open source fournissant des outils de design et de vérification pour les systèmes matériels, logiciels ou mixtes. SystemC est devenu un standard *de facto* car basé sur un langage déjà bien implanté dans la communauté [43], maintenant standardisé IEEE 1664. Il permet notamment la description à différents niveaux d'abstraction selon les besoins du

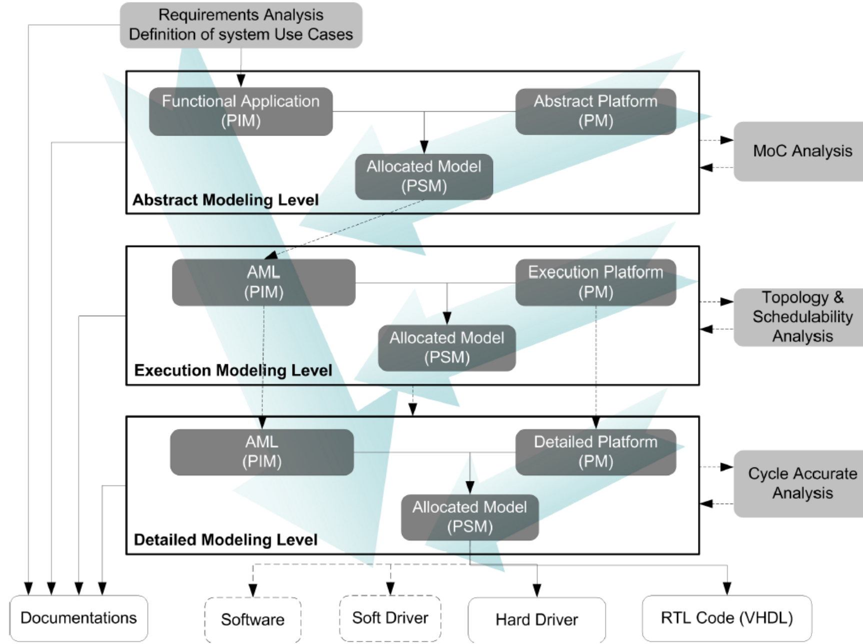


FIGURE 2.8 – *La méthodologie MoPCoM [7]*

concepteur (voir figure 2.9). Ainsi, une description peut être faite au niveau RTL (*Register Transfer Level*) synthétisable jusqu'à des modèles fonctionnels plus abstraits, intégrant ou non la notion de temps.

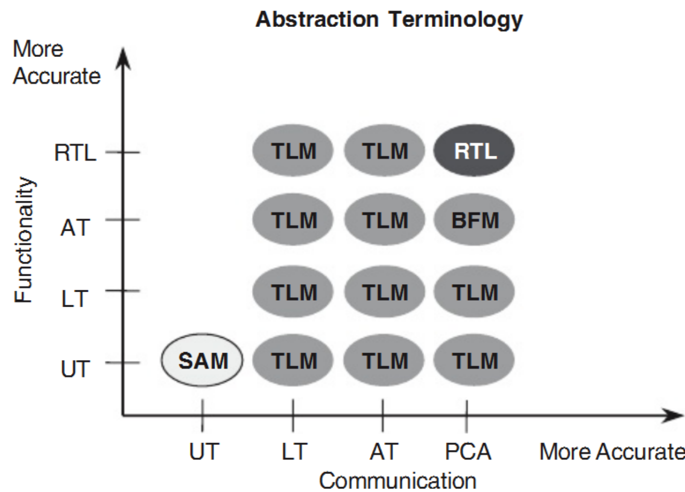


FIGURE 2.9 – *Différents niveaux d'abstraction pour la modélisation SystemC [8]*

Une façon triviale de modéliser le comportement de la reconfiguration dynamique serait de pouvoir remplacer un module en cours de simulation. Malheureusement, SystemC ne permet pas de modifier les connexions d'un module en cours de simulation. Il faut donc trouver d'autres solutions pour ajouter la gestion de la reconfiguration dynamique en SystemC. Parmi ces solutions, on distingue deux groupes : les approches ajoutant des fonctionnalités à SystemC par le biais d'une librairie ou bien la modification du noyau de simulation de SystemC. La première méthode sera toutefois préférée dans le but de conserver une approche

standardisée de la simulation et de pouvoir bénéficier des mises à jour futures de SystemC sans devoir déployer trop d'effort dans la mise à jour du noyau modifié.

Les travaux présentés en [9] apportent une extension à SystemC pour la description et la simulation de systèmes reconfigurables dynamiquement sous la forme d'une nouvelle librairie appelée ReChannel. Cette librairie contourne l'impossibilité de changer les connexions et les instanciations de modules en cours de simulation en instanciant immédiatement tous les modules reconfigurables possibles. Tous les modules se partageant un même zone reconfigurable sont connectés à travers un portail, prenant en charge les types primitifs de SystemC, qui les relie à la partie statique du design tout en transférant les événements et les données vers le module instancié à cet instant (voir figure 2.10). Cette approche est appelée *Dynamic Circuit Switching*. La figure 2.11 montre comment ReChannel utilise les modules natifs de SystemC pour bâtir les modules reconfigurables. ReChannel introduit également un contrôleur de reconfiguration, responsable de l'ordonnancement des tâches matérielles et de la configuration des portails. On pourra toutefois regretter l'impossibilité de migration des tâches vers une autre zone reconfigurable que celle qui lui est assigné en début de simulation.

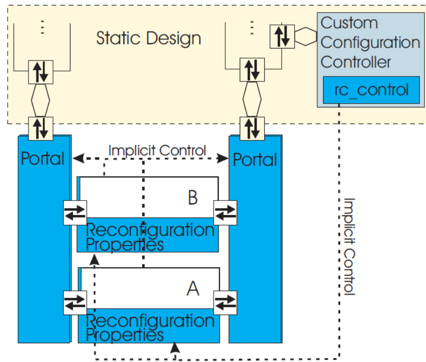


FIGURE 2.10 – *Exemple de design utilisant ReChannel [9]*

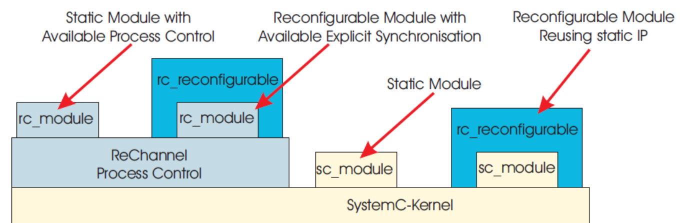


FIGURE 2.11 – *Processus de contrôle de ReChannel [9]*

Les travaux menés dans le cadre du projet européen ADRIATIC (*Advanced Methodology for Designing Reconfigurable SoC and Application-Targeted IP-entities in wireless Communications*) ont menés au concept de fabrique reconfigurable dynamiquement (*Dynamic Reconfigurable Fabric*), contenant les différentes IP pouvant être instanciées dynamiquement sur le FPGA [44]. Cette approche, similaire à celle de ReChannel, permet de changer d'implémentation parmi celles disponibles dans la fabrique. Le flot automatise la transformation du code SystemC statique dans lequel est instancié chaque module prétendant à une implémentation reconfigurable. Le flot génère alors la fabrique de modules reconfigurables ainsi que les entités de contrôle associées. Les candidats sélectionnés pour faire partie de la fabrique sont ceux partageant la même interface. Il n'est pas question ici de prendre en compte les ressources nécessaires à l'implémentation des modules, ce qui pourrait résulter en une fragmentation interne très importante. De nombreuses autres limitations sont décrites en [44] et nous ne savons pas si le projet a abouti depuis.

Toutefois, les travaux en [45] présentent une évolution de ce projet. On y retrouve un équivalent plus complet de la fabrique reconfigurable contenant notamment un ordonnanceur. Dès qu'un appel à un module reconfigurable est effectué, l'ordonnanceur va vérifier si une

reconfiguration est nécessaire. Si oui, une procédure de reconfiguration est appelée, prenant en compte le trafic mémoire généré par la reconfiguration (par exemple l'accès aux bitstreams) ainsi que le temps de reconfiguration associé.

OSSS+R [46] est une méthodologie permettant la spécification algorithmique, la simulation fonctionnelle ainsi que la synthèse automatisée. Elle est basée sur OSSS (*Oldenburg System Synthesis Subset*) [47], anciennement SystemC-Plus, une extension de SystemC qui se concentre sur l'introduction de techniques de modélisation orientées objet durant le design de systèmes mixtes et inclut un outil de synthèse de haut niveau, FOSSY (*Functional Oldenburg System Sythesizer*), utilisé notamment pour la synthèse du contrôleur de reconfiguration. OSSS+R apporte la gestion du reconfigurable en jouant sur le polymorphisme : un conteneur (équivalent d'une zone reconfigurable) défini avec une classe template T pourra accueillir tous les types dérivant de T, modélisant ainsi l'aspect changement de fonctionnalité de la reconfiguration dynamique. Les temps de reconfiguration sont gérés par le simulateur ainsi que les temps liés à la sauvegarde et à la restauration du contexte. Toutefois, OSSS+R est très fortement couplé à son contrôleur de reconfiguration et il est impossible de simuler un design avec un autre contrôleur ou de simplement modifier le contrôleur existant.

Les travaux en [48] décrivent une méthodologie pour la modélisation de systèmes reconfigurables dynamiquement pouvant être appliquée à n'importe quel simulateur dont l'ordonnanceur est basé sur les événements. L'idée est ici de modifier le noyau du simulateur afin de bloquer l'exécution des modules qui ne sont pas configurés sur le FPGA. Cette approche a été validée avec le simulateur SystemC, dont le noyau est facilement modifiable car open-source. L'avantage de cette approche réside dans le fait qu'elle peut être adaptée à n'importe quel simulateur basé sur les événements. Toutefois, elle nécessite la modification du noyau, ce que nous considérons comme une mauvaise solution.

Les auteurs de [10] utilisent une fonctionnalité introduite dans la version 2.1 de SystemC pour rendre dynamique la construction et la destruction de modules : les threads dynamiques. Contrairement aux threads statiques définis pendant la phase d'élaboration du simulateur, les threads dynamiques peuvent être créés à n'importe quel instant de la simulation. Il est donc possible de changer facilement le comportement d'un module en cours de simulation.

Ce module dynamique est composé de deux threads dynamiques : un pour le processus utilisateur et un pour la gestion des processus utilisateurs appelé *spawn_control*. Lors de la création d'un module dynamique, un processus de contrôle est géré. Il est chargé ensuite de la création/destruction et des permutations entre les processus utilisateurs durant la vie du module. La figure 2.12 montre un exemple de suppression du processus utilisateur : une requête est transmise au processus de contrôle qui va à son tour la transmettre au processus utilisateur par le biais d'un événement. Le processus de contrôle se met alors en attente de la fin du processus utilisateur, puis le supprime. En addition des threads dynamiques, les auteurs introduisent des interfaces modifiables en cours de simulation (à l'inverse des interfaces natives de SystemC), ce qui permet de modifier les interfaces du module en cours de simulation.

L'utilisation directe de fonctionnalités natives de SystemC rend cette méthode très flexible et très simple à mettre en œuvre. Néanmoins, les auteurs décrivent uniquement la partie applicative de leur modèle et ne mentionnent pas la partie architecturale. Il semble que ce

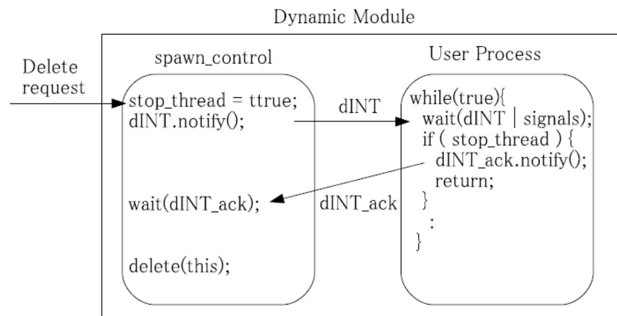


FIGURE 2.12 – *Contrôle de modules dynamiques [10]*

travail ne prenne pas en compte les zones reconfigurables avec la partie allocation des tâches sur une ressource matérielle.

2.3.1.c Modélisation transactionnelle

Un autre paradigme de modélisation apparaissant sur la figure 2.9 présentée en page 19 est la modélisation au niveau transactionnel (de l'anglais TLM, *Transaction Level Modeling*) [49]. Les communications entre modules sont abstraites et ne s'effectuent plus que par transactions. Par exemple, une transaction représentant un accès bus classique contiendra des informations sur le type d'accès (lecture ou écriture en mode simple ou en mode rafale), l'adresse des données, les données à écrire dans le cas d'un accès en écriture et ainsi de suite. La transaction est associée à une information de temps qui représente l'estimation du temps d'accès de l'accès bus réel.

Les avantages de la modélisation transactionnelle sont nombreux. En s'affranchissant d'une représentation trop bas niveau et trop précise du système durant les premières étapes du processus de développement, la modélisation transactionnelle permet de débuter le développement des parties logicielles du système sans attendre la réalisation complète de la partie matérielle. Dans le cadre d'une utilisation industrielle, l'utilisation de TLM permet donc de réduire le temps de mise sur le marché de nouveaux designs, communément appelé *Time to market*. Du point de vue du développeur, l'abstraction des communications précises au cycle près et au bit près (en anglais CABA, pour *Cycle Accurate Bit Accurate*) permet également de réduire drastiquement le temps de simulation du système [50].

La modélisation transactionnelle a été standardisée par l'*Accellera Systems Initiative*, éditrice de SystemC, sous le nom TLM-2.0 pour être aisément intégrée à des modèles SystemC. Des travaux récents mettent en avant une modélisation avec un niveau d'abstraction encore plus élevé, baptisé TLM+ [51]. La méthode consiste à mélanger la partie matérielle avec les pilotes logiciels bas niveau. Ainsi, une séquence de transactions TLM peut être réduite à une seule transaction TLM+, réduisant par la même les temps de simulation. TLM+ s'apparente alors à une abstraction du TLM. Les auteurs annoncent des accélérations d'un facteur 1000 avec un taux d'erreur au niveau des timings de l'ordre de 10% (erreurs dues à l'atomicité augmentée des transactions).

2.3.1.d Autres approches

Jusqu'à présent, nous nous sommes focalisés sur des approches tendant à élever le niveau d'abstraction du modèle au maximum pour s'affranchir des détails d'implémentation et pouvoir utiliser ces modèles le plus tôt possible dans le cycle de développement du produit. Toutefois, il est parfois nécessaire de valider l'implémentation d'un système au niveau HDL. Cela n'est pas possible nativement avec les simulateurs HDL actuels ou avec les approches d'émulation permettant la réduction du temps de validation d'un système, comme l'outil Veloce de Mentor Graphics [52] ou Palladium de Cadence [53]. La première méthodologie pour la modélisation de circuits reconfigurables dynamiquement date de 2000 et utilise le langage de description VHDL [54]. Cette approche repose en fait sur la simulation statique de l'ensemble des implémentations possibles. Par exemple, si le design est composé d'une seule zone reconfigurable ayant n implémentations possibles, l'outil, appelé DCSTech (pour *Dynamic Circuit Switching*), génère $n+1$ designs statiques : un pour chaque implémentation possible et un design minimal, dans lequel la zone reconfigurable n'est pas utilisée. Ces designs statiques peuvent alors être simulés en utilisant les outils classiques de design, comme par exemple ModelSim. L'outil gère le partage des informations de timing entre les différents designs, pouvant ainsi prendre en compte les temps de reconfiguration éventuels.

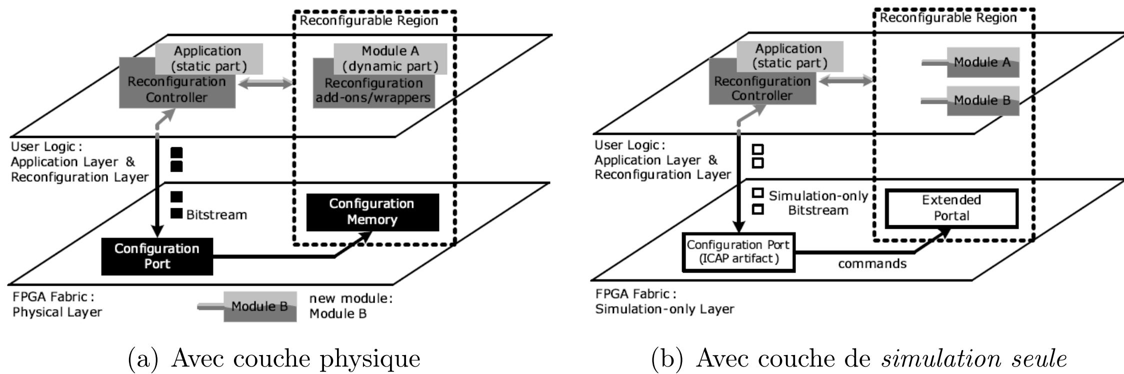
Le principal inconvénient de l'approche précédente est que la simulation se basait sur un temps de reconfiguration et ne simulait pas réellement le processus de reconfiguration. Si cette solution est valable à un niveau d'abstraction élevé, la précision n'est pas suffisante pour des simulations *Cycle-Accurate, Bit-Accurate*. Les travaux présentés en [11] surviennent à ce besoin de précision, en se basant également sur le concept de *Dynamic Circuit Switching* pour changer de module en cours de simulation. Toutefois, pour prendre en compte le temps de reconfiguration du système, les auteurs définissent une couche de *simulation seule*, s'affranchissant de la couche physique, dépendante du FPGA, comme le bitstream de configuration, le port de reconfiguration et la mémoire de configuration (voir figure 2.13). Ainsi, le bitstream physique est remplacé par un bitstream de simulation, à l'architecture identique à un bitstream réel, mais avec des données de configuration aléatoires. La partie commande du bitstream est elle la même que pour un bitstream réel afin d'avoir une simulation très précise. Le contrôleur de reconfiguration, appelé *ICAP artifact*, est capable d'extraire du bitstream de simulation les informations de configuration relatives au module à instancier et pourra alors, par le biais d'un mécanisme de portail, changer le module instancié dans la simulation.

L'avantage de cette approche est que l'on se rapproche au plus près de l'implémentation, tout en étant séparé des considérations physiques liées à la technologie. Ainsi, cette approche simule les accès à la mémoire de stockage de bitstream, permettant de détecter d'éventuelles congestions dans le trafic mémoire.

Le tableau 2.1 résume les avantages et inconvénients des principales méthodes décrites jusqu'à présent pour la modélisation de la reconfiguration dynamique.

2.3.2 Modèle de coût de la reconfiguration dynamique

Comme nous venons de le voir, les travaux les plus avancés intègrent un contrôleur de reconfiguration en charge de l'ordonnancement des tâches matérielles. L'intérêt de tels si-



(a) Avec couche physique (b) Avec couche de *simulation seule*

FIGURE 2.13 – *Analyse structurelle de la reconfiguration dynamique [11]*

TABLEAU 2.1 – *Comparaison des principales approches de modélisation*

Approche	Avantages	Inconvénients
RecoMARTE [6]	- Extension très utilisée	- Pas de modification de l'ordonnancement - Travaux encore en cours
MoPCoM [41, 7]	- Approche en Y - Raffinements successifs du niveau d'abstraction	- Pas de modèle du temps de reconfiguration - Quel ordonnacement ?
ReChannel [9]	- Prise en charge des types natifs de SystemC pour les portails - Extension du type sc_module	- Pas de niveau RTL - Pas de migration de tâche
ADRIATIC [44]	- Transformation d'un code SystemC statique	- Regroupement des modules par interface uniquement - Ressources non gérées
OSSS+R [46]	- Utilisation de l'héritage pour les modules reconfigurables - Sauvegarde/Restauration de contexte	- Impossibilité de changer de contrôleur de reconfiguration
Threads dynamiques [10]	- Flexible et simple à mettre en place	- Pas de description de la partie architecturale
TLM [49]	- Haut niveau d'abstraction - Temps de simulation réduits	- Peu adapté au développment d'IP
ReSim [11]	- Haut niveau d'abstraction - Simulation très fidèle du système	- Très bas niveau d'abstraction

mulateurs repose en partie sur la possibilité de développer des modèles de coûts pour les enrichir.

2.3.2.a Coût en temps de la reconfiguration

À notre connaissance, très peu de travaux ont été menés à des fins d'estimation du temps de reconfiguration. Evidemment, ces travaux sont dédiés à un contrôleur particulier. Dans le cas du contrôleur de Xilinx, les auteurs de [55] présentent leur modèle de coût pour le temps de reconfiguration sur une architecture basée sur le modèle privilégié par Xilinx. Dans ce cas, un microprocesseur MicroBlaze s'occupe de transférer les données depuis une mémoire externe (type Compact Flash ou DDR-SDRAM) vers l'ICAP. Cette approche se base également sur le modèle de coût de Xilinx, très succinct, qui prend uniquement en compte le débit théorique de la macro ICAP (400 Mo/s @ 100 MHz) [15]. Le principal problème de ce modèle reste sa précision. En effet, les auteurs annoncent une variation allant de 30 à 60 % entre la valeur théorique et la valeur expérimentale. De plus, cette approche repose sur une architecture non-optimisée dépendante d'un processeur, une approche à éviter si l'utilisateur souhaite obtenir des performances convenables.

2.3.2.b Coût énergétique de la reconfiguration

Comme nous l'avons déjà vu précédemment, un des avantages de la reconfiguration dynamique est la réduction de la consommation énergétique du système [17, 18, 56]. Dès lors, il serait intéressant de pouvoir intégrer des modèles de consommation au niveau de l'ordonnanceur afin de prendre des décisions en fonction de la consommation courante et souhaitée ou simplement de savoir si une solution est compatible avec les exigences d'un système en termes de consommation.

Nous avons déjà présenté UPaRC [5], un contrôleur de reconfiguration rapide utilisant notamment la compression des bitstreams pour réduire le temps de reconfiguration. Leurs auteurs ont également étudié la consommation énergétique de leur solution pour en déduire un modèle de coût énergétique. Comparé à une architecture basique MicroBlaze + HWICAP, UPaRC est 45 fois plus efficace en termes de consommation ($30\mu J/ko$ contre $0.66\mu J/ko$ de bitstream à transférer). Les auteurs ont également effectué des mesures de consommation pour UPaRC à différentes fréquences de travail pour réaliser une trace de la consommation, représentée en figure 2.14. On peut voir sur la trace que lorsque la fréquence de travail est doublée, le temps de reconfiguration est divisé par deux alors que la consommation n'est pas doublée.

Les travaux décrits en [57] étudient également le gain en consommation qui peut être réalisé en utilisant un accélérateur matériel ou la reconfiguration dynamique. Les auteurs en déduisent un ensemble de règles pour évaluer l'utilité de la reconfiguration dynamique et en arrivent à une conclusion similaire à celle des travaux précédents : les gains en énergie sont maximisés lorsque la fréquence de travail est maximisée. Les pertes liées à l'utilisation de la reconfiguration dynamique à des "hautes" fréquences sont en effet négligeables en comparaison des gains effectués.

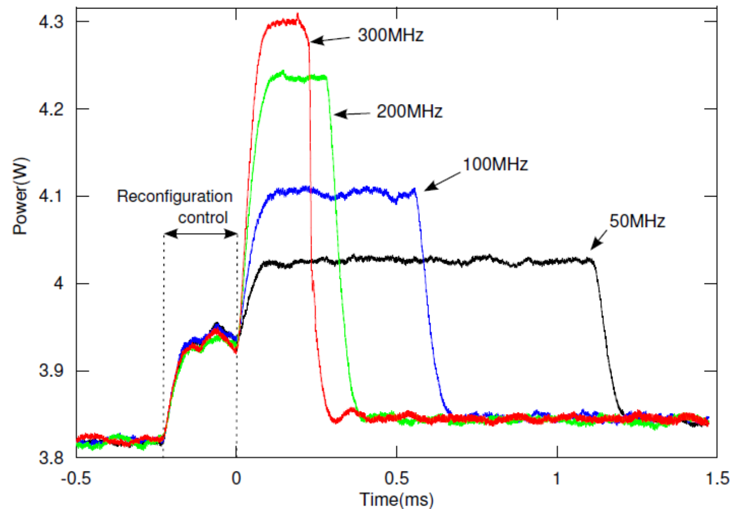


FIGURE 2.14 – *Mesures de consommation d'UPaRC pendant la reconfiguration [5]*

2.4 Exploration d'architectures reconfigurables

Nous avons vu dans la section précédente que les outils de simulation présentés prennent en entrée une architecture cible, c'est-à-dire un FPGA avec des zones reconfigurables placées. On parle de *floorplan*. Sous certaines circonstances, la définition du floorplan, opération appelée *floorplanning*, peut être fait manuellement sans trop de difficultés. Toutefois, au-delà de deux ou trois zones reconfigurables, l'opération peut être compliquée : en effet, en dehors de l'aspect placement des zones reconfigurables, l'outil de floorplanning doit être capable de déterminer quelles zones pourront accueillir chacune des tâches. Dans le cas de systèmes temps-réel avec de fortes contraintes de temps, la solution doit également intégrer un algorithme d'ordonnancement qui assurera au développeur que ces contraintes de temps seront toujours respectées par le système.

Les algorithmes de placement peuvent être séparés en deux catégories : hors ligne et en ligne. Dans la première catégorie, le problème du placement est résolu durant la phase de développement et il est donc envisageable de rechercher une solution quasi-optimale quitte à y passer plus de temps. Dans un scénario en ligne, il est crucial de pouvoir respecter les contraintes de temps de l'application, ce qui nécessite une prise de décision rapide lors du floorplanning. Parmi les travaux effectués sur les algorithmes de placement en ligne, la plupart se consacrent à des résolutions du placement basées sur des modèles mathématiques qui ne prennent pas en compte les contraintes physiques inhérentes à la technologie. Ainsi, la référence en matière d'algorithmes de placement en ligne est [58], décrivant une approche basée sur deux techniques de partitionnement : *Keeping All Maximal Empty Rectangles* (KAMER) et *Keeping Non-Overlapping Empty Rectangles*. Ces techniques cherchent à optimiser les ressources libres après le placement des tâches matérielles (*empty rectangles*). De nombreuses approches ont été dérivées de cette référence, par exemple [59] ou la méthode dite des escaliers [60]. Le point commun à toutes ces approches est que les tâches matérielles sont considérées comme capables d'être relocalisées (c'est-à-dire placées à n'importe quel endroit du FPGA dans une zone reconfigurable de forme quelconque tant que les ressources

allouées sont suffisantes) ce qui n'est pas du tout évident dans la technologie actuelle. Entre autres choses, il faut assurer la compatibilité entre les zones reconfigurables et prendre en compte un éventuel routage statique sur la zone cible [61, 62]. Ainsi, les méthodes en ligne ne nous semblent pas adaptées aux problématiques actuelles du placement de tâches matérielles, puisque le placement des zones reconfigurables doit être réalisé durant la phase de développement. Pour cette raison, nous nous concentrerons ici sur les méthodes de placement hors ligne.

Le placement des tâches matérielles est un problème NP-complet, ce qui signifie qu'un tel problème est résolu dans un temps augmentant exponentiellement avec le nombre de tâches. Certains travaux ([63, 64]) présentent des approches pour la résolution exacte des problèmes d'ordonnancement. Ces approches sont toujours très coûteuses en temps et supportent mal la mise à l'échelle : il est alors compliqué de travailler sur une application composée d'une dizaine de tâches. Plutôt que de rechercher la solution optimale à ce problème, il est très souvent plus avantageux de rechercher une solution quasi-optimale dans un temps plus raisonnable en se basant sur des heuristiques. Par exemple, les travaux mentionnés en [12] utilisent la technique dite du recuit simulé (en anglais *simulated annealing*), méthode inspirée d'un processus utilisé en métallurgie. Dans ce processus, on alterne des cycles de refroidissement lent et de réchauffage (recuit) qui ont pour effet de minimiser l'énergie du matériau. Cette méthode est transposée en optimisation pour trouver les extrema d'une fonction : à chaque itération, la solution courante est aléatoirement modifiée puis testée. Si la nouvelle solution est meilleure, elle est adoptée comme nouvelle solution et le processus se poursuit. Sinon, une nouvelle modification de la solution courante est effectuée. Ainsi, cet algorithme converge au fil des itérations vers une solution quasi-optimale.

Les auteurs de [12] présentent leurs unités reconfigurables, baptisées RFUOP (*Reconfigurable Functionnal Units Operations*). Ces unités reconfigurables sont une représentation en trois dimensions du placement des zones reconfigurables où la troisième dimension représente le temps (voir figure 2.15). Ces boîtes représentant l'utilisation temporelle des zones reconfigurables ne doivent pas se chevaucher pour que la solution soit valide. L'algorithme débute sur un floorplan vierge et recherche les RFUOP qui minimisent les pénalités dues à la réjection des tâches pour trouver la première solution valide.

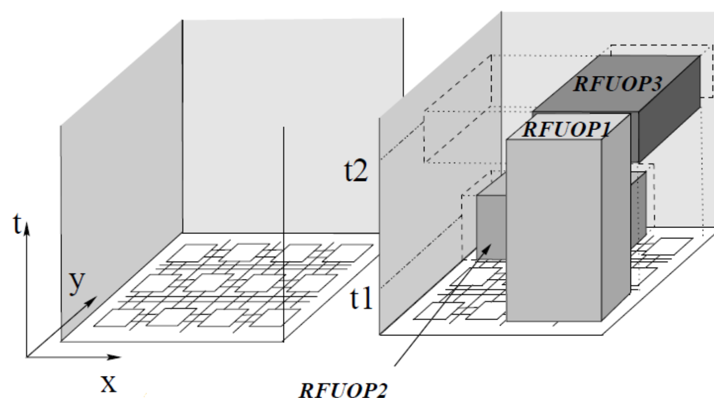


FIGURE 2.15 – *Représentation des unités reconfigurables RFUOP [12]*

Une autre approche basée sur le recuit simulé est présentée en [65]. Les auteurs utilisent

également des paires de séquences pour représenter les différents designs possibles du système qui permettent de déterminer les parties communes des designs. Cette partie commune deviendra la partie statique du système final qui ne sera pas reconfiguré dynamiquement. Les auteurs ont également pris en compte la congestion introduite dans le floorplan par le placement trop rapproché de zones reconfigurables afin de maximiser la faisabilité de l'implémentation du design. D'autres approches utilisent toutefois une métrique inverse afin de qualifier une solution : la fragmentation externe [66]. Cette métrique quantifie la distance entre les différentes zones reconfigurables dans le design, de même que pour la métrique de la longueur de fils externes. Cette distance peut être définie comme étant la distance de Manhattan entre les zones, c'est-à-dire la somme de la distance en abscisse et en ordonnées.

Les travaux en [67] présentent un flot de conception prenant en compte les ressources des tâches et des zones reconfigurables. Les auteurs utilisent des métriques telles que la longueur de fils externes (qui représente la longueur totale de tous les fils nécessaires aux connexions des zones reconfigurables entre elles et avec la partie statique) pour évaluer leurs solutions et avancent une réduction de moitié de cette métrique. Le point faible de cette approche réside dans la partie ressources de leur travail, qui oublie de prendre en compte la nature hétérogène des FPGA actuels. En effet, comme nous l'avons présenté dans la figure 2.1 page 10, les FPGA actuels n'embarquent pas uniquement des cellules configurables (CLB) mais aussi des DSP et des mémoires BRAM qui sont également reconfigurables dynamiquement mais dont le placement diffère d'un FPGA à l'autre. Il est donc impossible de considérer une telle architecture comme homogène. De ce fait, la modélisation des ressources du FPGA se doit d'être flexible pour être facilement adaptable à des FPGA de familles différentes.

À l'inverse, les travaux décrits en [68] prennent en compte l'hétérogénéité des FPGA mais oublient la granularité de reconfiguration, présentée en section 2.1.2. En effet, il est préférable de définir des zones reconfigurables qui ont une hauteur représentant un multiple de la hauteur d'un domaine d'horloge afin de ne pas gâcher les ressources reconfigurables et de réduire la taille des bitstreams générés.

Les auteurs de [69] présentent une méthodologie pour le placement des zones reconfigurables prenant en compte l'architecture cible ainsi que les délais introduits par la reconfiguration dynamique. Le floorplan est déduit à partir des ressources nécessaires aux tâches matérielles après une première étape de partitionnement, décrite en [13]. Chaque solution proposée par l'outil est estimée par le biais d'une fonction de coût définie par la longueur de fils externes et la quantité de ressources gâchées par la solution (fragmentation interne). Toutefois, cette approche considère uniquement des tâches indépendantes en ne considérant pas d'éventuelles dépendances entre tâches. Nous pensons que la dépendance de tâches est une notion très importante dans de nombreux systèmes (par exemple des systèmes robotiques ou dédiés aux transmissions vidéo temps-réel) et constitue donc une vraie lacune de cette approche.

Toutefois, dans [13], les auteurs proposent un partitionnement particulier où deux tâches peuvent être groupées au sein de la même zone reconfigurable, comme présenté dans la figure 2.16. Dans un cas favorable comme celui présenté dans la figure, un tel partitionnement se traduit par un gain au niveau des ressources reconfigurables du système : plutôt que de devoir reconfigurer deux zones en même temps, il est plus efficace de grouper ces deux zones

en une seule, sans avoir besoin de faire un nouveau design, réduisant ainsi le besoin général en ressources reconfigurables.

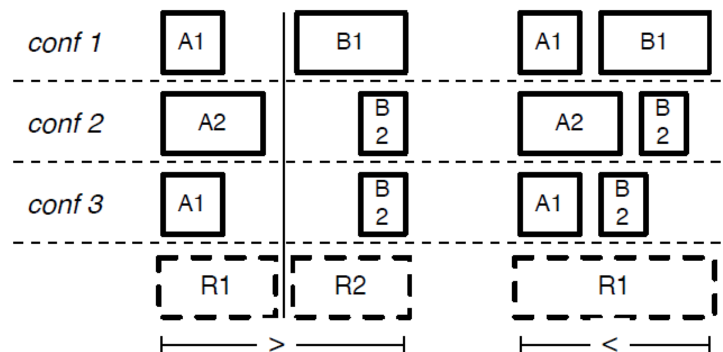
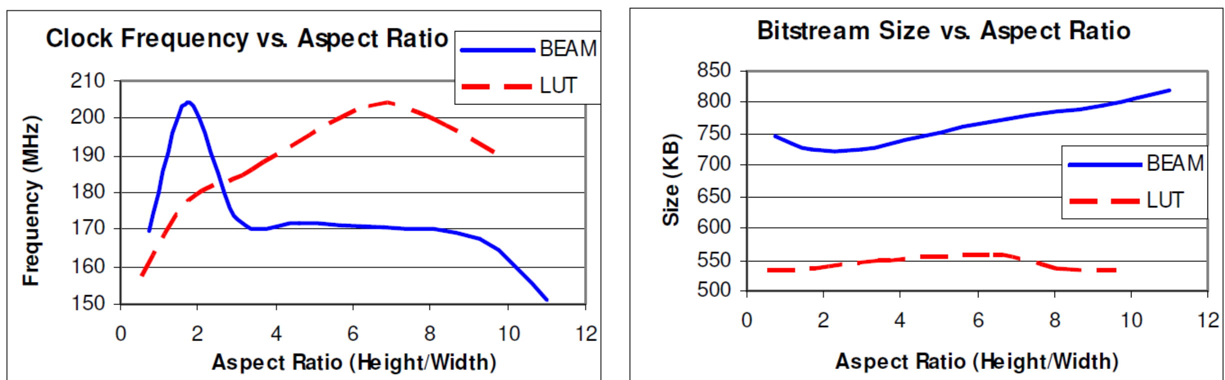


FIGURE 2.16 – *Regroupement de tâches dans une zone reconfigurable [13]*

Cette solution peut toutefois être compliquée à mettre en œuvre : pour conserver les mêmes performances dans les deux cas, il est nécessaire d'avoir exactement les mêmes interfaces pour la zone reconfigurable unifiée que pour les deux zones séparées. Des travaux ont été menés dans cette optique en utilisant la technologie des *bus macros* [70]. Ces macros représentent les interfaces entre la zone reconfigurable et la partie statique du design et permettent notamment d'éviter les *glitches* dans les signaux à l'interface. Elles devaient être définies manuellement par le concepteur, ce qui rendait le design d'applications reconfigurables peu flexibles. Ainsi, Xilinx a remplacé les *bus macros* en 2008 par des *partition pins*, qui sont en fait de simples LUT (*Look-Up Table*, ou table de correspondance en français) placées automatiquement lors de l'implémentation par leur outil de floorplanning, PlanAhead. Des travaux sont actuellement menés dans le cadre du projet ARDMAHN dans le but de porter cette approche sur les *partition pins*.

Les travaux décrits dans [14] présentent une méthodologie pour le développement de systèmes reconfigurables dynamiquement. Cette approche sépare les systèmes en deux classes : systèmes génériques et systèmes spécialisés. Pour les systèmes génériques, les auteurs souhaitent définir une architecture qui pourra être utilisée pour une application qui n'est pas définie lors de la conception de la plate-forme. De ce point de vue, l'architecture cible se doit d'être très générique au niveau des ressources et ne pas proposer d'interfaces trop exotiques. Ainsi, les auteurs, conscients de cette limitation, utilisent alors des interfaces de type bus. Les zones reconfigurables sont alors vues par le système comme des périphériques. Toutefois, une architecture générique est presque toujours une architecture non-optimale. Parmi les caractéristiques utilisées pour quantifier une solution, les auteurs utilisent le temps d'implémentation de la solution, la surface reconfigurables ainsi que le nombre d'entrées/sorties par région (quantifiant la taille des interfaces entre région statique et région reconfigurable dynamiquement. Les auteurs ont également réalisé une étude comparative de la fréquence maximale autorisée après synthèse ainsi que la taille du bitstream compressé en fonction du rapport entre hauteur et largeur de zone reconfigurable (exprimées en nombre de slices) pour deux tâches différentes (voir figure 2.17). La tâche BEAM (pour *beamforming*, une technique de traitement de signal pour la transmission et la réception de signaux directionnels) est très

gourmande en ressources contrairement à la tâche LUT, une table correspondance de sinus et cosinus. Ces graphiques permettent de définir un rapport optimal pour la forme de la zone reconfigurable. Pour des tâches gourmandes en ressources, le rapport optimal entre hauteur et largeur de zone se situe entre deux et quatre. Les auteurs expliquent ce phénomène par la similitude avec le rapport hauteur sur largeur du FPGA ciblé qui est de 2,66. A l'inverse, dans le cas de tâches dont les besoins en ressources sont moindres, le rapport optimal est bien supérieur à 4, ce qui peut s'expliquer par la distribution des ressources plus rares telles que les DSP48 et les BRAM dans le FPGA. Ces résultats peuvent être utilisés afin d'optimiser les zones reconfigurables ou bien simplement de favoriser les zones avec un bon rapport hauteur sur largeur.



(a) Influence sur la fréquence maximale (b) Influence sur la taille du bitstream compressé

FIGURE 2.17 – *Influence du rapport hauteur/largeur des zones reconfigurables [14]*

Les auteurs de [71] présentent leur méthodologie pour l'exploration d'architectures reconfigurables, baptisée FoRSE pour *Formulation-level partial Reconfiguration design Space Exploration*. Les auteurs proposent une exploration basée sur une formulation mathématique du problème pour trouver un optimum de Pareto pour l'application considérée, permettant ainsi de réduire considérablement le temps d'exploration d'architectures en comparaison à des méthodes nécessitant l'implémentation du système (*Implementation-level* en opposition au *Formulation-level*). Un optimum de Pareto est défini comme un état pour lequel il est impossible d'optimiser une métrique sans dégrader les autres. Les métriques considérées par la méthodologie sont assez classiques et ont déjà été énoncées tout au long de ce chapitre. Encore une fois, cette méthodologue ne prend pas en compte d'éventuelles contraintes de temps de l'application. Ainsi, le cas d'utilisation choisi pour leur approche définit déjà le nombre de zones reconfigurables nécessaires au système ainsi que les ressources qui sont contraintes pour chacune d'entre elles. Toutefois, cette méthodologie utilise une directement les modèles de FPGA de Xilinx pour connaître l'architecture interne du composant cible, i.e. le positionnement des différentes ressources dans le FPGA. Ces informations sont contenues dans un fichier au format *Native Packet Header*, représentant l'architecture au format XDL (*Xilinx Description Language*). Les auteurs ont ainsi défini un analyseur syntaxique permettant de récupérer les informations de n'importe quel FPGA, ce qui permet de s'affranchir de la définition manuelle des différents types de FPGA avec une modélisation dépendante de l'outil.

2.5 Conclusion

Dans ce chapitre, nous avons décrit l'état de l'art sur la partie matérielle de la thèse avec les différents moyens d'optimisation du temps de reconfiguration dynamique. Il est possible d'optimiser l'architecture du contrôleur en intégrant une mémoire locale rapide, de type BRAM, pour agir comme une mémoire cache et accélérer considérablement les temps de configuration. Cette mémoire peut également être utilisée à des fins de pré-chargement des bitstreams. Dans le cas où l'ordonnanceur est capable de prédire les configurations futures du système, le pré-chargement des bitstreams permettrait de cacher une partie du temps de reconfiguration, correspondant au transfert depuis dépôt de bitstream vers la BRAM, par l'exécution de la tâche précédente. L'optimisation de l'architecture peut facilement être couplée à une technique de compression des bitstreams. En plus de réduire les besoins mémoire du système pour le stockage des bitstreams, la compression réduit également le nombre de transferts à effectuer pour transférer le bitstream complet vers l'ICAP, réduisant ainsi le temps de reconfiguration. Nous avons vu qu'en dehors du débit réalisé par l'algorithme de décompression, implanté sur le FPGA, celui-ci se doit d'être efficace en termes de ressources nécessaires afin d'être le plus transparent à l'intérieur du système.

Nous avons également vu différentes façons de modéliser le comportement d'architectures reconfigurable dynamiquement avec des approches basées globalement sur le profil MARTE d'UML et sur la librairie SystemC. Nous pensons qu'une approche sur SystemC est plus flexible, permettant notamment la co-simulation avec du code VHDL ou SystemC synthétisable pour simuler directement avec du code d'implémentation. L'approche dite "en Y" est régulièrement utilisée, modélisant séparément l'application et l'architecture cible pour les réunir dans un modèle dit alloué. Cette approche permet aux différents modèles d'être indépendants les uns des autres afin de faciliter leur développement. Un tel simulateur doit également permettre le développement et la validation d'algorithmes d'ordonnancement qui reste un point clé dans le développement de systèmes reconfigurables. Pour cela, des modèles de coût du temps de reconfiguration ou du coût énergétique de la reconfiguration doivent pouvoir être intégrés au simulateur pour explorer des algorithmes d'ordonnancement capables de prendre en compte ces problématiques.

Enfin, nous avons étudié les problématiques d'exploration d'architectures reconfigurables. En particulier, le problème du placement des zones reconfigurables a été abordé. Le floorplan doit être défini hors ligne afin de correspondre aux flots de développement d'architectures actuels comme celui de Xilinx, OpenPR et bientôt Altera. Le placement étant un problème NP-complet, les approches étudiées reposent sur l'utilisation d'heuristiques et d'algorithmes itératifs afin de converger vers une solution quasi-optimale plus rapidement que pour une solution optimale. Par exemple, certaines approches sont basées sur la méthode du recuit simulé. Les solutions sont estimées grâce à des fonctions de coût utilisant des métriques telles que la fragmentation externe, la longueur de fils externes ou la congestion du design. Nous avons vu que le modèle de ressources des FPGA ciblés doit être suffisamment flexible pour bien refléter l'hétérogénéité des FPGA actuels et permettre une détermination des zones reconfigurables efficace. Enfin, nous pensons que le problématiques de respect des contraintes temps-réel ainsi que l'ordonnancement des tâches matérielles, parfois délaissées au profit

d'études très théoriques des techniques de floorplanning, doivent être placées au cœur de l'étude du placement des zones reconfigurables.

De nombreuses études ont ainsi été menées pour proposer des solutions aux problèmes de performances ou de placement des tâches reconfigurables. Toutefois, aucune approche ne prend en considération le problème dans son ensemble. Les solutions proposées à des problèmes distincts sont alors incompatibles ou simplement pas étudiées pour fonctionner ensemble de manière optimale. Nous souhaitons répondre à cette problématique en proposant notre méthodologie pour la conception et la validation d'architectures reconfigurables dynamiquement.

CHAPITRE 3

FaRM : un contrôleur de reconfiguration dynamique pour l'optimisation des performances

Comme nous l'avons vu dans la section 2, les moyens déjà mis en œuvre pour améliorer les performances de la reconfiguration dynamique consistent en une architecture optimisée, utilisant une mémoire locale rapide comme mémoire cache des données de reconfiguration, ainsi qu'en l'utilisation de techniques de compression des bitstreams afin de réduire le nombre d'accès mémoire nécessaires. Nous avons aussi vu que le préchargement des données permettait de cacher une partie du temps de configuration par l'exécution de la tâche précédente. Nous proposons de combiner ces approches au sein de FaRM (pour *Fast Reconfiguration Manager*), une IP développée dans le but de proposer des performances proches de la limite physique du port de configuration des FPGA Xilinx, l'ICAP. Dans le cadre d'une utilisation à haut niveau d'abstraction dans des outils de développement de systèmes reconfigurables dynamiquement, nous proposons également un modèle de coût en temps de reconfiguration associé à l'utilisation de FaRM. L'IP sera enfin testée sur une application d'encodage et de décodage AES (*Advanced Encryption Standard*). Ces résultats nous permettront de décrire une démarche pour le dimensionnement des mémoires internes au contrôleur.

3.1 Notre approche

Dans cette section, nous présenterons l'approche utilisée pour le développement de FaRM. Ainsi, nous décrirons l'architecture de FaRM ainsi que la technique de compression de bitstreams utilisée afin de réduire les temps de reconfiguration. Enfin, nous présenterons les différents modes de fonctionnement intégrés dans FaRM avec notamment la fonctionnalité de relecture permettant de lire les données de configuration d'une zone du FPGA.

3.1.1 Architecture de FaRM

L'architecture de l'IP FaRM est illustrée en figure 3.1. Basée sur une interface intelligente contenant la machine à états finis de contrôle, FaRM dispose d'une interface esclave (comme le contrôleur de Xilinx) chargée de la lecture/écriture dans les registres de contrôle, mais aussi d'une interface maître, responsable des lectures/écritures de données de configuration depuis/vers la mémoire stockant les bitstreams. Sur la figure, ces interfaces suivent le standard AXI [25] de ARM, mais n'importe quel type de bus peut être supporté, du moment que le wrapper correspondant est disponible. L'IP a d'ailleurs été développée initialement pour des bus PLB (*Processor Local Bus*), un standard introduit par IBM dans le cadre de l'architecture CoreConnect.

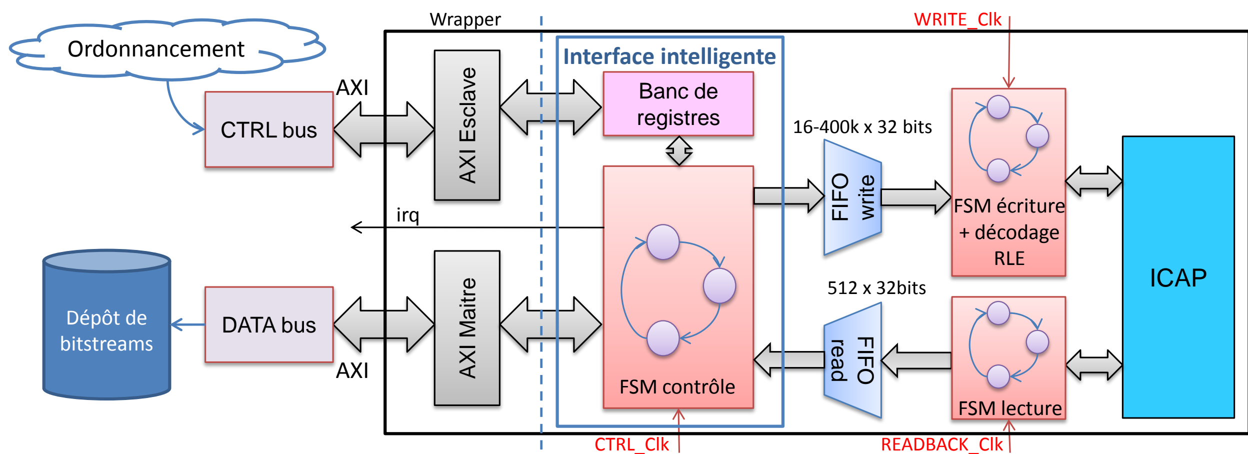


FIGURE 3.1 – *Architecture de l'IP FaRM*

L'interface maître permet à FaRM d'être indépendante du processeur pour les accès de données : le processeur n'est alors utilisé que pour envoyer les commandes de configuration à l'IP (et il peut alors être facilement remplacé par un simple module matériel) ou pour faire tourner l'algorithme d'ordonnancement. L'IP fonctionne à la manière d'un DMA et réduit donc déjà considérablement les temps de reconfiguration. Les deux interfaces peuvent être connectées à des bus différents afin de séparer les accès rapides des données aux accès, plus lents, liés au contrôle de l'IP. C'est d'ailleurs le cas de la norme AXI qui distingue les deux types d'interconnexions : l'AXI est optimisé pour les performances aux dépends de la surface utilisée en générant une interconnexion de type matricielle où les modules sont connectés aux autres à travers une grille d'interconnexion alors que l'AXI-Lite est lui plutôt réservé au contrôle des périphériques avec une connexion de type bus, avec accès partagé entre les différents modules. Un sous-système de configuration AXI que l'on pourrait qualifier de typique est illustré en figure 3.2. Il est donc articulé autour de bus AXI et AXI-Lite. Le bus AXI permet le transfert des bitstreams depuis le dépôt, ici une mémoire DDR externe au FPGA, peu coûteuse. Le bus AXI-Lite sert à la configuration de FaRM à travers l'écriture de ses registres depuis l'ordonnanceur, ici un processeur MicroBlaze. Les zones reconfigurables sont également connectées au bus par une interface esclave à des fins de contrôle. En effet, il est nécessaire d'isoler la zone, de récupérer les résultats de l'exécution de l'IP ou bien de sauvegarder le contexte avant la reconfiguration. Il est important de noter

que FaRM est indépendant de l'application : le sous-système de configuration peut en effet être complètement isolé de la partie applicative du système. Aucune contrainte n'est ainsi héritée de l'utilisation de FaRM.

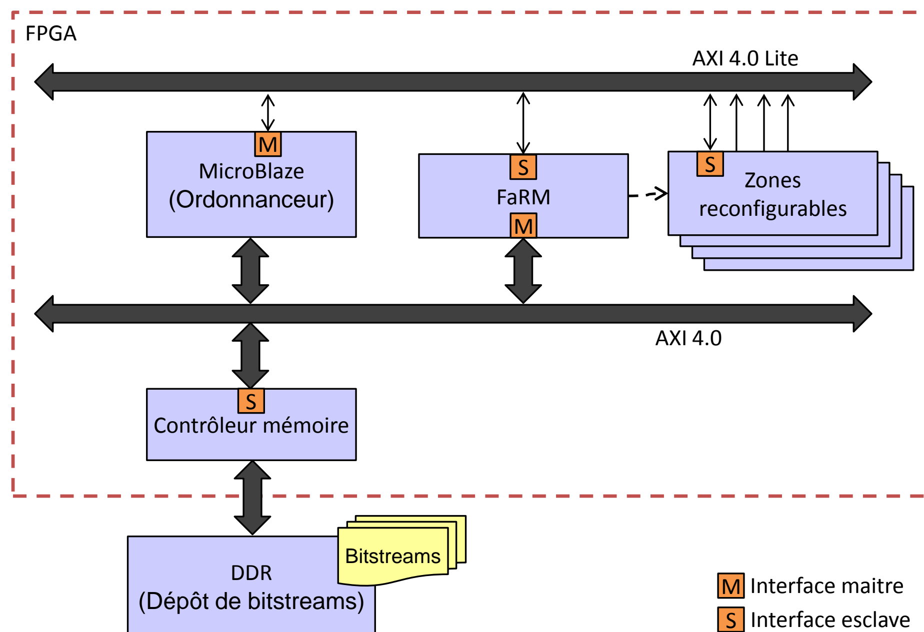


FIGURE 3.2 – *Sous-système de configuration AXI utilisant FaRM*

L'ICAP est contrôlée par deux machines à états finis, respectivement pour la lecture et l'écriture des bitstreams, elles-mêmes interfacées avec deux FIFO afin de séparer les domaines d'horloge au sein de l'IP. Il est en effet fréquent de devoir différencier les trois horloges présentes dans FaRM (horloges de contrôle, d'écriture et de lecture). Par exemple, nous n'avons pas réussi à augmenter la fréquence pour le mode de relecture des données de configuration alors qu'il est possible d'overclocker considérablement les accès en écriture à l'ICAP. Nous avons ainsi réussi à atteindre la fréquence de 200 MHz car au-delà, les accès depuis le bus ne suivaient pas la vitesse imposée par l'ICAP, mais nous avons vu dans l'état de l'art qu'une fréquence de 362.5 MHz était atteignable [5].

La taille de la FIFO d'écriture est paramétrable avec des valeurs allant de 16 à 400k mots de 32 bits. Dans le cas d'une utilisation classique, c'est-à-dire sans préchargement de bitstream, la FIFO se remplit et se vide à la volée et donc une taille de l'ordre de quelques mots n'est pas problématique. Dans le cas du préchargement, la taille de FIFO est cruciale puisqu'elle conditionne les performances de la reconfiguration : par exemple, lorsque la FIFO peut contenir l'intégralité du bitstream, il n'y aura plus besoin d'effectuer d'accès bus supplémentaires lors de la phase de configuration. Le système atteint alors les limites physiques de l'ICAP. Le compromis à trouver entre la taille de FIFO et les performances de la reconfiguration dans le mode de préchargement sera traité dans la section 3.3.3.

3.1.2 Compression des bitstreams

La compression des bitstreams est une technique très intéressante pour réduire le temps de reconfiguration. Pour être efficace, le taux de compression moyen doit être relativement bon et le débit de la partie décompression, implantée sur le FPGA, doit avoir un débit très rapide pour pouvoir nourrir correctement l'ICAP. Nous souhaitons également que l'ajout du décompresseur ne se fasse pas au détriment des ressources disponibles sur le FPGA. Nous visons donc un algorithme de compression relativement simple. Dans le cas particulier des bitstreams, les mémoires BRAM sont les plus gourmandes en données de configuration : il faut notamment initialiser le contenu de la mémoire, même si la valeur est souvent zéro. Cette redondance dans les fichiers de configuration devra être supprimée par l'algorithme de compression. Pour ces raisons, nous avons utilisé un algorithme de compression basé sur le RLE (*Run Length Encoding*, en français codage par plages).

3.1.2.a Principe de fonctionnement du RLE

RLE est un algorithme de compression sans pertes avec lequel les séquences de données identiques sont codées en utilisant un compteur représentant le nombre de fois où la donnée est présente. Cet algorithme semble donc parfaitement adapté à notre cas d'utilisation. Prenons le cas de la séquence de données présentée dans la figure 3.3. Le mot 0x87654321 est répété cinq fois dans la séquence à coder. Dans la séquence résultat, le mot sera présent deux fois pour indiquer une séquence de mots identiques, suivi par un compteur représentant le nombre de fois que le mot apparait en plus des deux premiers dans la séquence à coder. Dans le cas du mot 0x87654321, il faut encore écrire le mot trois fois : le compteur est donc mis à trois.

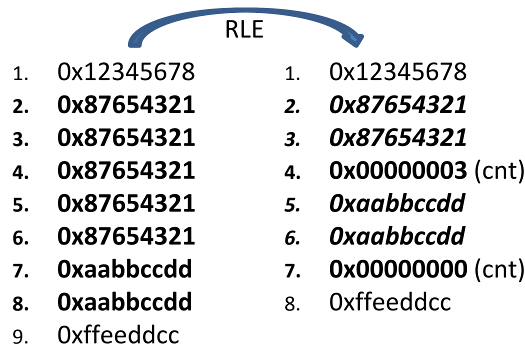


Figure 3.3 – *Exemple de compression RLE*

L'exemple de la figure 3.3 met également en avant le problème de cet algorithme. Dans le cas du mot 0xaabbccdd, présent deux fois dans la séquence à coder, la séquence codée contiendra elle aussi deux fois le mot 0xaabbccdd, mais également le compteur dont la valeur sera zéro. Une séquence de deux mots pourra donc être codée en trois mots, soit 50% de données supplémentaires. Pour parer à cette éventualité, nous avons développé notre propre algorithme à partir de RLE.

3.1.2.b Principe de fonctionnement de l'Offset RLE

L'offset RLE, ou O-RLE, est l'algorithme de compression que nous avons développé afin de supprimer le mot supplémentaire ajouté lors du codage d'une séquence de deux mots identiques. Notre approche est décrite dans la figure 3.4.

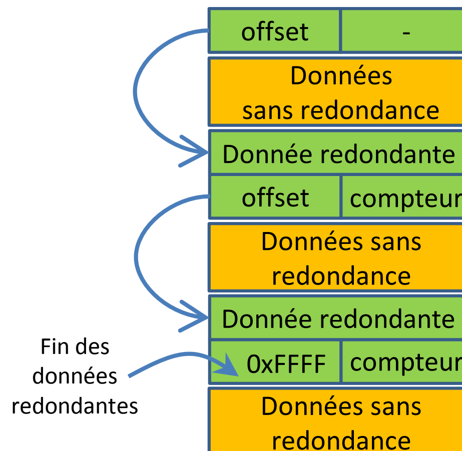


Figure 3.4 – *Principe de la compression O-RLE*

Afin d'indiquer une séquence de données redondantes dans le fichier compressé, O-RLE utilise un mécanisme de pointeurs vers les prochaines données redondantes. Au début du fichier compressé, nous insérons un mot contenant l'offset correspondant à la prochaine donnée redondante. Ainsi, il n'est plus nécessaire de répéter les données redondantes dans le fichier obtenu après codage, puisque l'algorithme connaît la position de cette donnée à partir de l'information d'offset. Après la donnée redondante, nous insérons un mot contenant deux informations : le nombre de fois qu'il faut répéter la donnée (équivalent au compteur du RLE) ainsi que l'offset vers la prochaine donnée redondante. S'il n'y a plus de données redondantes dans le fichier à coder, le mot 0xFFFF est utilisé. La figure 3.5 montre un exemple de codage O-RLE sur la même séquence que celle étudiée pour l'algorithme RLE. Le premier mot de la séquence codée contient l'offset de la première redondance sur les deux octets de poids fort (ici, deux). La première redondance se trouvera donc deux mots plus loin, pour le mot 0x87654321. Le mot suivant correspond au compteur, sur les octets de poids faible (ici, trois) ainsi que l'offset pour la prochaine redondance, ici un mot plus tard. Pour le cas du mot 0xaabbccdd qui posait problème avec RLE, il est ici codé en deux mots, soit autant que la séquence non codée. Dans le cas de l'exemple, c'est aussi la dernière redondance de la séquence, expliquant la présence du 0xFFFF pour coder l'offset. Avec cette technique, on gagne donc un mot supplémentaire par rapport à RLE pour chaque redondance du fichier à coder.

Les informations de redondance et d'offset sont toutes les deux codées sur deux octets. La valeur maximale pour l'offset est 65535 (le mot 0xFFFF étant réservé) alors que celle de la redondance et de 65536, ce qui signifie qu'on ne peut pas coder directement une redondance ou un offset de plus de 65536 mots. Dans ce cas, il faut interpréter ces longues séquences comme plusieurs séquences plus petites, afin de pouvoir coder ces informations en plusieurs fois. Le codage n'est alors plus optimal, mais n'introduit qu'un surplus de deux mots dans

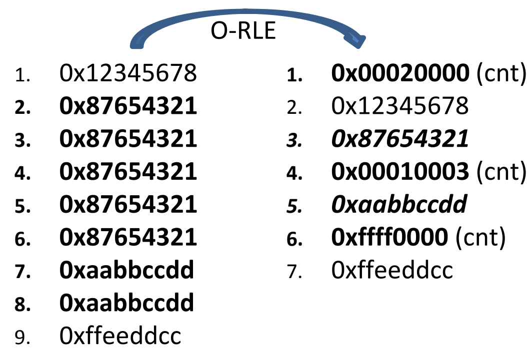


Figure 3.5 – *Exemple de compression O-RLE*

le code, ce qui reste très négligeable sur une séquence de plus de 65000 mots. Toutefois, l'expérience montre que de tels cas sont extrêmement rares.

3.1.2.c Résultats de la compression O-RLE

Le tableau 3.1 présente les taux de compressions obtenus pour l'algorithme O-RLE utilisé par FaRM et les compare aux taux de compression obtenus pour d'autres algorithmes réputés comme RLE, ZIP ou 7z. Les mesures ont été effectuées sur des bitstreams de configuration générés pour des IP hétérogènes en termes de taille, de complexité et de ressources afin de représenter au mieux la réalité des taux de compression. Pour chaque algorithme, un taux de compression moyen est calculé. Nous avons utilisé une moyenne géométrique plutôt qu'une moyenne arithmétique afin de lisser le résultat et d'éviter de trop prendre en compte les taux extrêmes que nous avons pu mesurer (par exemple pour le bitstream *AES black box*, un bitstream assure que la zone reconfigurable est bien vierge, limitant ainsi la consommation d'une zone inutilisée).

Tableau 3.1 – *Comparaison de taux de compression pour différents algorithmes. Les valeurs données représentent les gains réalisés (exprimés en pourcentage de la taille du bitstream initial)*

IP	Taille (octet)	O-RLE	RLE 16	RLE 32	ZIP	7z
AES decoder	97676	23.4	21.6	20.9	57	60.8
AES encoder	97676	31	31.6	26.6	67.6	69.7
AES Black Box	97676	90.8	93	88.1	97.3	97.5
Basic DES	24348	7	4.4	4.1	48.3	50.9
Basic RSA	68156	36.9	35.4	35.8	61.8	65.4
DCT	66844	41.1	40.5	38.8	65.3	67.9
FFT64	161308	32.9	31.3	32.2	57.3	60.8
FIR_7_16_8	17132	11.7	7.6	7.3	64.3	65
FIR_7_16_16	23036	12.8	10.3	7.5	65.3	67.1
Hibert	24348	19.9	18.7	14.8	65.4	66.7
Count 1	6632	51.2	48.9	44.8	78.3	78.5
Count 2	6632	51	48.8	44.6	78.2	78.3
Moyenne géométrique		**26.7**	**23.9**	**21.5**	**66.1**	**68.1**

Nous obtenons ainsi un taux de compression moyen de 26.7% pour FaRM, ce qui représente une certaine amélioration par rapport aux algorithmes RLE en granularité 16 bits et 32 bits. Toutefois, nous obtenons des taux de compression bien plus faibles que pour des algorithmes largement répandus sur ordinateurs comme ZIP et 7z. Toutefois, notre solution a des besoins en ressources relativement faibles et bénéficie d'un débit d'un mot par cycle d'horloge ce qui en fait un bon candidat pour une implantation matérielle au sein de FaRM.

3.1.3 Modes de fonctionnement

En dehors du mode de fonctionnement classique d'écriture de bitstreams, qu'ils soient compressés ou non, FaRM propose deux autres modes de fonctionnement permettant respectivement le préchargement de bitstream ou la relecture de données de configurations.

3.1.3.a Mode de préchargement

Ce mode permet à l'utilisateur de précharger tout ou partie d'un bitstream dans la FIFO d'écriture de FaRM. Ainsi, le bitstream préchargé pourra être envoyé directement vers l'ICAP en s'affranchissant des transferts mémoire sur le bus qui sont cachés par l'exécution de la tâche précédente. Afin d'obtenir le débit théorique maximal, il n'est pas nécessaire que la FIFO soit aussi grande que le bitstream. En effet, pendant que la FIFO se vide dans l'ICAP, elle se remplit également de l'autre côté avec la suite du bitstream de configuration. Des études plus poussées sur le dimensionnement de la FIFO seront effectuées dans la section 3.3.3. La figure 3.6 montre le fonctionnement du mode de préchargement et le compare au mode d'écriture basique. On remarque bien que la configuration de la seconde tâche est plus rapide du fait d'un préchargement des données de configuration pendant que la première tâche est toujours en cours d'exécution. Pendant la période de configuration effective, le débit atteint 800 Mo/s (32 bits @ 200 MHz) pendant toute la période durant laquelle la FIFO n'est pas vide.

3.1.3.b Mode de relecture automatisée

L'une des fonctionnalités majeures de FaRM est son mode de relecture automatisée, qui fournit un moyen simple et efficace de lire les données de configuration de n'importe quelle partie du FPGA, donc en particulier la configuration des zones reconfigurables. Ce mode de relecture peut être utilisé afin de sauvegarder le contexte d'une tâche qui n'a été prévue à l'origine pour être préemptée. Deux solutions sont alors disponibles pour l'utilisateur : relire l'intégralité de la zone reconfigurable sur laquelle est placée la tâche, une solution très gourmande en temps et en mémoire pour stocker les données, ou bien lire les données de configuration à l'emplacement de registres particuliers (par exemple les registres de configuration ou d'état de l'IP), dont la position est définie durant la phase de génération du bitstream.

La relecture est réalisée en trois temps, comme illustré dans la figure 3.6 :

1. Une suite de commandes est envoyée à l'ICAP, permettant de configurer l'accès en relecture avec la quantité de données à transférer ainsi que l'adresse de la frame de départ. Cette adresse correspond au FAR (*Frame Address Register*) de l'ICAP [15]

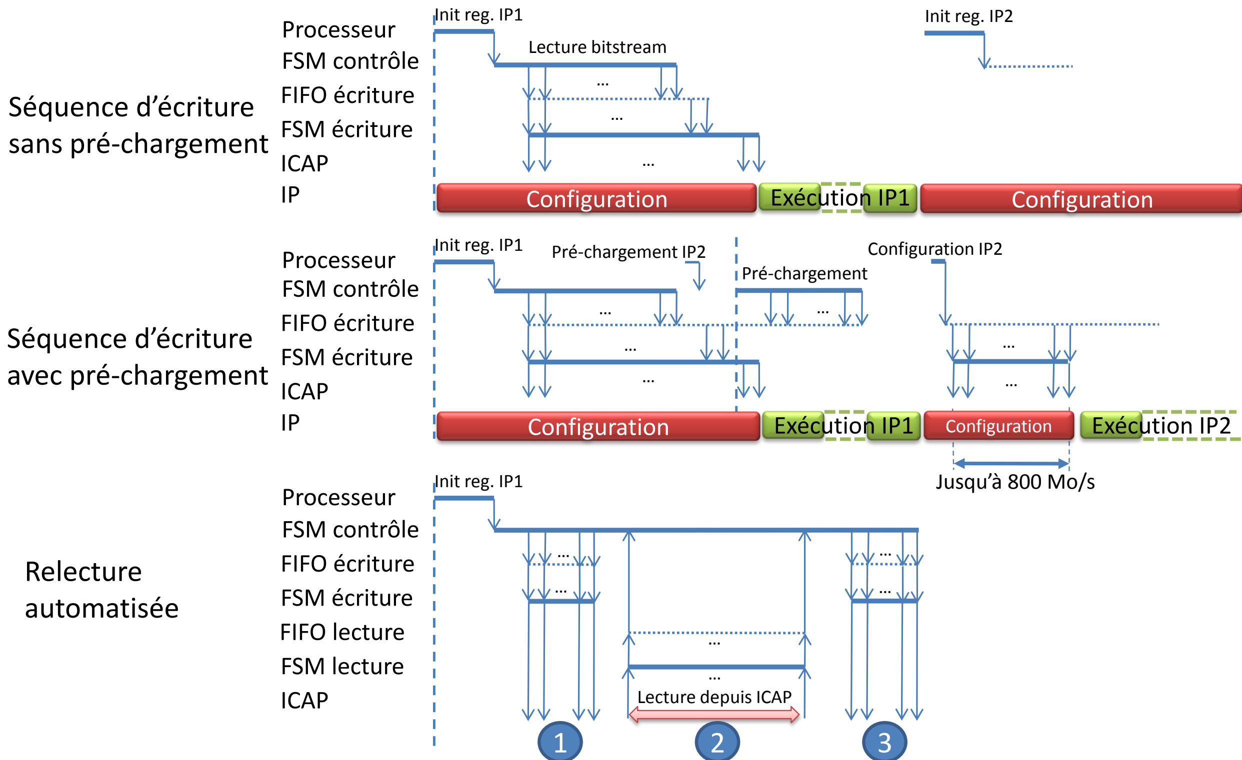


FIGURE 3.6 – *Modes de fonctionnement de FaRM*

dont la composition est donnée par la figure 3.7. Une frame est ainsi définie par le type de ressource qu'elle configure (*Block Type*), sa position dans la partie haute ou basse du FPGA (*Top/Bottom Row*), la ligne et la colonne de ressources concernée (respectivement *Row Address* et *Major Address*) et enfin le numéro de la frame parmi toutes celles composant la colonne (*Minor Address*). C'est FaRM qui va s'occuper d'aller lire cette séquence de commandes qui doit être stockée en mémoire. L'ordonnanceur s'occupe de configurer le registre correspondant à l'adresse de cette zone mémoire dans FaRM.

2. La relecture effective est alors possible. Les données sont lues depuis l'ICAP pour être transférées par la machine à états de contrôle vers l'adresse définie dans le registre correspondant de FaRM.
3. Une dernière séquence de commandes doit être envoyée à l'ICAP pour mettre fin à la relecture. Comme pour la première séquence d'écriture, cette séquence doit être stockée à un emplacement mémoire dont l'adresse est fournie à FaRM par le biais d'un registre dédié.

Le processus de relecture de FaRM est entièrement automatisé : une fois les registres configurés (par exemple avec les emplacements mémoires des trois composantes de la relecture ou la quantité de données à extraire), les trois étapes s'enchainent sans intervention externe. La fin de la relecture est notifiée à l'ordonnanceur par le biais d'une interruption ou par attente active (*polling*) sur le registre d'état de FaRM. Les accès à la mémoire en lecture ou en écriture sur le bus utilisent des accès en mode rafale (*burst*) de taille configurable. Pour le standard de bus PLB, utilisé pour la première version de développement de FaRM,

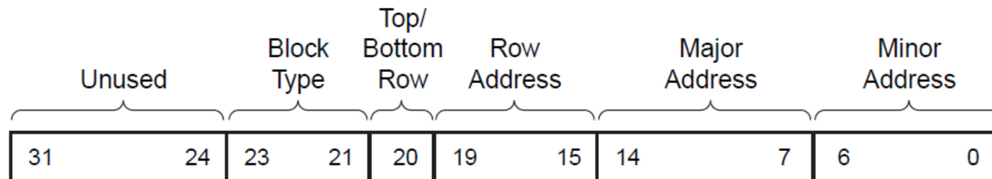


FIGURE 3.7 – *Adressage des frames de configuration [15]*

la taille de ces accès pouvait varier entre deux et seize mots de 32 bits. Dans sa version AXI, FaRM autorise des accès rafale de taille allant jusqu'à 256 mots. La figure 3.8 montre une comparaison des débits obtenus en fonction de la taille des rafales à une fréquence de fonctionnement de 100 MHz. Nous pouvons voir qu'à taille équivalente, les protocoles AXI et PLB ont des performances très similaires. Lorsque l'on augmente la longueur des rafales, nous remarquons une forte augmentation du débit qui approche le débit théorique de 400 Mo/s pour un fonctionnement à une fréquence de 100 MHz. Ceci est dû à la réduction du nombre d'accès nécessaires à la complétion du transfert, qui réduit ainsi le temps passé à la configuration du transfert sur le bus. En contrepartie, l'accès au bus est réservé pendant une plus grande période de temps pour FaRM, ce qui peut poser des problèmes dans le cas où le bus est commun à plusieurs IP. Dans le cas d'un sous-système comme celui présenté dans la figure 3.2 en page 35, le bus est dédié au sous-système et l'augmentation de la longueur des rafales n'est plus problématique. Les résultats de la figure 3.8 démontrent clairement que le facteur limitant pour le débit lors de la reconfiguration se situe au niveau des accès bus, alors que l'ICAP est capable d'être utilisée avec un débit proche d'un mot par cycle d'horloge.

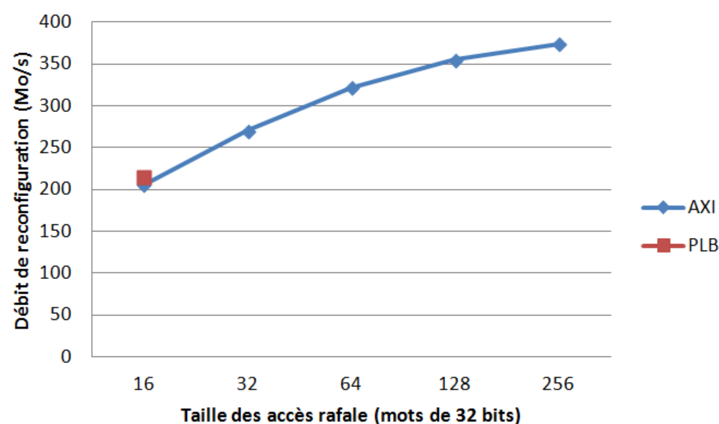


FIGURE 3.8 – *Influence des tailles des accès rafales sur la reconfiguration*

Il est important de noter que FaRM peut effectuer une relecture pour n'importe quelle taille de donnée souhaitée, ce qui n'est pas le cas de l'IP HwICAP de Xilinx qui ne peut relire qu'une frame à la fois. Ainsi, dans le cas de la relecture de n frames avec le contrôleur de Xilinx, il sera nécessaire d'écrire n fois les données de configuration de l'ICAP (phases 1 et 3 décrites précédemment), résultant en un coût en temps non négligeable. De plus, chaque frame lue est accompagnée d'une frame dite de remplissage (*padding*) ne contenant pas de données mais étant liée au fonctionnement de l'ICAP. En effet, la relecture d'une frame nécessite de vider un buffer avant d'accéder aux données de configuration effectives. Cette

frame de remplissage est donc obligatoirement lue, mais il n'est absolument pas nécessaire de la transférer sur le bus ou de la stocker en mémoire. Cette amélioration a été mise en place dans FaRM, ce qui permet de réduire les accès bus en lecture de moitié lors de la relecture d'une seule frame. Lors de la lecture de plusieurs frames permise par FaRM, les gains sont moins importants puisqu'il n'y aura toujours qu'une seule frame de remplissage insérée en début de relecture.

Enfin, FaRM instancie la macro CAPTURE [15], un bloc fondu au sein de l'ICAP qui permet d'échantillonner les sorties des BRAM ainsi que l'état de configuration des CLB et des IOB. Ce mode de relecture permet donc en théorie de sauvegarder l'état exact d'une partition reconfigurable pour pouvoir la restaurer par la suite. Nous avons appelé ce mode *Writeback* (réécriture). En pratique, le contenu des BRAM ne peut pas être échantillonné correctement ce qui fait que la partition réécrite n'est pas toujours utilisable dans sa nouvelle configuration, ce qui représente un véritable obstacle à l'utilisation de ce mode de fonctionnement. Toutefois, les bases de l'utilisation de la relecture et de la réécriture sont posées et seront a priori réutilisable pour les futures générations de FPGA Xilinx.

3.2 Modèle de coût du temps de reconfiguration

Nous allons maintenant introduire le modèle de coût en temps de reconfiguration développé pour FaRM, qui fait le lien entre l'utilisation de FaRM et la modélisation comportementale d'architectures reconfigurables dynamiquement. En effet, la connaissance du temps de reconfiguration est nécessaire afin de pouvoir affirmer que le comportement du système suivra bien les exigences de l'application.

Le temps de reconfiguration a été estimé pour les différents modes de fonctionnement de FaRM décrits dans la section 3.1.3. Ces modèles de coût ont été initialement développés pour la version PLB de FaRM. Toutefois, ces modèles sont suffisamment paramétrables pour être utilisés pour la version AXI de FaRM ainsi que pour tout autre standard de bus tant que l'architecture du sous-système de configuration est similaire.

3.2.1 Coût du mode d'écriture simple

De nombreuses mesures ont été effectuées en utilisant l'outil Chipscope Pro de Xilinx, un analyseur logique dédié au FPGA et permettant de réaliser des traces de signaux internes au FPGA en cours d'exécution. En particulier, il est possible de visualiser les signaux du bus PLB. La figure 3.9 montre un exemple de trace au format VCD (*Value Change Dump*) obtenue avec Chispcope Pro et visualisée avec l'outil de simulation de Mentor Graphics ModelSim.

Ces traces nous ont permis de valider le fait que le facteur limitant du débit de reconfiguration est l'accès à la mémoire depuis le bus, phénomène que nous avions déjà mis en avant avec la figure 3.8. En effet, la figure 3.9 nous montre qu'une fois qu'un accès est terminé sur le bus, l'ICAP a le temps d'utiliser les données produites avant la fin de l'accès suivant. La fin d'un accès bus est représenté par le signal *ipif_Bus2IP_Mst_Cmplt* dont les fronts montants ont été marqués par des curseurs sur la figure. La période de fonctionnement de l'ICAP est représentée par le signal *icap_busy* ainsi que par les nombreuses modifications

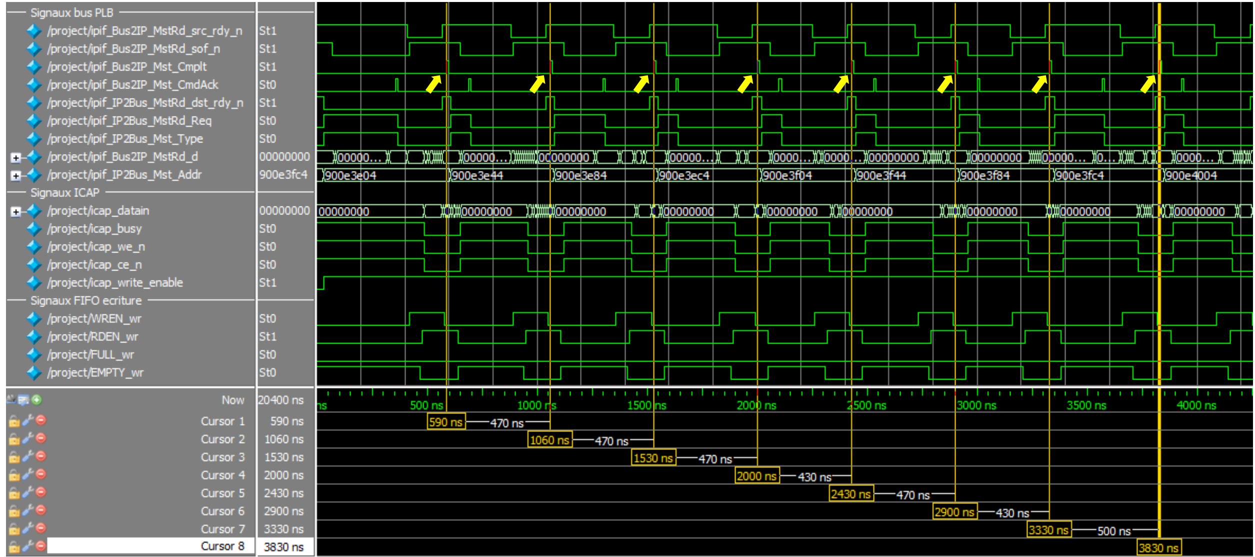


FIGURE 3.9 – *Exemple de trace obtenue avec Chipscope Pro*

de la valeur du bus de données *icap_ datain*. Ainsi, le coût en temps de reconfiguration est majoritairement représenté par le coût en transferts sur le bus. Nous obtenons donc une première équation correspondant au temps de reconfiguration dans le mode d'écriture classique et sans compression. Considérons les notations suivantes :

$N_{bitstream}$: la taille du bitstream en nombre de mots de 32 bits

T_{bus} : la période d'horloge du bus en nanosecondes

$latence$: la latence de l'accès bus en nombre de cycles d'horloge

t_{burst} : le temps moyen pour effectuer un accès rafale en nombre de cycles d'horloge

N_{burst} : la longueur des accès rafales en nombre de mots de 32 bits

L'équation (3.1) donne une estimation du temps de reconfiguration pour le mode d'écriture normal sans utiliser la compression des bitstreams :

$$t_{\text{write}}\,(\text{ns}) = (latence + t_{\text{burst}} * \lfloor \frac{N_{\text{bitstream}}}{N_{\text{burst}}} \rfloor + (t_{\text{burst}} - N_{\text{bitstream}} \bmod N_{\text{burst}})) * T_{\text{bus}} \tag{3.1}$$

Pour estimer ce temps, nous calculons tout d'abord le nombre d'accès nécessaires à l'écriture de l'intégralité du bitstream. Ce nombre d'accès est ensuite multiplié au temps moyen mesuré pour des accès rafales identiques. Dans notre cas, ces mesures ont été effectuées à l'aide de Chipscope Pro. Un exemple de ces mesures est visible sur la figure 3.9. Selon les cas, un accès rafale plus court (de taille inférieure à la taille maximale fixée) pourra être effectué en fin d'écriture. Le temps nécessaire à cet accès supplémentaire n'est pas exactement le même que pour un accès rafale complet. Le temps est ajusté en enlevant le nombre de mots restant à écrire : l'idée derrière cet ajustement est que le débit au sein d'un accès rafale est d'un mot par cycle après la latence d'initialisation de l'accès. On ajoute enfin la latence initiale pour configurer la procédure d'écriture. La figure 3.10 illustre le déroulement d'une transaction sur le bus obtenue avec Chipscope Pro. Le premier curseur marque le début de l'accès bus,

lorsque les signaux de contrôle sont positionnés. Les deux derniers curseurs marquent respectivement le début et la fin du transfert des seize données qui est rendu possible lorsque la source du transfert (ici une DDR2) est disponible (signal *ipif_ Bus2IP_ MstRd_ src_ rdy_ n*).

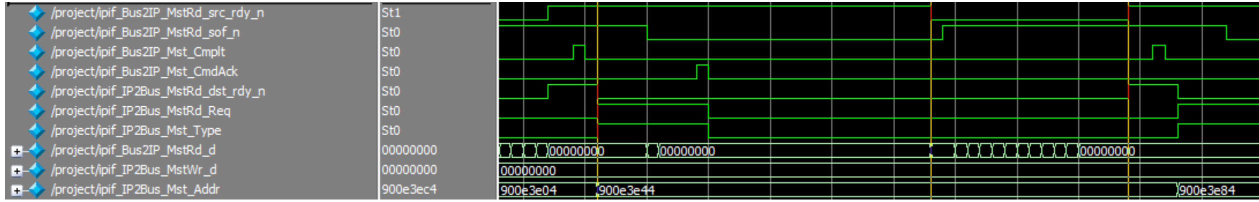


FIGURE 3.10 – *Trace VCD représentant une transaction sur le bus PLB*

Dans le cas de bitstreams compressés, le temps de reconfiguration ne peut pas être précisément estimé car la compression n'est pas homogène au sein d'un fichier. En effet, la partie configuration du contenu d'une mémoire BRAM, localisée à la fin des bitstreams, donnera lieu à un taux de compression local très fort. D'un point de vue écriture vers l'ICAP, un taux de compression important signifie qu'un mot du bitstream compressé pourra générer une longue séquence de mots dans le bitstream décompressé qui sera envoyée vers le port de configuration. Dans le cas (optimiste) d'une compression régulière, il suffit d'introduire le taux de compression dans l'équation (3.1) pour en déduire la taille du bitstream compressé à transférer. Dans le cas le plus défavorable, la compression est entièrement localisée à la fin du bitstream : une fois le bitstream transféré depuis la mémoire vers FaRM, il faut encore attendre que tous les mots issus de la décompression du dernier accès soient envoyés vers l'ICAP. Considérons les notations suivantes :

T_{icap} : la période d'horloge de l'ICAP en nanosecondes

$ratio$: le taux de compression du bitstream (valeur entre 0 et 1, 1 représentant un bitstream non compressé)

Les équations (3.2) et (3.3) nous donnent respectivement un minorant et un majorant du temps de reconfiguration nécessaire pour des bitstreams compressés :

$$t_{\text{min}}^{compressed}(\text{ns}) = (latency + t_{\text{burst}} * \lfloor \frac{ratio * N_{\text{bitstream}}}{N_{\text{burst}}} \rfloor + (t_{\text{burst}} - (ratio * N_{\text{bitstream}}) \bmod N_{\text{burst}})) * T_{\text{bus}} \tag{3.2}$$

$$t_{\text{max}}^{compressed}(\text{ns}) = t_{\text{min}}^{compressed} + N_{\text{bitstream}} * (1 - ratio) * T_{\text{icap}} \tag{3.3}$$

Dans le cas d'une utilisation du modèle de coût pour une IP homogène, ne nécessitant par exemple que des blocs CLB, l'équation (3.2) fournira une bonne estimation, puisque dans ce cas, la compression peut être considérée comme régulière. A l'inverse, dans le cas d'une IP nécessitant des mémoires BRAM, l'équation (3.3) donnant la borne supérieure sera probablement la plus proche du temps de reconfiguration effectif.

3.2.2 Coût du mode de préchargement

Avec l'utilisation du mode de préchargement, il devient plus compliqué d'estimer précisément le temps de reconfiguration nécessaire. Le principal obstacle vient du fait que pendant que les données pré-chargées sont envoyées sur l'ICAP, de nouvelles données peuvent être écrites dans la FIFO (dans le cas où le bitstream de configuration est plus large que la FIFO). Nous avons donc séparé le processus de préchargement en deux parties :

- Une première partie durant laquelle les données préchargées sont envoyées sur l'ICAP, en atteignant le débit théorique du port de configuration (un mot par cycle, indépendamment de la fréquence de fonctionnement jusqu'à 200 MHz). Le nombre de mots envoyés vers l'ICAP durant cette phase n'est pas exactement le même que la taille de la FIFO, puisque celle-ci se remplit grâce aux accès bus dans le même temps. La compression influe également sur le nombre de mots transférés.
- Une seconde phase qui démarre une fois que la FIFO a réussi à se vider complètement. Plus aucune donnée n'est préchargée et le comportement de FaRM est le même que pour le mode d'écriture classiques, sans préchargement.

L'algorithme 1 que nous avons mis au point permet d'estimer le nombre de mots écrits durant la première phase du préchargement. Tout d'abord l'algorithme vérifie si le bitstream complet est contenu dans la FIFO (lignes 1 à 5). Nous supposons ici que le système a eu suffisamment de temps pour compléter le préchargement, c'est-à-dire que la FIFO a été entièrement remplie ou que le bitstream a été intégralement préchargé.

L'algorithme itère ensuite tant que la FIFO n'est pas vide (signifiant que la première phase du préchargement est terminée) ou tant que le bitstream n'est pas complètement transféré. À chaque itération, l'algorithme calcule le temps nécessaire à l'écriture de l'intégralité des mots contenus dans la FIFO (t_{iter}, ligne 7), en considérant un débit d'un mot par cycle d'horloge au niveau de l'ICAP. Ce temps est alors utilisé pour calculer le nombre de mots qui ont été écrits dans la FIFO durant cette période $N_{wordNext}$, en prenant en compte le taux de compression et le temps nécessaires aux accès bus (ligne 8), de la même manière que pour les équations 3.2 et 3.3. La quantité de données écrite est corrigée dans le cas où tout le bitstream a été transféré à l'ICAP (lignes 11 à 14). La boucle s'arrête lorsqu'il n'y a plus de mots à écrire dans la FIFO (bitstream intégralement transféré au débit maximal théorique) ou lorsque la FIFO a été complètement vidée. Les deux données importantes résultant de cet algorithme sont alors le temps passé durant la première phase t_{load} ainsi que le nombre de mots restants à écrire pendant la seconde phase du préchargement $N_{wordLeft}$.

Ce résultat peut alors être combiné avec les équations précédemment définies pour estimer les temps de reconfiguration en préchargement. Dans le cas d'une utilisation du mode de préchargement sans compression, l'estimation du temps de reconfiguration est obtenue en combinant les résultats de l'algorithme 1 avec l'équation 3.1. Considérons les notations suivantes :

t_{load} : le temps passé lors de la première phase du préchargement

$N_{wordLeft}$: le nombre de mots restant à écrire lors de la seconde phase

Nous obtenons alors l'estimation du temps de reconfiguration en mode préchargement sans compression avec l'équation 3.4 :

Algorithme 1 Nombre de mots écrits durant la première phase du préchargement

1: **si** $N_{bitstream} * ratio < fifo_depth$ **alors**
2: $N_{word} \leftarrow N_{bitstream}$
3: **sinon**
4: $N_{word} \leftarrow fifo_depth/ratio$
5: **fin si**
6: **répéter**
7: $t_{iter} \leftarrow T_{icap} * N_{word}$
8: $N_{wordNext} \leftarrow ((t_{iter}/(t_{burst} * T_{bus})) * N_{burst})/ratio$
9: $t_{total} \leftarrow t_{total} + t_{iter}$
10: $N_{wordTotal} \leftarrow N_{wordTotal} + N_{word}$
11: **si** $N_{wordTotal} + N_{wordNext} > N_{bitstream}$ **alors**
12: $sat \leftarrow 1$
13: $N_{wordNext} \leftarrow N_{bitstream} - N_{wordTotal}$
14: **fin si**
15: $N_{word} \leftarrow N_{wordNext}$
16: **tant que** $N_{word} > 1$ **et** $\lceil N_{wordTotal} \rceil \neq N_{bitstream}$ **et** $(N_{wordNext} > N_{burst}$ **ou** $sat = 1)$
17: $t_{load} \leftarrow t_{total}$
18: $N_{wordLeft} \leftarrow N_{bitstream} - N_{wordTotal}$

$$t^{uncompressed}_{\text{preload}}(\text{ns}) = t_{\text{load}} + (latency + t_{\text{burst}} * \lfloor \frac{N_{\text{wordLeft}}}{N_{\text{burst}}} \rfloor + (t_{\text{burst}} - N_{\text{wordLeft}} \bmod N_{\text{burst}})) * T_{\text{bus}} \quad (3.4)$$

L'utilisation des versions compressées des bitstreams pose une nouvelle fois problème. En effet, le taux de compression des bitstreams n'est pas constant tout au long du bitstream et l'estimation était souvent faussée. Nous avons alors différencié les taux de compression pour les deux phases du préchargement. Ces taux partiels peuvent être obtenus avec notre compresseur en prenant comme paramètre le nombre de mots représentant la séparation entre les deux phases. Prenons les notations suivantes :

$ratio_1$: le taux de compression lors de la première partie du préchargement
$ratio_2$: le taux de compression lors de la seconde partie du préchargement
$t^{compressed}_{load}$: le temps passé lors de la première phase du préchargement en mode compressé, obtenu avec l'algorithme 1 pour un taux de compression $ratio_1$

Les équations 3.5 et 3.6 nous donnent respectivement une borne inférieure et une borne supérieure du temps de reconfiguration dans le mode de préchargement avec compression :

$$t^{compressed}_{\text{preload,min}}(\text{ns}) = t^{compressed}_{\text{load}} + (latency + t_{\text{burst}} * \lfloor \frac{ratio_2 * N_{\text{wordLeft}}}{N_{\text{burst}}} \rfloor + (t_{\text{burst}} - (ratio_2 * N_{\text{wordLeft}}) \bmod N_{\text{burst}})) * T_{\text{bus}} \quad (3.5)$$

$$t_{\text{preload,max}}^{compressed}\,(\text{ns}) = t_{\text{preload,min}}^{compressed} + N_{\text{wordLeft}} * (1 - ratio_2) * T_{\text{icap}} \tag{3.6}$$

Toutefois, les entrées de ces équations que sont $ratio_1$ et $ratio_2$ ne sont pas connues initialement puisque leur calcul nécessite de connaître le nombre de mots écrits durant la première phase du préchargement, qui dépend lui de $ratio_1$. Il est donc nécessaire de fixer une valeur initiale arbitraire aux deux taux de compression partiels, qui sera le taux de compression global du bitstream. On itère alors en alternant l'utilisation de l'algorithme 1 et de l'équation 3.5 jusqu'à obtenir des valeurs stables pour les taux de compression et donc pour le temps de reconfiguration.

3.2.3 Coût du mode de relecture

Nous avons vu que le mode de relecture était séparé en trois phases, dont la première et la dernière consistait en des accès en écriture sur l'ICAP dont nous pouvons estimer le coût temporel grâce aux équations (3.1), (3.2) et (3.3). Il reste donc à estimer le temps passé à effectivement relire les données de configuration depuis l'ICAP. Dans ce cas, le comportement est similaire à celui d'une écriture et c'est encore une fois le transfert sur le bus qui est un facteur limitant. La seule différence réside dans la latence initiale de la relecture. Il ne s'agit pas simplement ici de la latence de l'accès bus puisqu'il faut également prendre en compte la latence introduite par la lecture de la frame de remplissage au début de la lecture. Même si cette frame n'est pas transmise sur le bus, elle consomme tout de même 41 cycles pour un FPGA Virtex-5. Prenons les notations suivantes :

t_{w1} : temps requis pour l'écriture des données de configuration de la relecture (phase 1)
t_{w2} : temps requis pour l'écriture des données finales de la relecture (phase 3)
N_{data} : la quantité de données à relire
$framesize$: la taille d'une frame pour la technologie choisie

Nous obtenons donc l'équation 3.7 qui donne le temps global de relecture pour un nombre de frames donné :

$$\begin{aligned} t_{\text{read}}\,(\text{ns}) = t_{w1} + t_{w2} + framesize * T_{\text{icap}} + (latency + t_{\text{burst}} * \lfloor \frac{N_{\text{data}}}{N_{\text{burst}}} \rfloor + \\ (t_{\text{burst}} - N_{\text{data}} \bmod N_{\text{burst}})) * T_{\text{bus}} \end{aligned} \tag{3.7}$$

3.3 Application

FaRM a été développée dans un premier temps pour les FPGA Virtex-5 en utilisant le standard de bus PLB. Depuis, FaRM a été portée pour les FPGA Virtex-6 et serait a priori compatible avec les récents FPGA de la série 7 capable de reconfiguration dynamique (Virtex-7 et Kintex-7). Un portage AXI a aussi été réalisé puisque Xilinx a choisi ce standard pour les générations suivantes. Les tests mentionnés dans cette partie ont été réalisés sur une carte de prototypage Xilinx ML507, intégrant un FPGA vfx70t.

Les tests de FaRM ont été réalisés avec une application d'encodage/décodage AES et de transformée de Fourier rapide 64 bits (FFT, *Fast Fourier Transform*) comme décrite en

figure 3.11. Une zone reconfigurable a été définie pour chacune des applications afin que ces zones aient la même interface physique. Le sous-système de configuration est celui décrit dans la section 3.1.1 page 34. Nous comparerons les performances de FaRM avec celles du contrôleur de Xilinx HwICAP. Enfin, nous avons ajouté un contrôleur de mémoire Compact Flash, appelé SystemACE, une mémoire non-volatile où seront stockés les bitstreams avant la mise sous tension de la carte. Au démarrage du système, les bitstreams seront copiés depuis la mémoire Compact Flash vers la mémoire DDR, volatile.

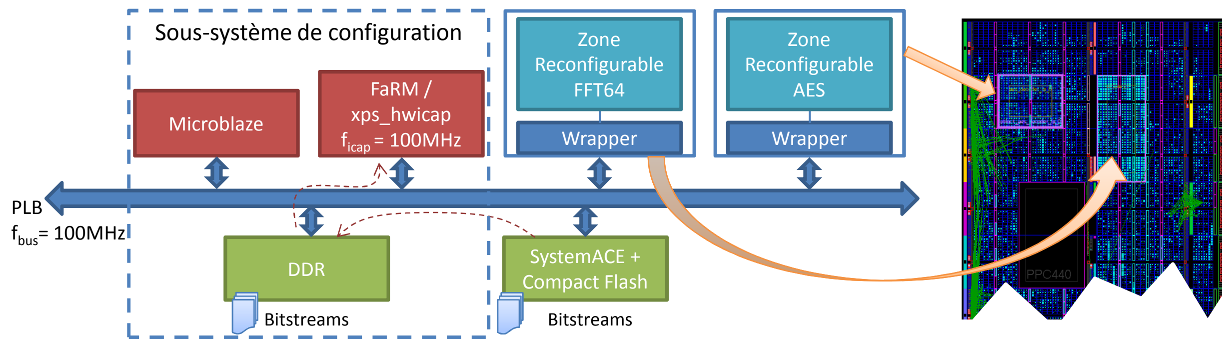


FIGURE 3.11 – *Architecture de l'application de test AES/FFT64*

Le tableau 3.2 décrit les ressources contraintes par les zones reconfigurables en respectant la granularité de reconfiguration. Ces zones reconfigurables ont été définies telles que les ressources sont au plus proche des besoins de chaque tâche. Toutefois, comme les besoins peuvent être différents pour l'encodeur et le décodeur de la même fonction (c'est le cas de l'AES), une partie des ressources de la zone reconfigurable sont gâchées lorsque l'encodeur est implémenté. Au niveau du temps de reconfiguration, cette différence de ressources par rapport à une zone optimale est en grande partie masquée par l'utilisation de la compression. En effet, moins la zone est occupée, moins ses ressources seront configurées, plus le taux de compression sera important. Enfin, comme nous l'avons mentionné précédemment, la relecture des BRAM ne donne pas de données significatives. Nous avons donc préféré les écarter lors de la relecture, ce qui fait que nous ne travaillons plus que sur un sous-ensemble de 462 frames pour l'AES.

TABLEAU 3.2 – *Ressources contraintes par les zones reconfigurables*

Zone reconfigurable	Colonnes de CLB	Colonnes de DSP	Colonnes de BRAM	Nombre total de slices	Taille du bitstream (en mots de 32 bits)
AES	12	-	1	590	24 419
FFT64	16	2	2	952	39 218

Les tests ont été effectués avec plusieurs contrôleurs de reconfiguration différents. Nous avons ainsi testé les versions de FaRM avec et sans compression. Le mode de préchargement fera l'objet d'une étude plus poussée dans la section 3.3.3. Nous avons également testé le contrôleur HwICAP de Xilinx. Nous l'avons également couplé à un DMA afin de nous rapprocher de l'architecture de FaRM.

3.3.1 Evaluation des performances

Les résultats obtenus pour la zone reconfigurable AES avec l'implémentation de l'encodeur sont rapportés dans le tableau 3.3. Les mesures effectuées sont les suivantes :

- Reconfiguration : une écriture de bitstream classique.
- Relecture frame/frame : une relecture effectuée frame par frame. Cette mesure est nécessaire pour pouvoir comparer les performances avec l'HwICAP qui n'est pas capable de lire plus d'une frame à la fois.
- Relecture globale : une relecture directe de l'intégralité des données de configuration de la zone reconfigurable, uniquement avec FaRM.

Nous avons distingué le débit sur le bus du débit de l'ICAP. La distinction est nécessaire lors de l'utilisation de FaRM avec la compression ou en mode relecture. En mode écriture avec compression, la quantité de données transférée sur le bus correspond à la taille du bitstream compressé alors que la quantité de données effectivement écrite dans l'ICAP correspond à la taille du bitstream décompressé. En mode relecture, FaRM ne transmet pas la frame de remplissage lue au début du processus de relecture. Dans le cas d'une relecture frame par frame, le débit sur le bus est ainsi réduit de moitié par rapport au débit de l'ICAP.

Tableau 3.3 – *Temps de reconfiguration et de relecture*

f_{bus} = 100MHz f_{icap} = 100MHz	HwICAP			HwICAP + DMA		
	Temps (μs)	Débit COM (Mo/s)	Débit ICAP (Mo/s)	Temps (μs)	Débit COM (Mo/s)	Débit ICAP (Mo/s)
Reconfiguration	72800	1.29	1.29	1550	60.5	**60.5**
Relecture frame/frame	159000	0.92	0.92	49200	2.97	2.97
Relecture globale	-	-	-	-	-	-
	FaRM sans compression			FaRM avec compression		
Reconfiguration	731	128	**128**	538	127	**174**
Relecture frame/frame	1536	47	95	1600	45	91
Relecture globale	560	129	129	560	129	129

Ces résultats mettent bien en avant le gain de performances offert par FaRM dont la version sans compression est déjà deux fois plus rapide que la solution HwICAP couplé à un DMA. En utilisant la compression, nous obtenons un débit de 174 Mo/s au niveau de l'ICAP qui est assez éloigné de son débit théorique de 400 Mo/s (en fonctionnant à une fréquence de 100 MHz). Cela met bien en évidence la limitation due à la lecture d'une mémoire externe, mais peu coûteuse. Nous verrons dans la section suivante comment le préchargement nous permet d'atteindre le débit théorique de l'ICAP.

Avec FaRM, les résultats obtenus pour une relecture avec ou sans compression sont similaires. En fait, la compression n'est pas du tout utilisée dans le cas de la relecture. Pour les phases de la relecture qui nécessitent une écriture de paramètres de configuration vers l'ICAP (phases 1 et 3 de la figure 3.6 page 40, il n'y a que très peu de données à écrire (pas plus de vingt mots) en plus d'un gain dû à la compression très faible vu qu'il n'y a quasiment aucune redondance dans ces séquences de données. Pour la phase de lecture effective (phase 2 de la figure), la limitation vient de l'algorithme de compression qui n'est pas implémenté en ma-

tériel. Il est d'ailleurs difficilement réalisable si l'on souhaite obtenir de bonnes performances puisque l'algorithme doit parcourir l'intégralité du fichier à compresser pour récupérer les données d'offset. La latence avant de pouvoir transférer le fichier pourrait être désastreuse et incompatible avec des contraintes de temps assez fortes : dans le cas (extrême) d'un fichier avec très peu voire aucune compression, le compresseur devrait attendre de rencontrer la première donnée redondante avant de pouvoir démarrer le transfert de toutes les données accumulées jusque là. Nous sommes conscients des limites que cela représente. Pour éviter cela, il serait possible de donner une valeur limite à cet offset, par exemple la même que la taille de la FIFO. La latence serait ainsi limitée tout en conservant un gain non négligeable en nombre de transferts de données. Par exemple, en considérant un taux de compression moyen de 26.7%, la latence induite par le remplissage de la FIFO serait compensée par la compression pour des relectures de taille supérieure à 3835 mots de 32 bits, soit environ 15 ko. A titre de comparaison, parmi les bitstreams utilisés pour les tests de compression listés dans le tableau 3.1, seuls deux bitstreams ont une taille inférieure à ce seuil. Une solution consisterait à implémenter un autre algorithme de compression pour le mode de relecture plus simple et plus efficace, par exemple un algorithme RLE canonique.

3.3.2 Overclocking de l'ICAP

Nous avons également procédé à des tests en poussant la fréquence de l'ICAP à 200 MHz. La fréquence du bus PLB a aussi été augmentée à 125 MHz afin de limiter son effet sur les performances. Dans cette configuration, les résultats obtenus auraient été similaires à ceux avec une fréquence de 100 MHz pour l'ICAP. En effet, si l'ICAP est capable d'assimiler toutes les données issues du bus à 100 MHz, il le sera aussi à 200 MHz. Nous avons donc couplé ce changement de fréquence à l'utilisation du mode de préchargement, où les performances maximales de l'ICAP sont atteintes durant la première phase du préchargement.

Les bitstreams de test sont cette fois-ci les bitstreams appelés *Count1* et *Count2* dans la table 3.1 en page 38. En théorie, l'ICAP aurait besoin de 1658 cycles pour écrire complètement les 6632 octets de ces bitstreams décompressés, soit 8290 nanosecondes. En pratique, nous obtenons un temps de reconfiguration de 8304 nanosecondes ce qui signifie que nous avons bien atteint le débit maximal de l'ICAP en fonctionnement à 200 MHz.

3.3.3 Influence de la FIFO sur le préchargement des bitstreams

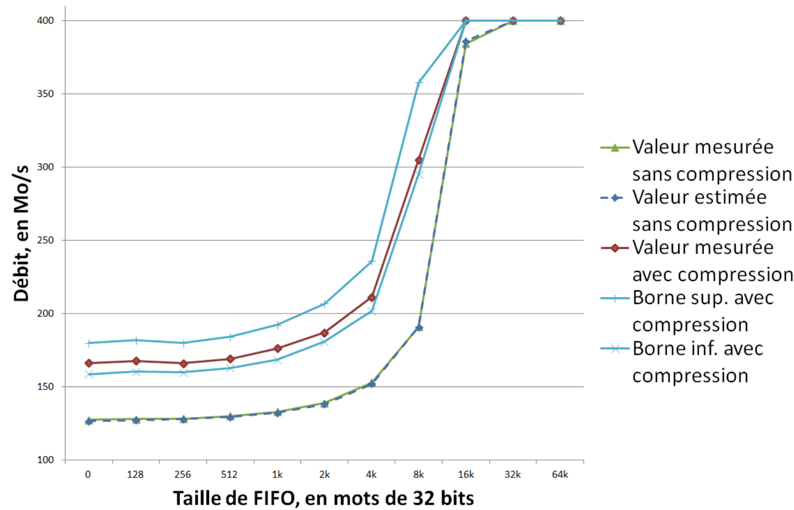
Dans cette partie, nous allons nous intéresser au mode de préchargement des bitstreams. Nous mesurerons ses performances et comparerons ces valeurs avec les estimations fournies par notre modèle de coût. La figure 3.12 montre les résultats obtenus ainsi que les estimations issues du modèle de coût avec différentes tailles de FIFO pour l'IP AES (3.12(a)) ainsi que pour la FFT (3.12(b)). Le point correspondant à une taille de FIFO nulle représente l'utilisation du mode d'écriture classique, sans préchargement. Les estimations ont été réalisées avec les valeurs suivantes :

latence : 10 cycles d'horloge, obtenu avec Chipscope Pro

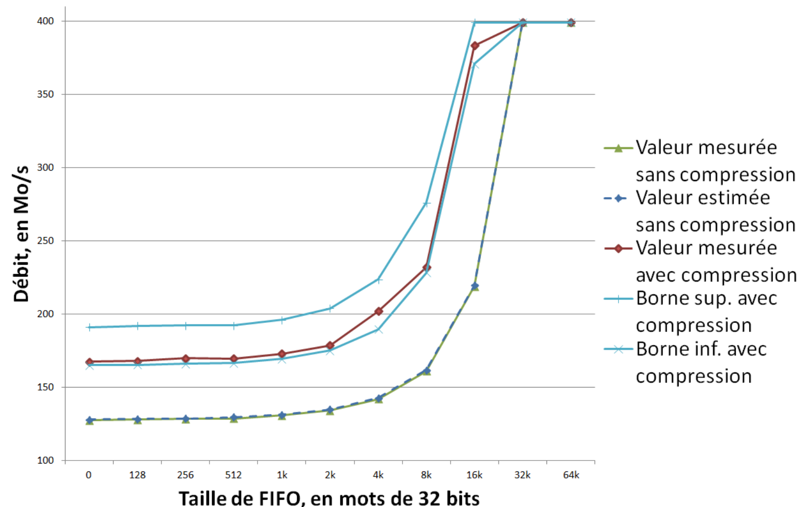
N_{burst} : 16 mots par accès rafale

t_{burst} : 50 cycles pour effectuer un accès rafale, obtenu avec Chipscope Pro

T_{bus} & T_{icap} : 10 nanosecondes



(a) Résultats pour l'AES



(b) Résultats pour la FFT

FIGURE 3.12 – *Influence de la taille de la FIFO sur le débit de l'ICAP*

Nous pouvons voir sur ces graphiques que pour atteindre le débit maximal de 400 Mo/s pour la reconfiguration de l'IP AES, la taille de FIFO requise est de 16000 mots, avec ou sans utilisation de la compression. Ce résultat dépend bien sûr de la taille du bitstream et du taux de compression réalisé : dans le cas de l'IP FFT, la taille de FIFO nécessaire à l'obtention du débit maximal est de 32000 mots.

La figure montre également que notre modèle de coût propose des estimations du temps de reconfiguration très proches de la réalité. Dans le cas d'une utilisation sans compression, les courbes obtenues sont superposées. Pour ce qui est d'une utilisation de FaRM avec compression, la valeur réelle reste bien dans le gabarit défini par notre modèle, en se rapprochant particulièrement de la limite basse en termes de débit : la compression se situe donc bien à la fin du bitstream. Ce résultat était prévisible car nous savons que l'initialisation des mémoires

se fait toujours à la fin du bitstream et nos tâches utilisaient chacune des mémoires BRAM (deux pour la FFT, ce qui explique le rapprochement plus prononcé avec la borne inférieure). Ainsi, pour des IP nécessitant des blocs mémoire ou instanciés sur des zones reconfigurables contraignant des blocs mémoires, l'estimation la plus juste serait réalisée en prenant la borne inférieure du temps de reconfiguration estimé donnée par l'équation 3.5.

Nous avons vu que pour atteindre le débit théorique du port de configuration, les besoins en BRAM sont assez important et peuvent être prohibitifs, particulièrement sur des FPGA de type Virtex-5 qui offrent assez peu de ressources mémoire. Le concepteur doit bien souvent faire un compromis entre les performances du système et le coût en termes de ressources. Pour cela, nous avons estimé le nombre de mots écrits au débit maximum (en phase 1 du mode préchargement, avant que les accès mémoire redeviennent le facteur limitant) en fonction du taux de compression réalisé et de la taille de FIFO, le tout pour une taille de bitstream fixée (ici la taille du bitstream AES). Cela signifie que ces résultats sont utilisables pour toute IP implémentée sur une même zone reconfigurable. Les résultats sont illustrés par la figure 3.13.

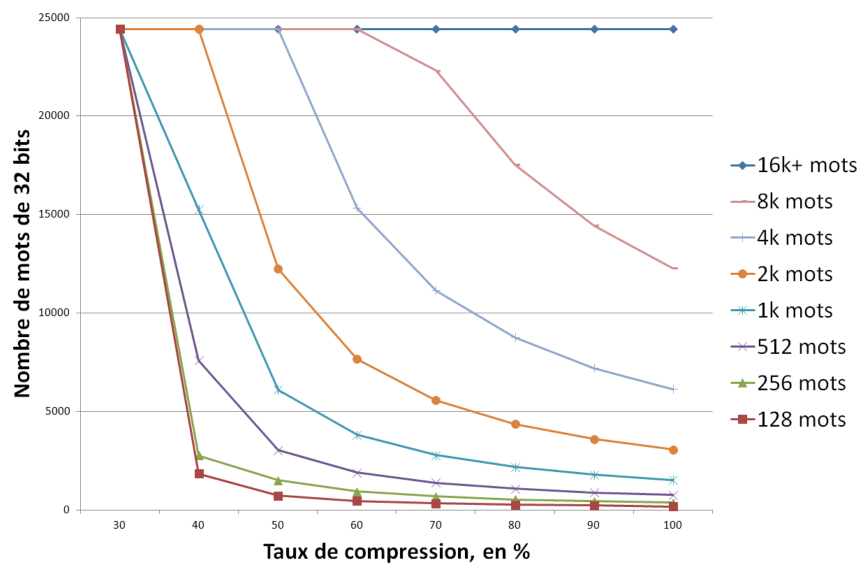


FIGURE 3.13 – *Nombre de mots écrits au débit maximal*

Assez logiquement, cette quantité de données augmente avec le taux de compression. Nous remarquons que pour une taille de FIFO supérieure ou égale à 16000 mots, soit 64 ko, le bitstream peut être transféré intégralement, indépendamment du taux de compression réalisé. Cela signifie que tous les bitstreams du tableau 3.1 en page 3.1 à l'exception de la FFT64 pourront être transférés au débit maximal avec une telle taille de FIFO. Nous pouvons généraliser ce résultat en disant qu'une taille de FIFO de 64 ko serait suffisante dans la plupart des cas pour tirer parti des performances maximales de la reconfiguration dynamique avec un coût mémoire très faible si l'on considère la famille de FPGA Virtex-6, dont les composants offrent entre 5 et 32 Mo de mémoire BRAM [72].

3.4 Positionnement de FaRM

Dans cette section, nous allons nous intéresser au positionnement de FaRM par rapport à certains contrôleurs mentionnés pendant l'analyse de l'existant. En particulier, intéressons-nous au contrôleur HwICAP de Xilinx. Même si nous avons mis en avant ses faibles performances, il reste le contrôleur utilisé de base dans les designs reconfigurables dynamiquement et représente donc la première expérience utilisateur de la technologie. Le tableau 3.4 propose une comparaison des fonctionnalités offertes par FaRM et l'HwICAP ainsi que les ressources nécessaires à leur implémentation sur un FPGA de la technologie Virtex-5.

Tableau 3.4 – *Comparaison de FaRM avec l'HwICAP*

	xps_hwicap		FaRM	
Ecriture simple	Faibles performances		Débit maximal	
Relecture	Frame par frame uniquement		Taille variable, automatisé	
Relecture & capture	Par registre		Capture automatique	
Compression de bitstream	Non		Oui	
Mode préchargement	Par interface esclave		Automatique	
Interfaces	Esclave uniquement		Maitre/Esclave	
DMA	Non		Oui	
Ressources	HwICAP	HwICAP + DMA	Complet	O-RLE seul
- Slice registers	427	991	1130	243
- Slice LUTs	1748	2455	1342	357
- LUT Flip Flop	1216	2072	1786	390
- BRAM	2	2	2	-

Avec des besoins en ressources plutôt équivalent, FaRM propose des performances bien plus conformes à une utilisation dans un cadre temps-réel strict que le contrôleur originel de Xilinx. En comparaison des contrôleurs mentionnés dans l'analyse de l'existant, les performances de FaRM ont déjà été réalisées, notamment par les auteurs de [2]. Nous pensons que notre approche est plus complète et fournit une solution non seulement performante, mais également très facile à utiliser grâce à la couche logicielle développée pour une utilisation *standalone* (sans système d'exploitation) ou couplée à un système GNU/Linux [73]. Notre approche se distingue également par le mode de relecture, même s'il n'est pas complètement opérationnel à cause de la technologie, mais surtout par le modèle de coût du temps de reconfiguration. En effet, l'optimisation des temps de reconfiguration n'est utile que dans les cas ou les contraintes de temps du système sont critiques. Il est alors nécessaire de coupler l'utilisation de la reconfiguration dynamique avec un ordonnancement des tâches matérielles, dont l'étude requiert la connaissance des temps de reconfiguration afin d'être représentative du comportement réel du système.

Pour ce qui est du contrôleur UPaRC [5], développé postérieurement à FaRM, la compression est effectivement très intéressante et meilleure que celle utilisée au sein de FaRM. Toutefois, cela vient aux dépends d'une utilisation de ressources bien plus importante (pour rappel, plus de 1000 slices en technologie Virtex-5, un slice contenant 4 LUT et 4 registres).

Encore une fois, il s'agit d'un compromis à réaliser entre performances et ressources pour lequel nous avons choisi de privilégier l'aspect ressources. Néanmoins, l'overclocking de l'ICAP à des fréquences supérieures à 200 MHz est une idée intéressante si le système utilise le mode de préchargement. Dans le cas inverse, l'ICAP n'est pas le facteur limitant et l'overclocking ne serait alors qu'une source de consommation supplémentaire.

3.5 Conclusion

Dans ce chapitre, nous avons décrit les moyens mis en œuvre au sein de l'IP FaRM (Fast Reconfiguration Manager) dans le but de réduire les temps de reconfiguration. FaRM s'appuie ainsi sur une architecture optimisée pour les performances avec une interface maître lui permettant d'être autonome pour les accès aux bitstreams de configuration. FaRM intègre également une FIFO locale lui permettant de précharger les bitstreams afin d'atteindre le débit théorique du port de reconfiguration de 400 Mo/s et même 800 Mo/s grâce à l'overclocking de l'ICAP. Nous avons également développé un algorithme de compression sur la base du RLE, appelé O-RLE, permettant de réduire la taille des bitstreams et donc de réduire le nombre d'accès effectués sur le bus, qui reste le facteur limitant du temps de reconfiguration. Enfin, nous proposons au sein de FaRM un mode de relecture automatique qui permet de facilement récupérer les données de configuration d'une zone reconfigurable. Les tests effectués sur une application AES et FFT mettent en avant le gain en performances très important en comparaison du contrôleur de Xilinx, avec un débit ICAP allant jusqu'à 174 Mo/s en compression.

Nous avons également développé un modèle de coût du temps de reconfiguration. Ce modèle, dont nous avons démontré la précision, permet de connaître le temps nécessaire pour le chargement d'un bitstream dans chacun des modes de fonctionnement de FaRM. Ce modèle nous permet donc de mettre en place des politiques d'ordonnancement des tâches matérielles prenant en compte le temps de reconfiguration comme nous le montrerons dans le chapitre suivant.

Les travaux sur FaRM ont donné lieu aux publications [74] et [75].

CHAPITRE 4

RecoSim : un simulateur d'architectures reconfigurables dynamiquement pour des applications temps-réel

Dans ce chapitre, nous présenterons l'élément central de notre méthodologie, RecoSim pour *Reconfiguration Simulator*. RecoSim est un simulateur d'architectures reconfigurables dynamiquement hautement paramétrable pour la validation d'applications temps-réel. Basée sur la librairie open-source SystemC, la modélisation de ces applications est effectuée à un haut niveau d'abstraction pour permettre des études préalables durant les premières phases du processus de conception du système. Pour cela, RecoSim utilise la modélisation transactionnelle (TLM) pour abstraire les communications entre modules et ainsi réduire drastiquement le temps de simulation. RecoSim offre également la possibilité de développer et de tester les algorithmes d'ordonnancement qui seront utilisés lors de l'implémentation du système sur carte. Afin de valider RecoSim, nous avons utilisé une application de transmission sécurisée de flux vidéo ainsi qu'une application de transcodage vidéo H.264/MPEG-2, deux applications aux contraintes temps-réel fortes. Nous montrerons ainsi que l'utilisation de RecoSim permet de déterminer rapidement si une solution reconfigurable dynamiquement est possible et surtout avantageuse en termes de surface par rapport à une solution statique.

4.1 L'approche RecoSim

Dans cette section, nous présenterons l'approche de RecoSim ainsi que les concepts utilisés. Nous présenterons le module reconfigurable, brique de base du simulateur ainsi que le gestionnaire de reconfiguration qui centralise l'intelligence du système. Enfin, nous évoquerons l'utilisation de la modélisation transactionnelle pour représenter les communications au sein du simulateur.

4.1.1 Approche en Y de la modélisation

La modélisation au sein de RecoSim est basée sur une approche en Y, comme le montre la figure 4.1. Cette approche se base sur une séparation de la modélisation de la partie applicative et de la partie architecturale du système. Cette approche tend à faciliter la modélisation du système en le séparant en deux parties indépendantes. Ces deux modèles sont alors fusionnés pour former un modèle alloué, qui représente notamment le placement des différentes unités de l'application sur les composantes de l'architecture.

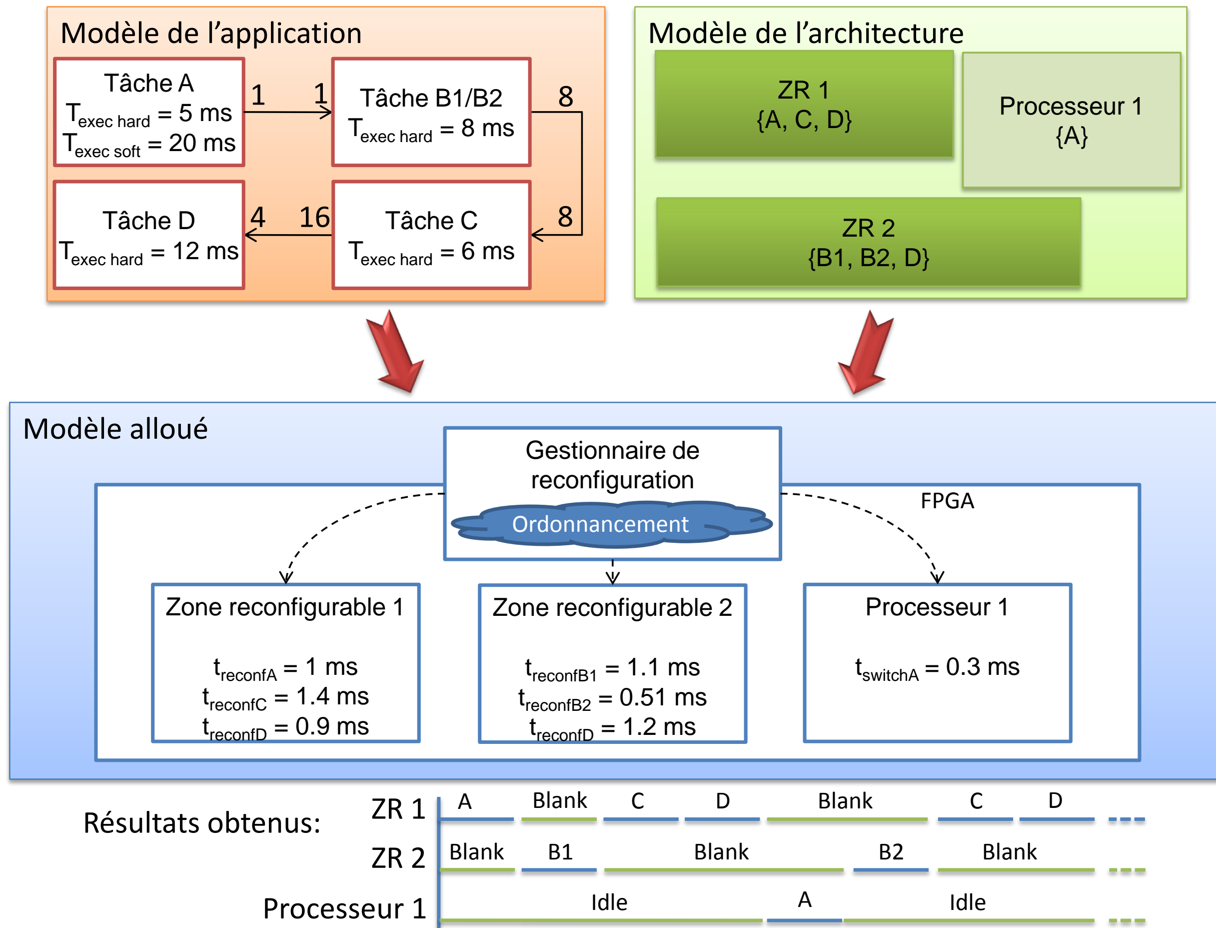


FIGURE 4.1 – *Approche en Y de la modélisation avec RecoSim*

Le modèle de l'application consiste en un graphe orienté acyclique (de l'anglais *Directed Acyclic Graph*, DAG) qui représente l'application statique avec de possibles dépendances de données. Chaque nœud contient des informations sur les temps d'exécution de la tâche selon la cible considérée, par exemple une zone reconfigurable (matérielle) ou un processeur particulier (logiciel). Dans notre exemple, toutes les tâches peuvent êtres implémentées matériellement sur le FPGA, donc sur une zone reconfigurable, et la tâche A a également la possibilité de s'exécuter sur un processeur. Les nœuds contiennent également des informations concernant les communications entre les modules : il est ainsi possible de configurer la quantité de données entrantes et sortantes pour la tâche ainsi que le temps nécessaire à la récupération de ces données. Chaque nœud peut également être défini comme statique, c'est-à-dire qu'il

sera instancié à tout moment sur le FPGA. Dans ce cas, il ne peut y avoir qu'une seule implémentation pour la tâche, matérielle ou logicielle.

Le modèle de l'architecture correspond à l'ensemble des unités de traitement à la disposition du simulateur, c'est-à-dire l'ensemble des zones reconfigurables et des processeurs disponibles pour le placement des tâches. Ce modèle ne fait pas apparaître le sous-système de configuration tel que nous l'avons défini dans la section 3.1.1 page 34. Nous nous sommes placés dans une approche où l'architecture est dépendante de l'application plutôt que dans le cas d'une plate-forme générique comme nous l'avons vu dans l'analyse de l'existant. Ainsi, les zones reconfigurables sont définies à partir de l'application. Le concepteur doit donc définir les possibilités de placement des tâches, c'est-à-dire quelles tâches pourront être placées sur quelles zones reconfigurables. Cette étape d'allocation des ressources aux tâches de l'application permet ainsi de définir la taille de la zone reconfigurable, qui doit pouvoir accueillir toutes les tâches qui lui ont été assignées. Dans l'exemple de la figure précédente, la zone reconfigurable 1 (ZR 1) peut accueillir les tâches A, C et D alors que la seconde zone reconfigurable peut accueillir les tâches B1, B2 et D. L'unique processeur présent dans la partie dynamique de l'architecture peut quant à lui accueillir la tâche A. Notons que les zones reconfigurables définies par l'utilisateur ne sont pas placées sur le FPGA et n'assurent donc en aucun cas la faisabilité de la solution sur un composant. Pour s'en assurer, il incombe au concepteur de vérifier manuellement la validité du placement des zones sur le FPGA ciblé au moyen d'outils de floorplanning comme Xilinx PlanAhead.

Le modèle architectural comporte également des informations sur les temps de reconfiguration des différentes tâches en fonction des zones reconfigurables. Comme nous l'avons vu dans le chapitre précédent, la compression permet de réduire la taille des bitstreams de configuration qui peuvent alors être différents pour une même zone reconfigurable avec différentes tâches. Il y a donc autant de temps de reconfiguration à calculer que de paires tâches/zones reconfigurables. Dans notre cas, ces temps de reconfiguration sont donnés par le modèle de coût proposé dans la section 3.2 en page 42. Pour un processeur, ce temps le temps de reconfiguration correspond à un temps de basculement depuis la tâche précédente vers la nouvelle tâche dont l'exécutable doit être chargé en mémoire.

Une fois ces deux modèles définis, le simulateur peut être utilisé pour définir le modèle alloué du système. Ce modèle représente le comportement du système pendant la simulation, c'est-à-dire comment et à quel moment sont placées les tâches sur les zones reconfigurables. La simulation se base sur l'algorithme d'ordonnancement écrit par le concepteur afin d'effectuer le placement des tâches sur les zones reconfigurables. A chaque fois qu'un paquet de données est traité, le simulateur vérifie si les contraintes de temps ont été respectées. RecoSim gère les contraintes de temps dures (de l'anglais *Hard Real Time*, HRT) où les contraintes de temps doivent toujours être respectées ainsi que les contraintes de temps souples (*Soft Real Time*, SRT) où il est possible de ne pas respecter toutes les contraintes de temps du moment qu'une certaine qualité de service est respectée (c'est le cas par exemple du streaming de flux vidéo, où la perte de paquets n'est pas critique même si elle peut devenir gênante). Pour assurer qu'un système respecte ces contraintes de temps, il est nécessaire d'effectuer une simulation sur un temps qui est au moins aussi grand que l'hyperpériode. Dans le cas de tâches périodiques, l'hyperpériode est définie comme le plus petit commun multiple (PPCM) des périodes des

tâches [76]. Il faut également veiller à ce que le régime permanent du système soit atteint, c'est-à-dire que toutes les tâches doivent s'exécuter simultanément. Notons $WCET(T_i)$ le temps d'exécution de la tâche i dans le pire cas (Worst Case Execution Time). Nous obtenons ainsi l'équation 4.1 pour le temps de simulation minimal du système :

$$t_{\text{simulation,min}} = PPCM\left(T_0..T_n\right) + \sum_{i=1}^{n} WCET\left(T_i\right) \tag{4.1}$$

Dans le cas où le temps de simulation est inférieur à cette valeur, la capacité du système à respecter ses contraintes de temps ne peut pas être assurée et il advient au concepteur de modifier le banc de test en conséquence.

Notons que pour le moment, la qualité de service est calculée comme le rapport entre le nombre d'exécutions de tâches respectant leur échéance et le nombre d'exécutions globales. Nous ne faisons donc pas la distinction entre différentes applications potentiellement indépendantes au sein du système qui pourraient avoir des spécifications de qualité de service différentes. De la même manière, le calcul de la qualité de service obtenue est effectué sur le temps de simulation global et donc pas forcémenent sur un nombre entier d'hyperpériodes, ce qui peut fausser le calcul. RecoSim sera modifié en conséquence dans les prochaines versions pour supprimer ces limitations.

4.1.2 Définition du module reconfigurable

4.1.2.a Description globale

Dans cette partie, nous nous intéressons au module reconfigurable, élément de base de RecoSim. Un tel module représente la tâche statique ainsi que tous les mécanismes nécessaires pour prendre en compte la reconfiguration dynamique. Les modules sont instanciés à tout instant de la simulation, comme nous avons vu qu'il est impossible de modifier les modules SystemC ou les connexions entre eux en cours de simulation. Toutefois, ils ne sont actifs que si le gestionnaire de reconfiguration les a placés au préalable sur le FPGA. Le comportement des architectures reconfigurables dynamiquement est ainsi correctement modélisé sans pour autant modifier les modules en cours de simulation, donc sans modifier le noyau de simulation de SystemC.

L'architecture d'un tel module est illustrée dans la figure 4.2. Comme nous pouvons l'observer, chaque module est connecté au gestionnaire de reconfiguration qui possède toute l'intelligence et prends les décisions relatives à l'ordonnancement des tâches de l'application, aussi bien logicielles que matérielles. Ainsi, la communication avec le gestionnaire de reconfiguration permet par exemple au module d'être exécuté uniquement si la tâche est effectivement implantée sur le FPGA (que ce soit sur une zone reconfigurable ou sur un processeur). Nous développerons plus précisément le fonctionnement du gestionnaire dans la section suivante.

Le module reconfigurable est construit autour de deux principales caractéristiques : l'utilisation de threads dynamiques et l'utilisation de la modélisation transactionnelle grâce à des sockets TLM au niveau des connexions avec les autres modules.

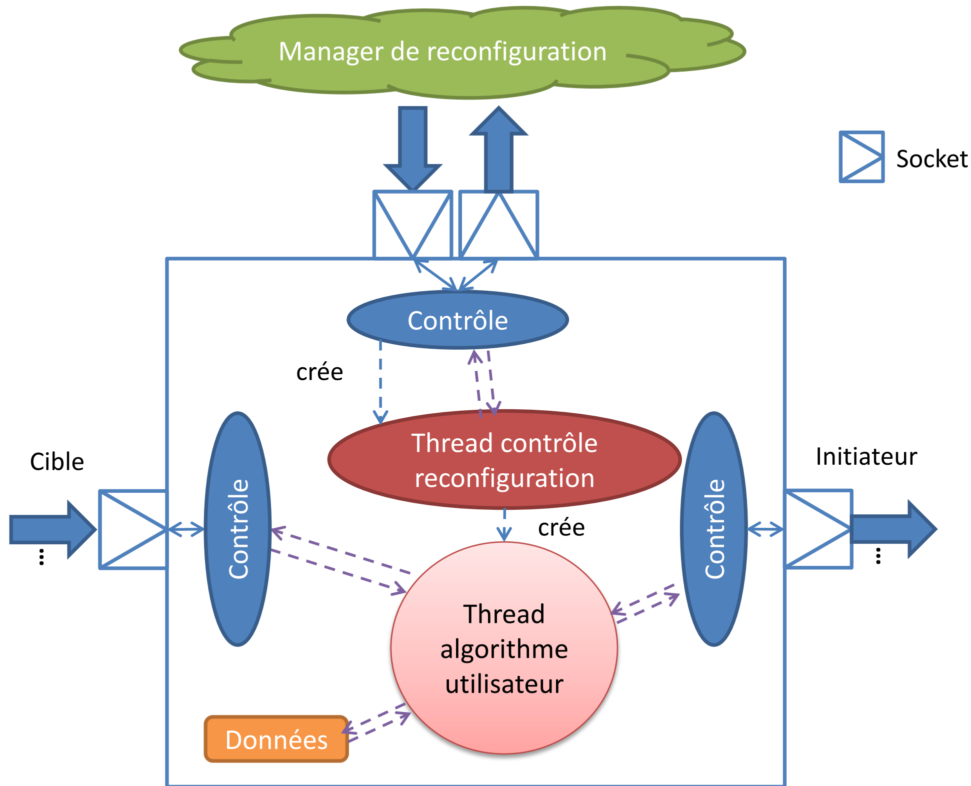


FIGURE 4.2 – *Architecture d'un module reconfigurable*

4.1.2.b Utilisation de threads dynamiques

Comme nous l'avons indiqué lors de l'analyse de l'existant, l'utilisation de threads dynamiques, fonctionnalité introduite avec SystemC 2.0, permet de facilement remplacer une fonction par une autre. Ces threads étaient utilisés afin de modifier le comportement d'une zone reconfigurable en cours de simulation. Nous avons choisi une approche un peu différente : en effet, nous considérons simplement un ensemble de modules ayant un comportement prédéfini (correspondant à la tâche à effectuer) que nous allons affecter à une zone reconfigurable. Les threads dynamiques sont alors utilisés pour modifier le comportement d'une tâche en cours de simulation (et non pour modéliser l'aspect partage des ressources de la reconfiguration dynamique). À des fins de débogage, les threads dynamiques permettent également de changer le type d'algorithme (RTL, SystemC ou juste un délai modélisant le temps d'exécution). La simulation RTL étant très longue, il est possible d'utiliser un mode de simulation plus rapide jusqu'à un certain temps de simulation où a été identifié un problème, puis de commuter vers le mode de simulation RTL pour bénéficier de sa précision sans pour autant avoir perdu trop de temps pour la simulation de parties déjà validées.

Chaque module reconfigurable utilise deux threads dynamiques. Le premier, appelé *Thread algorithme utilisateur* sur la figure 4.2, représente exactement le comportement de la tâche. C'est lui qui sera modifié lorsque l'utilisateur souhaite basculer vers un autre comportement ou un autre mode de fonctionnement. Pour cela, il est contrôlé par le second thread dynamique (*Thread contrôle reconfiguration* de la même figure) qui reçoit ses ordres du gestionnaire de reconfiguration. Ce thread est créé durant la phase d'élaboration du module et

pour cela, il aurait été possible de le définir de manière statique. Nous avons toutefois préféré le définir de manière dynamique afin de pouvoir l'isoler complètement de la classe et laisser ainsi le soin au concepteur de modifier son comportement sans avoir à modifier la classe du module reconfigurable. L'interface de programmation (en anglais API pour *Application Programming Interface*) proposée pour le thread utilisateur permet simplement d'accéder aux données récupérées depuis les modules précédents ainsi qu'aux données à envoyer vers les modules suivants. L'API fournit également un certain nombre de signaux de contrôle pour démarrer le thread ou pour notifier la fin de son exécution.

Le thread représentant le comportement du module peut être défini à différents niveaux d'abstraction. Pour diminuer le niveau d'abstraction d'un algorithme, il est possible d'utiliser des outils de synthèse de haut niveau qui sont capables de transformer du code écrit en langage C, C++ ou SystemC vers du code RTL synthétisable pour FPGA. C'est le cas par exemple de Xilinx Vivado [77], la nouvelle version de la suite Xilinx qui permet la synthèse de haut niveau grâce à un outil basé sur AutoESL. Un algorithme peut alors être développé et validé dans un langage haut niveau plutôt qu'en RTL, réduisant de manière très importante le temps (et donc le coût) de développement d'une IP, pour être ensuite transformé en du code synthétisable. Xilinx annonce des performances rivalisant avec du code HDL écrit manuellement tout en étant optimisé pour l'architecture interne des FPGA Xilinx, ce qui se confirmer à la lecture de l'étude [78].

Notons que ces threads (à la fois pour le contrôle et l'algorithme utilisateur) ont la possibilité d'accéder à leur module parent par le biais d'une interface dédiée, leur fournissant par exemple les méthodes nécessaires à la récupération des données entrantes ou à l'écriture des données sortantes. Encore une fois, cela permet à ces threads d'être définis en dehors de la classe de base et ainsi de ne pas avoir accès à tous ses membres afin de bien séparer les différentes composantes du module reconfigurable pour les rendre plus facilement modifiables.

4.1.2.c Utilisation des sockets TLM

Afin de réduire le temps de simulation avec RecoSim et de nous affranchir de détails trop bas niveau du système, nous avons choisi d'utiliser la modélisation transactionnelle, fourni comme une extension de SystemC appelée TLM (pour *Transaction-Level Modeling*). Pour cela, la norme TLM fournit des objets appelés sockets qui permettent de faire communiquer des modules ensemble par le biais de transactions. Afin d'avoir des modules flexibles en termes d'entrées/sorties, nous avons utilisé les objets *multi initiator socket* et *multi target socket*, représentant respectivement des sockets initiateurs et cibles de transactions et qui ont la particularité de pouvoir être connectés à plusieurs sockets à la fois.

La description détaillée de la modélisation transactionnelle et des protocoles utilisés feront l'objet de la section 4.1.4.

4.1.3 Gestionnaire de reconfiguration

Comme nous l'avons déjà mentionné, le gestionnaire de reconfiguration concentre toute l'intelligence du simulateur et est responsable du placement et de l'ordonnancement des tâches matérielles et logicielles. Celui-ci est guidé par la connaissance du diagramme de tâches

de l'application, donc des dépendances de données entre les tâches, ainsi que du modèle alloué représentant les différentes affectations possibles pour chaque tâche avec les temps de reconfiguration correspondant. Pour les implémentations logicielles des tâches, nous prenons également en compte le temps de chargement de l'exécutable sur le processeur qui peut être dépendant de sa taille.

4.1.3.a Requête d'ordonnancement

L'algorithme d'ordonnancement est exécuté à chaque fois qu'une tâche termine son exécution. Le diagramme de séquences pour l'exécution de l'algorithme d'ordonnancement à la fin de l'exécution d'un module est donné en figure 4.3. Lorsque le thread utilisateur d'un module se termine, le module utilise le socket de communication vers le gestionnaire afin de lui notifier la fin de son exécution. Alors, le gestionnaire va chercher à instancier sur le FPGA tous les modules connectés à la suite du module dont l'exécution s'est achevée en utilisant les informations issues du graphe orienté acyclique de l'application. Le temps d'exécution de l'algorithme d'ordonnancement est paramétrable pour bien représenter la complexité de l'algorithme ou bien son implémentation (par exemple un module hardware ou une exécution sur un processeur). À cet instant, les tâches qui ne sont pas encore instanciées sur le FPGA sont placées dans la file d'attente du gestionnaire de reconfiguration. Il s'agit en fait d'une simple FIFO qui respecte l'unicité du port de configuration et interdit la reconfiguration de plusieurs tâches en même temps. Dès qu'un module est reconfiguré, le module ayant initié la requête de reconfiguration est notifié par le gestionnaire, ce qui lui permet de démarrer le transfert des données vers ce module. Dans le cas où plusieurs modules sont connectés à la suite du module ayant terminé son exécution, ce dernier envoie les données dès que possible vers chacun de ses successeurs, sans attendre que tous soient placés sur le FPGA.

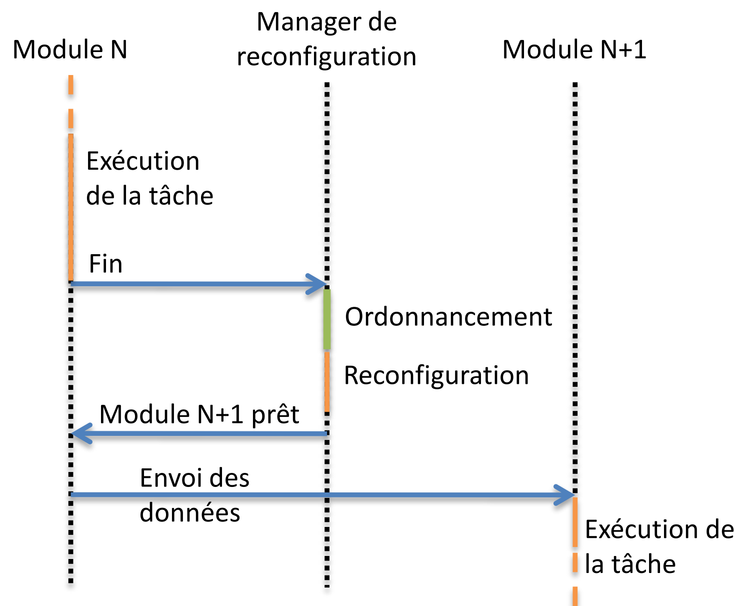


Figure 4.3 – *Diagramme de séquences pour la fin d'exécution d'un module*

4.1.3.b Placement et ordonnancement des tâches reconfigurables

Le placement des tâches reconfigurables sur des éléments de calculs comme des zones reconfigurables ou des processeurs est un problème très vaste pour le concepteur. L'objectif de cette thèse n'étant pas l'optimisation du placement en ligne des tâches mais plutôt l'aspect méthodologique du processus de conception, nous avons mis en place un placeur basique qui se base sur l'automate décrit dans la figure 4.4. Le FPGA est considéré vierge au démarrage de la simulation, ce qui fait que les tâches sont toutes dans l'état *En attente*. Lorsque l'implémentation d'une tâche est demandée, celle-ci va passer dans l'état *Configuration* et y restera pendant le processus de reconfiguration (y compris le temps passé en file d'attente pour accéder à l'ICAP), puis *Exécution*, signifiant que la tâche a été placée sur le FPGA et qu'elle peut désormais s'exécuter. Une fois son exécution terminée, la tâche va passer dans l'état *Placée* (tâche placée sur le FPGA mais non exécutée). Ainsi, lorsqu'une nouvelle exécution de la tâche est demandée, celle-ci pourra s'exécuter directement (passant ainsi dans l'état *Exécution*) sans nécessiter une quelconque reconfiguration. Par contre, si une tâche d'une priorité supérieure doit être placée par le gestionnaire, une tâche dans l'état *Placée* peut être évincée du FPGA, c'est-à-dire remplacée par la tâche plus prioritaire sur son unité d'exécution. La tâche supprimée repasse alors dans l'état *En attente* et devra de nouveau passer par le processus de reconfiguration avant de pouvoir s'exécuter.

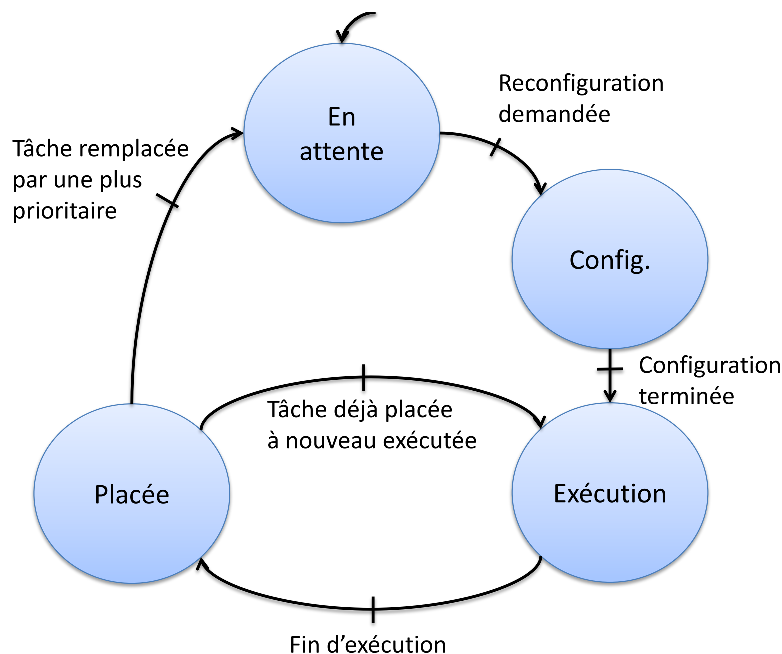


FIGURE 4.4 – *Automate fini pour les changements d'état des tâches*

Lorsqu'une tâche doit être configurée, le gestionnaire vérifie tout d'abord si une unité de calcul est disponible (c'est-à-dire non configurée, donc vierge). Si oui, la tâche va être reconfigurée sur l'unité disponible afin d'éviter d'enlever une tâche placée qui pourrait être réutilisée sans reconfiguration supplémentaire : on utilise donc le plus grand nombre d'unités possible. Si toutes les unités sont configurées, le gestionnaire recherche une tâche placée mais qui n'est pas en cours d'exécution. Pour le remplacement d'une tâche non utilisée, des

politiques classiques comme LRU (*Least Recently Used*, unité dont l'exécution remonte au plus longtemps) ou de préférence MRU(*Most Recently Used*, dernière unité exécutée) peuvent être utilisées. En effet, à l'inverse des mémoires caches où les données les plus récemment utilisées sont susceptibles d'être utilisées de nouveau très rapidement, les tâches qui viennent de s'exécuter ne seront à priori réutilisées que lors de la prochaine période et peuvent donc être sereinement remplacées par le gestionnaire. Enfin, si toutes les unités de calcul sont occupées, la configuration de la tâche est mise en attente de la libération d'une unité, c'est-à-dire en attente de la fin d'une autre tâche ou d'un point de préemption d'une tâche moins prioritaire, comme nous le verrons dans la section suivante. Notons que ce mode de fonctionnement peut être modifié par le concepteur pour privilégier par exemple l'optimisation de la consommation du système. Dans ce cas, il serait peut être plus intéressant de laisser les zones reconfigurables le plus longtemps possible dans un état vierge, sans configuration et qui donc ne consomme que très peu d'énergie.

Afin de gérer la priorité des tâches, nous avons basé notre ordonnancement sur l'algorithme très répandu EDF (*Earliest Deadline First*) qui considère la tâche dont l'échéance est la plus proche comme étant la plus prioritaire. Encore une fois, il s'agit principalement de valider la méthodologie plutôt que de développer un algorithme optimal.

Dans la première version de RecoSim, nous avons choisi d'ordonnancer les tâches indifféremment du type d'unité de calcul ciblée (zone reconfigurable ou processeur). Les mêmes politiques s'appliquent donc dans le cadre de la gestion des unités logicielles. Nous avons également pris en compte le temps de configuration de la tâche sur l'unité, représentant le temps de chargement de l'exécutable.

4.1.3.c Préemption des tâches

RecoSim gère également la préemption des tâches. La question principale à se poser est alors : quand faut-il préempter une tâche ? Nous avions donné un début de réponse à cette question dans l'état de l'art en disant que la préemption devait absolument être effectuée à des instants bien définis au cours de l'exécution de la tâche afin que le contexte soit facilement sauvegardé et restauré. Ainsi, nous avons choisi de définir explicitement les points de préemption possible de l'algorithme utilisateur de la tâche. Ce n'est donc pas le gestionnaire qui décide d'interrompre l'exécution d'une tâche mais au contraire la tâche qui, à chaque fois qu'elle atteint un point de préemption, demande au gestionnaire si elle peut continuer son exécution ou si elle doit laisser sa place à une tâche plus prioritaire. L'exécution d'une tâche est donc séparée en plusieurs parties qui sont vues comme des tâches à part entière par le gestionnaire de reconfiguration. A ce titre-là, les algorithmes d'ordonnancement utilisés ne sont pas préemptifs au sens des systèmes d'exploitation logiciels puisqu'ils ne peuvent pas directement interrompre l'exécution d'une tâche. Cette approche permet notamment au concepteur d'explorer les différentes possibilités de préemption possibles pour une tâche donnée afin de prendre une décision sur le positionnement final des points de préemption des tâches, à la fois matérielles et logicielles. Le processus est illustré par la figure 4.5. En atteignant le point de préemption P1, la tâche communique avec le gestionnaire en utilisant les sockets de communication dédiés. Ici, le gestionnaire demande l'interruption de la tâche en envoyant l'ordre Stop. Dès que possible, la gestionnaire notifie à la tâche qu'elle peut

reprendre son exécution depuis le dernier point de préemption. Entre temps, la tâche a pu migrer vers une autre entité du même type (matériel ou logiciel) ou d'un type différent, même s'il semble compliqué de migrer d'un paradigme vers un autre sans une étude au cas par cas. Il faut alors prendre en compte le coût engendré par le possible déplacement de données à effectuer.

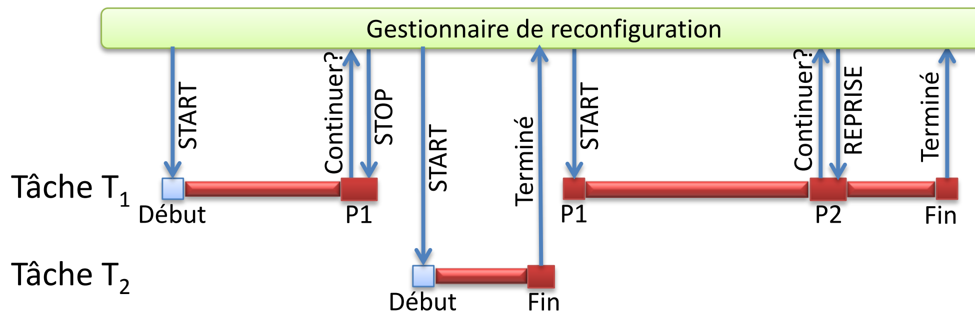


FIGURE 4.5 – *Gestion de la préemption au sein de RecoSim*

4.1.4 Communications et TLM

Les communications entre les différents composants du système comme les modules reconfigurables ou le gestionnaire de reconfiguration ont été décrites en utilisant le standard TLM-2.0 de l'Accelera Systems Initiative (anciennement OSCI) [16]. C'est une abstraction du niveau RTL dans laquelle les événements d'une simulation sont remplacés par des appels de fonction qui omettent les détails bas-niveau liés à l'implémentation du système, réduisant par la même le temps de simulation global.

La norme TLM-2 supporte deux styles de codage différents appelés *loosely-timed* (temps souple) et *approximately-timed* (temps approximatif). Le premier style de codage utilise des interfaces bloquantes qui ne fournissent que deux marques de temps pour chaque transaction, au niveau de l'appel et du retour de la fonction de transport. À l'inverse, le second style de codage utilise des interfaces de transport non-bloquantes et fournit plusieurs marques de temps. Dans le but d'affiner au maximum notre modèle et de gérer le parallélisme inhérent aux FPGA, nous avons choisi d'utiliser les interfaces non-bloquantes fournies par le style de codage *approximately timed* afin de modéliser les communications entre modules. Pour les communications avec le gestionnaire de reconfiguration, les communications n'ont pas besoin d'être gérées de manière si complexe et nous utiliserons donc des interfaces bloquantes avec le style de codage temps souple.

Plutôt que de redéfinir intégralement un protocole pour les interfaces des modules, nous avons choisi d'utiliser le protocole fourni par TLM-2, le *Base protocol*.

4.1.4.a Etude du *Base protocol*

Ce protocole sépare la transaction en quatre phases représentant autant de marques de temps possibles, comme l'illustre la figure 4.6. Durant la première phase, appelée *BEGIN_REQ*, un socket initiateur envoie une requête vers un socket cible qui peut alors ré-

pondre avec la phase *END_REQ*, acceptant ou rejetant la requête. Si la cible n'est pas en mesure de prendre une décision concernant la transaction lors de la phase *BEGIN_REQ*, elle peut rendre le contrôle au noyau de simulation SystemC. Dans l'exemple de la figure suivante, la simulation a avancé de 10 nanosecondes avant que la cible ne puisse prendre une décision. Ce comportement est possible pour chaque phase du protocole, comme c'est le cas dans notre exemple. Une fois la requête acceptée, la cible peut commencer le traitement des données. Une fois celui-ci terminé, la cible débute la séquence de réponse en envoyant la phase *BEGIN_RESP*. Enfin, le socket initiateur peut conclure la transaction par l'envoi de la phase *END_RESP*. Il existe plusieurs façons d'utiliser le protocole pour aboutir à des résultats similaires. Par exemple, il est possible d'utiliser le chemin inverse (*backward path*, en opposition au chemin direct, *forward path* qui est le chemin de données de l'initiateur vers la cible) ou le chemin de retour de la fonction (*return path*) pour les communications de la cible vers l'initiateur. Afin de conserver au maximum les différentes marques de temps du protocole, nous avons choisi d'utiliser uniquement les chemins directs et inverses pour les communications. Dans ce cas, le chemin de retour envoie toujours le message *TLM_ACCEPTED*, signifiant que la cible a bien reçu le message de l'initiateur mais qu'elle n'est pas encore en mesure de continuer la transaction.

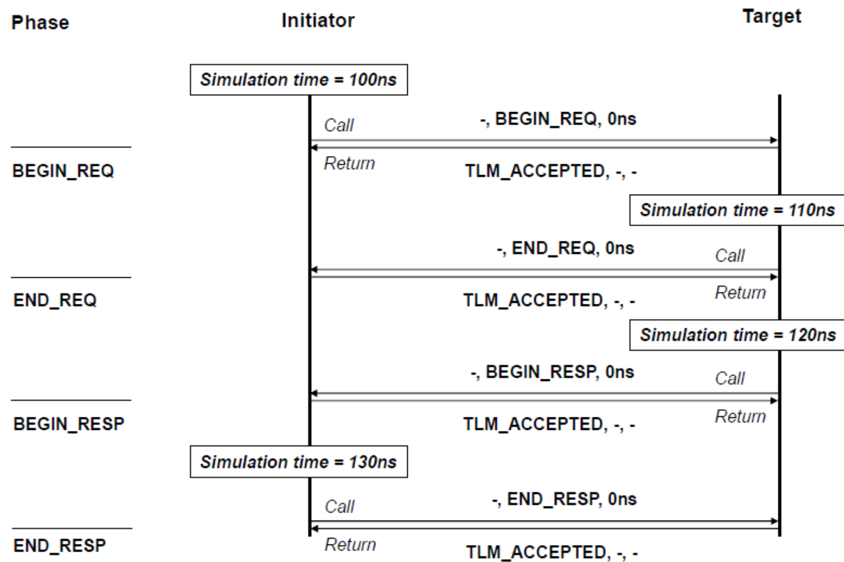


FIGURE 4.6 – *Diagramme de séquence d'une transaction suivant le Base protocol [16]*

Comme nous pouvons l'observer sur la figure, chaque phase du protocole est associée à un délai (ici 0 nanoseconde) qui représente la marque de temps de la phase. Pour la phase *BEGIN_REQ*, cette annotation de temps peut correspondre au temps nécessaire à l'écriture des données sur l'interface de sortie. Celle-ci peut aussi bien être une mémoire partagée sur un bus qu'une FIFO. Pour la phase *END_REQ*, le délai peut représenter le temps nécessaire à la récupération des données écrites par le prédécesseur. Pour le moment, ces informations sont laissées à la charge de l'utilisateur. Afin de le guider un peu plus dans le processus de définition de ces temps, il serait intéressant de proposer des profils prédéfinis pour certaines architectures typiques comme celles énoncées plus tôt.

4.1.4.b Transactions

Comme nous l'avons déjà mentionné, les sockets initiateurs et cibles communiquent par le biais d'une transaction. Celle-ci est censée représenter au mieux l'échange de données entre modules. Le standard TLM-2 définit ainsi le type *generic payload*, une transaction générique très liée aux protocoles basiques de TLM-2 et qui est initialement prévue pour représenter l'échange de données par un bus partagé. Ainsi, une transaction contient des informations concernant le type d'accès (lecture ou écriture), l'adresse à laquelle lire ou écrire les données, un pointeur vers la structure de données contenant les données à écrire ou qui contiendra les données lues depuis la cible ou encore la taille de l'accès.

Cette transaction générique n'est donc pas forcément adaptée à nos besoins. En fait, seuls les champs concernant les données nous sont nécessaires puisque nous travaillons à un haut niveau d'abstraction. Plutôt que de définir un nouveau type de transaction, nous avons choisi de conserver une transaction générique dont nous n'utiliserons que certains champs. L'avantage de cette solution est que notre module reconfigurable peut être connecté directement à d'autres modules gérant les transactions génériques sans avoir besoin de modifier l'une ou l'autre des classes.

Dans cette même volonté de standardisation de notre modèle, il est possible d'associer une extension à une transaction. Cette extension, qui peut contenir n'importe quel type de structure, permet d'enrichir les transactions avec des informations spécifiques à une application. RecoSim laisse la possibilité au concepteur d'ajouter une extension à la transaction, sans pour autant corrompre la généricité de la transaction. En effet, il est toujours possible de ne pas considérer l'extension si un module n'en a pas besoin. Par exemple, dans le cas d'une application de traitement vidéo, ce mécanisme d'extension peut être utilisé pour transporter les métadonnées de la vidéo à travers les différents éléments de la chaîne.

4.1.4.c Canaux prioritaires

Par défaut dans les modèles TLM-2, il est possible d'envoyer des données de manière indépendantes sur plusieurs sockets en même temps. Toutefois, il arrive très souvent que cela soit rendu impossible par l'implémentation de la tâche, qui devra peut-être partager la même interface d'entrée ou de sortie avec plusieurs sockets selon le type de zone reconfigurable sur lequel elle a été placée. À l'inverse, le concepteur peut savoir d'expérience qu'un socket particulier nécessitera un débit important vers le module suivant alors qu'un autre socket n'aura pas besoin de telles performances. C'est typiquement le cas d'un bus de données par rapport à un bus de métadonnées ou de contrôle (par exemple les bus AXI pour les données et AXI-Lite pour le contrôle et les accès registres).

Afin de prendre ces considérations en compte, nous avons introduit la notion de canal prioritaire, représentant un socket ayant toujours accès à un canal vers le module suivant (si c'est un socket initiateur) ou à un canal depuis le module précédent (si c'est un socket cible). En d'autres mots, le socket considéré a une interface qui lui est dédiée. Cela signifie que dès qu'un accès sera requis, il pourra être effectué dès lors que le module suivant est instancié sur le FPGA, indépendamment des communications des autres sockets. À l'inverse, les canaux

non prioritaires devront partager un seul canal physique. Un exemple de fonctionnement de ces canaux prioritaires est fourni en figure 4.7.

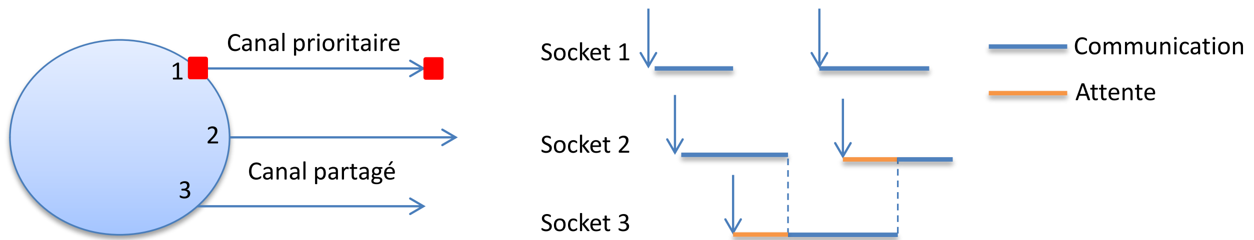


FIGURE 4.7 – *Exemple de fonctionnement des canaux prioritaires*

La définition de canaux prioritaires lève alors la question des interfaces. Même si elles ne sont pas gérées par le simulateur à proprement parler (dû à l'utilisation de la modélisation transactionnelle) les canaux prioritaires permettent de gérer partiellement cette problématique en permettant de définir des sockets bénéficiant d'une interface dédiée. Néanmoins, il est pour l'instant impossible de distinguer des groupes d'interfaces ou de regrouper les sockets par type.

4.2 Applications

Dans cette section, nous présenterons les deux applications utilisées pour valider RecoSim. La première application servira de point de repère pour la comparaison avec les solutions qui seront issues du flot FoRTReSS. La seconde application est celle du projet ARDMAHN, dont nous présenterons les cas d'utilisation qui doivent être pris en compte au sein du démonstrateur.

4.2.1 Application Secure Box

4.2.1.a Description

La première application utilisée afin de valider RecoSim est responsable de transmissions de flux vidéo sécurisés, comme le montre la figure 4.8. Le but de cette application est de sécuriser le flux vidéo issu d'une caméra pour pouvoir le transmettre dans le domaine public (qui peut être un réseau ou simplement un disque dur). La vidéo est donc encodée dans le standard MPEG-2, cryptée avec un algorithme de chiffrage AES 128 bits et enfin codé par le code correcteur Reed-Solomon avec un taux de correction 3/5, permettant de se prévaloir d'éventuelles erreurs lors de la transmission ou du stockage. Ici, le flux vidéo peut être envoyé dans le domaine public. La partie décodeur du système est l'exact opposé de la partie encodeur. Les algorithmes MPEG-2 ont été fournis par Viotech Communications [79], partenaire du projet ARDMAHN. Les IP AES et Reed-Solomon sont issues d'OpenCores [80].

Comme nous l'avons déjà mentionné, RecoSim est une aide au développeur afin de déterminer le nombre de zones reconfigurables nécessaires pour que l'application respecte ses contraintes de temps en suivant un algorithme d'ordonnancement donné. Nous allons donc essayer de déterminer l'architecture nécessaire à notre application de flux vidéo sécurisé. Pour

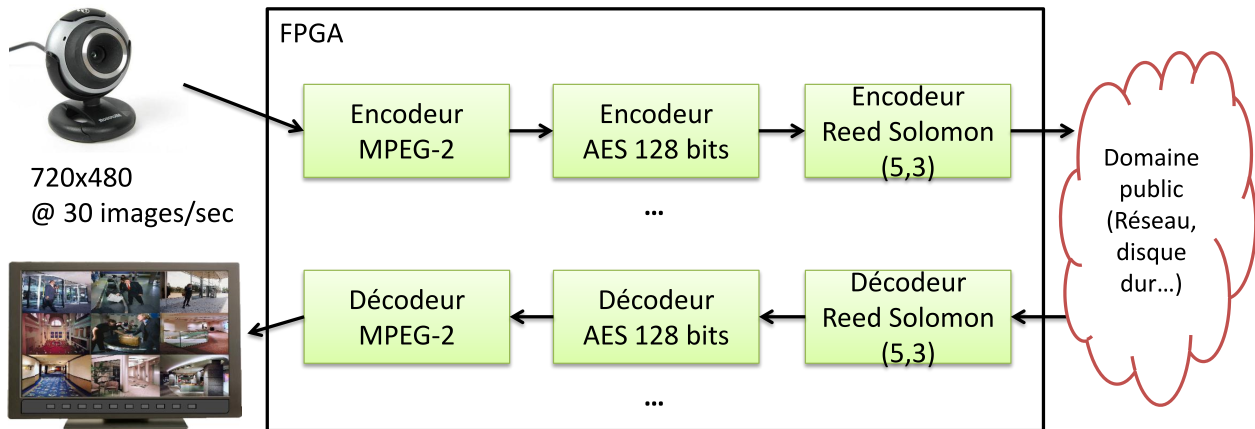


Figure 4.8 – *Application de transmissions vidéo sécurisées*

cela, nous allons commencer par fournir au simulateur une architecture avec une seule zone reconfigurable puis incrémenter ce nombre de zones jusqu'à obtention d'une solution viable.

Même si les zones reconfigurables ne sont pas placées sur le FPGA, il est important de définir clairement quelles sont les ressources qui seront nécessaires à chaque zone reconfigurable. En effet, le simulateur a besoin des temps de reconfiguration pour chaque paire tâche/zone reconfigurable afin d'avoir une solution la plus proche de la réalité possible. Plaçons nous dans le cas ou le concepteur n'est pas en possession des netlists, mais a pour seule information une estimation des ressources de l'IP. Il doit alors être capable à partir de ces informations de définir des zones reconfigurables qui pourront accueillir la tâche, pour dans un second temps déduire de la zone la taille des bitstreams générés afin de pouvoir utiliser notre modèle de coût défini dans le chapitre précédent, section 3.2 page 42. Pour cela, le concepteur a besoin des informations issues des rapports de synthèse de chaque tâche. Pour l'application considérée, les informations nécessaires sont compilées dans le tableau 4.1.

Tableau 4.1 – *Rapports de synthèse des tâches de l'application Secure Box pour un FPGA Virtex-6*

Tâche	Fréquence (MHz)	Temps d'exécution (ms)	Slice registres	Slice LUTs	Slice LUTs comme mémoire	RAMB36	DSP48
MPEG-2							
Encodeur	100	5.3	4684	5375	17	10	16
Décodeur	100	5.3	3190	3710	-	8	10
AES							
Encodeur	100	8.1	1325	2563	-	-	-
Décodeur	100	8.1	1324	3171	-	-	-
Reed-Solomon							
Encodeur	125	10	30	43	-	-	-
Décodeur	250	10	63	134	8	-	-

Les temps d'exécution fournis dans le tableau 4.1 correspondent au temps de traitement pour une image. Nous avons choisi cette granularité de traitement (et donc potentiellement,

une granularité de reconfiguration) à cause des temps de reconfiguration des tâches de l'application. En effet, les tâches MPEG-2 peuvent fonctionner à une granularité macrobloc, ce qui représente une petite partie d'image compressée. En considérant des zones reconfigurables optimales et en utilisant la compression de bitstreams au sein de FaRM, les temps de reconfiguration de ces tâches peuvent facilement atteindre 1.5 ms, ce qui est bien trop important en comparaison du temps de traitement d'un macrobloc qui est d'environ 400 cycles, c'est-à-dire $4 \mu s$ à 100 MHz. Pour atteindre un rapport entre temps d'exécution et temps de reconfiguration, il est nécessaire d'augmenter la granularité de traitement. À partir d'une granularité d'une image, composée de 1350 macroblocs pour une vidéo de résolution 720x480, nous obtenons des temps d'exécution supérieurs aux temps de reconfiguration comme le montre le tableau précédent. Les temps d'exécution pour les tâches AES et Reed-Solomon ont été déduits de leurs spécifications en considérant des flux de données d'environ 760 ko pour une image (un macrobloc étant lui-même composé de 6 blocs, chacun composé de 64 mots de 12 bits). Étant donné le fonctionnement de ces tâches, il est possible de récupérer les données au fur et à mesure plutôt qu'en une seule fois avant l'exécution de l'IP. Cela permet également de cacher les temps de récupération du paquet de données $n + 1$ par l'exécution de l'algorithme sur le paquet n. Ainsi, les délais de communication pour chaque module du système sont considérés nuls.

Le tableau 4.1 met également en évidence les fréquences de chaque tâche. Nous pouvons voir que la plupart fonctionnent à une fréquence de 100 MHz, à l'exception des encodeurs et décodeurs Reed-Solomon. Nous avons choisi d'augmenter la fréquence de fonctionnement de ces deux tâches afin d'homogénéiser les temps d'exécution et de laisser plus de flexibilité au simulateur pour les décisions de placement de tâches. En effet, l'échéance de chaque tâche est de 33.3 ms, le système devant traiter en pipeline 30 images par secondes alors que les temps d'exécution pour les tâches Reed-Solomon sont de 13 et 25 ms à 100 MHz. Il n'est donc pas toujours évident de partager une zone reconfigurable avec ces tâches tout en respectant les contraintes de temps. Toutefois, il est toujours possible de laisser ces tâches statiques plutôt que reconfigurables si jamais il n'apparait aucun gain en temps ou en surface, ou bien si des considérations énergétiques empêchent l'augmentation des fréquences de fonctionnement. Nous avons choisi de laisser toutes les tâches de l'application reconfigurables dynamiquement afin de mettre en avant les possibilités de RecoSim.

Le simulateur a besoin de connaître le modèle alloué du système. De notre point de vue, cela correspond à savoir quelle zone reconfigurable pourra accueillir quelle tâche. Cette étape est la plus compliquée car le degré de liberté du concepteur est bien plus important que pour les autres étapes. En effet, si l'on considère que l'on a n tâches à placer sur m zones reconfigurables, il y a au total $n*m$ paires tâche/zone reconfigurable possibles. Sachant qu'une solution peut être composée d'un nombre quelconque de paires tâche/zone reconfigurable, notre espace de solution a ainsi une taille de $\frac{n*m(n*m+1)}{2}$ éléments, ce qui peut être considérable pour un nombre de tâches ou de zones (souvent lié au nombre de tâches) élevé. Afin de limiter le nombre de solutions, nous partirons toujours d'une allocation qui est la moins optimale en terme de coût mais celle qui aura le plus de chances de réussir, à savoir placer les tâches sur toutes les zones qui peuvent les accueillir (en termes de ressources). Ainsi, l'ordonnanceur a

la plus grande liberté possible pour effectuer le placement des tâches : le cas où une tâche ne peut être instanciée sur le FPGA par manque de ressources arrivera ainsi bien moins souvent.

Dans le cadre de notre application, nous avons déterminé la taille optimale des zones reconfigurables pour chaque tâche. Pour cela, nous nous sommes basés sur le guide de configuration des FPGA Xilinx [15, chap. 6], qui fournit les informations sur la quantité de ressources présentes dans chaque colonne. Ainsi, en technologie Virtex-6, on sait qu'une colonne contiendra, selon son type, 40 CLB, 8 blocs mémoires RAMB36 ou 16 DSP48. Les CLB sont quant à eux constitués de deux slices qui peuvent être de type L ou M, la différence étant la présence dans les slices M de RAM distribuée et de registres à décalage. Dans la technologie Virtex-6, chaque slice est composé de 4 LUT et 8 flip-flops.

Pour déterminer le nombre de colonnes de DSP ou de BRAM nécessaires, il suffit juste de contraindre un nombre suffisant de RAMB36 ou DSP48. Pour les colonnes de slices, il faut tout d'abord prendre en compte les LUT utilisées comme mémoire (LUTRAM) qui donnera le nombre de SliceM. Ensuite, le nombre de registres et de LUT nécessaire nous donne un nombre de Slice (indifféremment L ou M). Pour cela, il faut regarder pour ces deux types de ressources le nombre de colonnes nécessaires et prendre le plus grand nombre des deux. Prenons l'exemple de l'encodeur MPEG-2 dont les besoins en ressources sont fournis par le tableau 4.1. Les 5375 LUT requises représentent $5375/4 = 1344$ slices, soit $1344/40 = 34$ colonnes de slices, alors que les 4684 registres représentent $4684/8 = 586$ slices, soit 15 colonnes. La zone reconfigurable doit donc contenir 34 colonnes de slices, c'est-à-dire 17 colonnes de CLB. L'ensemble des résultats obtenus sont consignés dans le tableau 4.2.

Tableau 4.2 – *Taille des zones reconfigurables optimales pour un FPGA Virtex-6*

Tâche	Colonne(s) de Slice	Colonne(s) de SliceM	Colonne(s) de BRAM	Colonne(s) de DSP
Encodeur MPEG-2	34	1	2	1
Décodeur MPEG-2	24	-	1	1
Encodeur AES	17	-	-	-
Décodeur AES	20	-	-	-
Encodeur Reed-Solomon	1	-	-	-
Décodeur Reed-Solomon	1	1	-	-

Concernant FaRM, nous n'utiliserons ici que le mode d'écriture simple (sans préchargement) couplé à la compression des bitstreams. En effet, l'algorithme d'ordonnancement utilisé (pour rappel, l'EDF), n'est pas capable de prévoir à l'avance les prochaines reconfigurations qui seront nécessaires, rendant inutile l'utilisation du préchargement. Pour ce qui est de la compression des bitstreams, nous l'utiliserons afin de réduire au maximum les temps de reconfiguration. Afin d'effectuer une première estimation rapide des solutions, nous considèrerons un taux de compression égal au taux de compression moyen obtenu lors des tests effectués et rapportés dans le tableau 3.1 en page 38, à savoir 26.7%.

4.2.1.b Résultats

Nous allons donc commencer par une simulation avec une seule zone reconfigurable. Cette zone devra être capable d'accueillir chacune des tâches de l'application (sinon l'application ne pourrait pas s'exécuter entièrement), ce qui signifie que les ressources contraintes par la zone correspondent au maximum des ressources pour chaque tâche et par type de ressources. Ainsi, en utilisant le tableau 4.2, nous estimons que cette zone reconfigurable sera composée de 34 colonnes de SliceL, 1 de SliceM, 2 de BRAM ainsi qu'une de DSP. À partir de cette information, il nous est possible d'estimer la taille du bitstream non compressé, qui sera nécessaire pour utiliser notre modèle de coût. Comme nous l'avons vu dans le chapitre précédent sur FaRM, la taille du bitstream est lié au nombre de frames de configuration nécessaires pour chaque type de ressources, chaque frame d'un FPGA Virtex-6 contenant 82 mots, soit le double que pour les FPGA Virtex-5 (les ressources par colonnes étant elles-aussi doublées). Nous obtenons ainsi une taille de bitstream de 318 ko pour un temps de reconfiguration de 1.8 ms en considérant une compression moyenne de 26.7%.

De manière assez prévisible, cette solution se traduit par un échec de la simulation après seulement quelques paquets envoyés. Il semblait un peu optimiste de faire partager une même ressource à des applications dont la somme des temps d'exécution sur une période est supérieure à la période elle-même. En effet, une fois le régime permanent atteint, le pipeline est rempli et toutes les tâches de l'application doivent s'exécuter en même temps (comprendre au sein de la même période) et donc partager les mêmes ressources.

Nous passons donc à une solution à deux zones reconfigurables, toutes deux capables d'accueillir l'intégralité des tâches de l'application. On double ainsi la solution prévue pour une zone reconfigurable, et les temps de reconfiguration pour les deux zones sont identiques à ceux de la première solution. Dans ce cas, la simulation est un succès : les tâches ont toujours réussi à satisfaire leurs contraintes de temps (mode de simulation temps-réel dur, HRT). Les résultats de la simulation apparaissent dans la tableau 4.3. Concernant le gain en ressources, nous avons séparé le gain en surface brut, considérant uniquement la surface des zones reconfigurables, du gain total, prenant également en compte les ressources nécessaires à FaRM pour la gestion de l'architecture dynamique.

Tableau 4.3 – *Résultats en surface de l'application Secure Box pour un FPGA Virtex-6*

Ressource	Surface statique	Surface reconfigurable (colonnes)	Surface reconfigurable	Gain (%)		Taux d'occupation ICAP (%)
				Brut	Total	
Slice	3750	70	2800	25.3	16.9	
BRAM	18	4	32	-77.7	-77.7	18.2
DSP	26	2	32	-23.1	-23.1	

Ces résultats nous montrent que même si le gain total en ressources de type CLB est déjà intéressant, cette solution gâche un grand nombre de ressources plus rares sur le FPGA et plus coûteuses en terme de temps de reconfiguration comme les BRAM ou les DSP. Ceci est dû au

fait que ces ressources sont requises uniquement par les tâches encodeur et décodeur MPEG-2 alors que les autres tâches n'en ont pas besoin. Il n'est donc peut-être pas nécessaire de contraindre ces types de ressources pour les deux zones reconfigurables. En d'autres termes, il faut maintenant partitionner le système pour déterminer quelles tâches pourront être placées sur quelles zones reconfigurables pour optimiser la fragmentation interne à tout instant.

Pour effectuer le partitionnement, intéressons-nous aux temps d'exécution des différentes tâches sur les zones reconfigurables à l'issue de la simulation. Ces résultats, consignés dans le tableau 4.4, montrent que le temps d'exécution cumulé des tâches MPEG-2 est très inférieur à 100%, ce qui signifie qu'il doit être possible de les faire partager une seule zone reconfigurable. Ainsi, il ne serait plus nécessaire de dimensionner la seconde zone reconfigurable pour qu'elle puisse accueillir les tâches MPEG-2, résultant en un gain en ressources BRAM et DSP. Pour pouvoir accueillir les tâches AES et Reed-Solomon, la seconde zone n'a en effet pas besoin de contraindre de mémoires BRAM ou de DSP. La seconde zone reconfigurable devrait ainsi générer un bitstream de taille 127 ko pour un temps de reconfiguration en mode compression de 979 ns.

Tableau 4.4 – *Temps d'exécution des tâches sur les zones reconfigurables (exprimés en pourcentage du temps d'exécution des zones reconfigurables). La somme des taux d'occupation sur une même zone sont donc de 100%.*

Tâche	Zone 1	Zone 2	Total
Encodeur MPEG-2	0.2	23	23.2
Décodeur MPEG-2	22.9	-	22.9
Encodeur AES	-	34.8	34.8
Décodeur AES	34.3	-	34.3
Encodeur Reed-Solomon	0.4	42.2	42.6
Décodeur Reed-Solomon	42.2	-	42.2

Cette solution nous donne les résultats du tableau 4.5. Cette fois-ci, une solution reconfigurable dynamiquement permettra un gain de ressources pour chaque type, allant jusqu'à 31.8% pour les CLB. Les résultats de la simulation pour les temps d'exécution sont donnés par le tableau 4.6.

Tableau 4.5 – *Résultats en surface de l'application Secure Box après partitionnement*

Ressource	Surface statique	Surface reconfigurable (colonnes)	Surface reconfigurable	Gain (%)		Taux d'occupation ICAP (%)
				Brut	Total	
Slice	3750	56	2240	40.3	31.8	
BRAM	18	2	16	11.1	11.1	32.1
DSP	26	1	16	38.5	38.5	

À la fin de chaque simulation, RecoSim génère une trace au format VCD (*Value Change Dump*) dont un exemple est donné par la figure 4.9. Elle intègre les signaux représentatifs

Tableau 4.6 – *Temps d'exécution des tâches sur les zones reconfigurables après partitionnement (exprimés en pourcentage du temps d'exécution des zones reconfigurables)*

Tâche	Zone 1	Zone 2	Total
Encodeur MPEG-2	22.4	x	22.4
Décodeur MPEG-2	22.4	x	22.4
Encodeur AES	33.3	0.4	33.7
Décodeur AES	0.3	35.2	35.5
Encodeur Reed-Solomon	20.8	21.8	42.6
Décodeur Reed-Solomon	0.8	42.6	42.6

des communications entre modules ainsi que l'exécution des différents algorithmes utilisateurs liés aux modules reconfigurables. Les informations fournies par la trace sont très utiles à la mise au point du système. Par exemple, il est possible d'observer l'évolution de l'utilisation de l'ICAP ou de son équivalent logiciel et donc les éventuels pics, représentant des périodes de temps durant lesquelles plusieurs reconfigurations se succèdent rapidement. Dans une telle situation, il est possible que les contraintes de temps du dernier module ne puissent être satisfaites uniquement à cause d'une utilisation de l'ICAP trop intensive. La trace met également en évidence l'utilisation des ressources du FPGA à un instant donné. On peut voir sur la trace que l'utilisation des ressources CLB évolue au fil du temps, dépendant des tâches implémentées sur les zones reconfigurables à l'instant étudié. Le concepteur sait alors précisément quelles ressources sont "gâchées" pour l'intégralité de la partie reconfigurable du design et quels sont les points à améliorer lors de la définition du modèle alloué de l'application. Notons que les échelles de temps sont différentes pour les deux traces présentées : la première trace met en évidences les changement d'état des tâches liés à l'utilisation de la reconfiguration dynamique et est affichée sur une période de temps beaucoup plus courte que la seconde trace qui montre l'évolution globale des taux d'occupation du système.

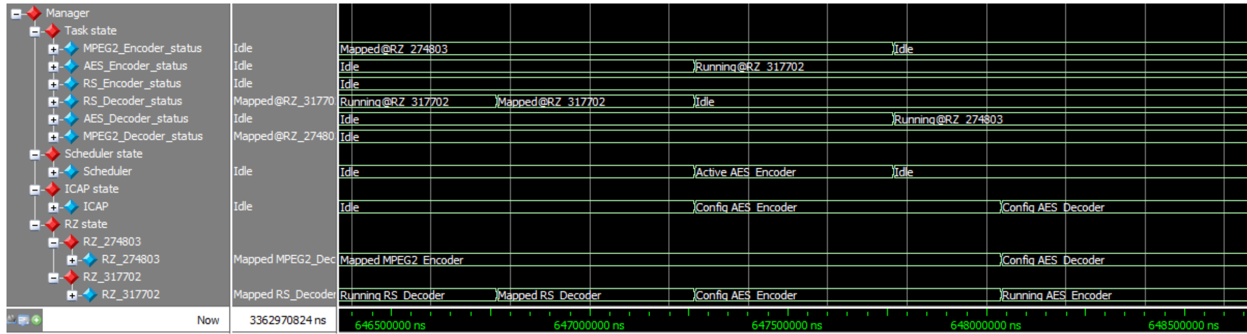
Sur la trace de la figure 4.9(b), nous voyons également que le taux d'occupation de l'ICAP tend à se stabiliser après un régime transitoire. Lorsque le régime permanent est atteint, le fonctionnement du système peut être prédit et il est alors inutile de continuer la simulation.

4.2.2 Application ARDMAHN

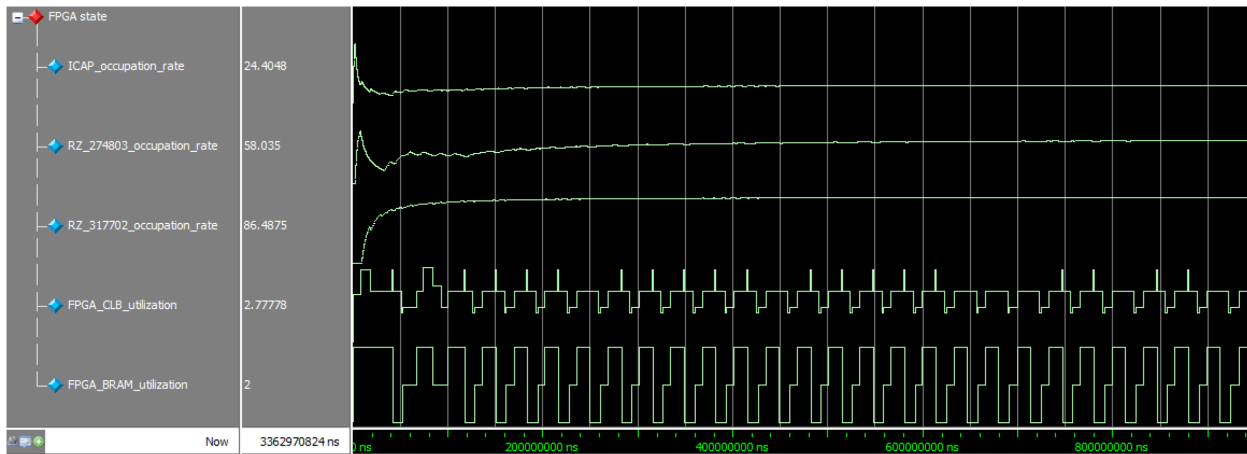
L'application étudiée au sein du projet ARDMAHN consiste en un transcodeur vidéo prenant en compte les standards H.264 et MPEG-2. Dans un premier temps, nous décrirons l'application ainsi que les différents cas d'utilisation définis par le consortium puis nous analyserons les résultats obtenus par RecoSim.

4.2.2.a Description

Le but du projet est que les utilisateurs de la set-top box puissent regarder n'importe quel flux vidéo, quel que soit son encodage, sur n'importe quel terminal, quels que soient les résolutions et encodages gérés. Ainsi, une vidéo en H.264 haute définition de résolution



(a) Trace pour le gestionnaire de reconfiguration



(b) Taux d'utilisation des zones et des ressources du FPGA

Figure 4.9 – *Exemples de traces VCD*

1920x1080 doit être transcodée pour pouvoir être lue dans une résolution de 720x480 plus adaptée aux terminaux mobiles comme certains smartphones.

Pour cela, la chaîne de transcodage a été séparée en trois étapes :

- Décodage : le flux vidéo est d'abord décodé vers un format d'images brut.
- Adaptation : le flux vidéo est adapté au terminal cible (résolution, bitrate...).
- Encodage : le flux vidéo adapté est alors encodé vers un format lisible par le terminal.

Pour des raisons pratiques, il a été décidé de ne pas reconfigurer dynamiquement le bloc d'adaptation. Celui-ci est donc statique mais paramétrable. Les cas d'utilisation du démonstrateur ARDMAHN sont listés dans le tableau 4.7.

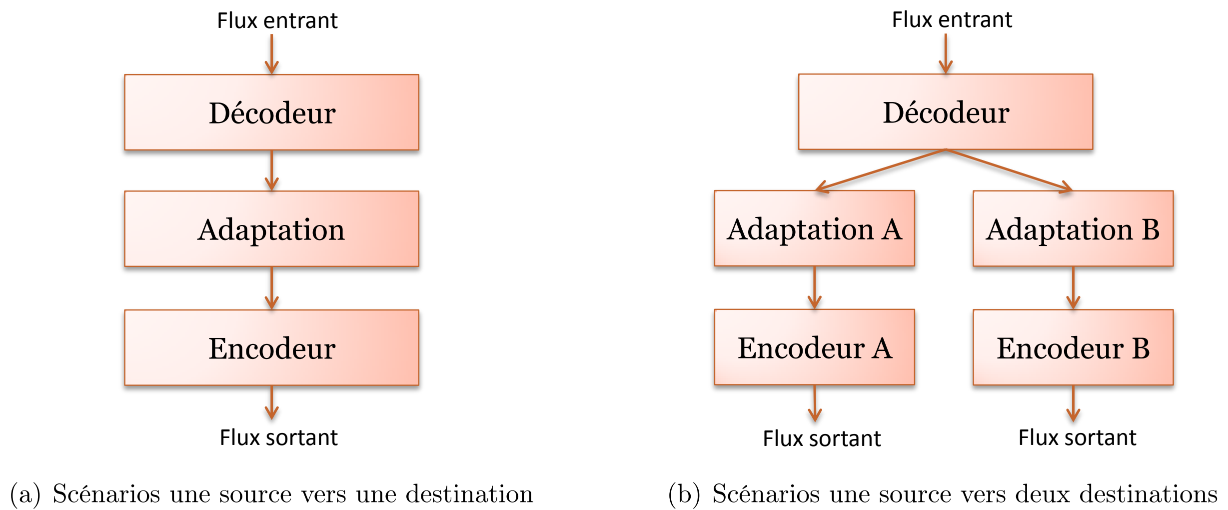
Les scénarios 1 et 2 ne possèdent pas d'adaptation mais permettent de valider le démonstrateur dans son fonctionnement minimal. Le scénario 3 permet de valider un redimensionnement homogène. Le scénario 4 permet de valider un changement de standard (transcodage) sans autre adaptation tandis que les scénarios 5 et 6 opèrent un transcodage couplé à une adaptation. Les scénarios 7 et 8 sont des scénarios multi-flux, ce qui signifie que le système doit gérer l'adaptation de plusieurs flux simultanément. Les cas définis pour les scénarios 7 et 8 sont de type "une source vers deux terminaux" : un flux arrive vers le démonstrateur et doit être adapté puis encodé pour convenir aux deux terminaux ciblés. Enfin, le démonstrateur devra être capable de gérer simultanément deux scénarios mono-flux (par exemple, les scénarios 5 et 6 doivent pouvoir s'exécuter en même temps sur le FPGA).

Tableau 4.7 – *Cas d'utilisation du démonstrateur ARDMAHN*

Scénario	Flux entrant		Terminal			Adaptation
	Résolution	Standard	Nom	Résolution	Standard	
1	HD	H.264	TV HD	HD	H.264	Rien
2	SD	MPEG-2	Tablette	SD	MPEG-2	Rien
3	HD	H.264	Smartphone	SD	H.264	Adaptation
4	SD	MPEG-2	Smartphone	SD	H.264	Transcodage
5	HD	H.264	Tablette	SD	MPEG-2	Transcodage + Adaptation
6	SD	MPEG-2	TV HD	HD	H.264	Transcodage + Adaptation
7	HD	H.264	TV HD	HD	H.264	Rien
			Tablette	SD	MPEG-2	Transcodage + Adaptation
8	SD	MPEG-2	TV HD	HD	H.264	Transcodage + Adaptation

Ainsi, les cas d'utilisation peuvent être résumés par les schémas-blocs de la figure 4.10. Le cas multi-flux de la figure 4.10(b) peut être ramené à des cas d'utilisation mono-flux s'exécutant simultanément sur le FPGA : au lieu de partager le décodeur pour les deux flux traités, il est possible de totalement séparer la gestion des deux flux en dupliquant la partie décodage. Cette solution n'est bien évidemment pas optimale d'un point de vue utilisation des ressources. Toutefois, le démonstrateur devant gérer tous les cas d'utilisation définis précédemment, il doit être capable d'exécuter sans problème le cas le plus critique, correspondant à la gestion simultanée de deux scénarios différents et donc potentiellement de deux chaînes de transcodage différentes. Ainsi, la simplification proposée du cas d'utilisation multi-flux est possible. Enfin, l'architecture finale du démonstrateur résultera du cas d'utilisation le plus contraignant : nous rechercherons ainsi l'architecture permettant de satisfaire les contraintes de temps de l'application dans le cas d'utilisation avec deux flux vidéo distincts simultanément.

Les ressources nécessaires aux tâches de l'application ARDMAHN pour les deux standards gérés (H.264 et MPEG-2) sont spécifiées dans le tableau 4.8. Nous y avons également inclus le bloc d'adaptation, même si celui-ci est toujours statique, afin de fournir les informations concernant son temps d'exécution et sa fréquence de travail. Les temps d'exécution correspondent au pires cas qui peut être obtenu (WCET, *Worst Case Execution Time*), c'est-à-dire à la plus grande résolution proposée : 720x480 pour MPEG-2, 1920x1080 pour H.264. Les temps d'exécution des décodeurs peuvent être estimés assez précisément à partir des simulations et de la taille de flux considérée puisque le nombre de données en sortie est connu et fixe (le décodeur génère un flux de données brutes). Pour l'encodeur, la taille du flux de sortie est inconnue et il faut donc utiliser une estimation du pire cas, à savoir lorsque le flux n'est pas du tout compressé et que la taille du flux de sortie est la même que celle du flux d'entrée. Dans ce cas, on atteint un temps d'exécution de plus de 30 millisecondes à 100 MHz pour



(a) Scénarios une source vers une destination

(b) Scénarios une source vers deux destinations

Figure 4.10 – *Schémas-blocs des cas d'utilisation ARDMAHN*

un flux full HD, ce qui est extrêmement proche de la limite de temps de 33.3 millisecondes si l'on considère un flux de trente images par secondes. Nous avons donc choisi d'augmenter la fréquence de ce bloc jusqu'à sa limite de 150 MHz ce qui permet de diminuer le temps d'exécution à un peu plus de 20 millisecondes, bien plus propice au partage des ressources avec l'utilisation de la reconfiguration dynamique.

Pour le moment, le décodeur H.264 est encore en phase de développement et nous n'avons pas encore d'informations pertinentes de temps d'exécution ou de besoins en ressources à utiliser dans notre modèle. Nous utiliserons alors les mêmes valeurs que pour l'encodeur H.264, ce qui semble être une estimation assez pessimiste pour les ressources si l'on compare à la différence existant entre le décodeur et l'encodeur MPEG-2.

Tableau 4.8 – *Rapports de synthèse des tâches de l'application ARDMAHN pour un FPGA Virtex-6*

Tâche	Fréquence (MHz)	Temps d'exécution (ms)	Slice registres	Slice LUTs	Slice LUTs comme mémoire	RAMB36	DSP48
MPEG-2							
Encodeur	100	5.3	4684	5375	17	10	16
Décodeur	100	5.3	3190	3710	-	8	10
H.264							
Encodeur	150	20.7	1128	5730	74	-	-
Adaptation	100	6.7	-	-	-	-	-

Concernant le bloc d'adaptation, le temps d'exécution varie en fonction des tailles de flux traités et de la différence entre les tailles de flux. En effet, le bloc d'adaptation est responsable du redimensionnement de l'image brute. Dans le cas d'une réduction de l'image, il est nécessaire de connaître plusieurs macroblocs du flux décodé afin de les fusionner en un

seul macrobloc du flux à encoder. Les trois types de réduction possible et leur influence sur le stockage des macroblocs du flux décodé sont les suivants :

- Division par 1 : conserve la taille de l'image mais modifie des paramètres comme le bitrate. Le bloc d'adaptation mémorise un macrobloc avant d'envoyer les données, ce qui correspond à une latence de 384 cycles (chaque macrobloc étant composé de 6 blocs de 64 données).
- Division par 2 : il faut mémoriser une ligne plus deux macroblocs pour générer le premier macrobloc en sortie. La latence est donc variable et dépend de la taille de l'image initiale. À titre d'exemple, pour un flux de résolution 1920x1080, il faut stocker 122 macroblocs (une ligne étant composée de 120 macroblocs) ce qui représente une latence de 6912 cycles.
- Division par 4 : il faut alors mémoriser trois lignes et 4 macroblocs du flux d'entrée. Pour une résolution de 1920x1080, cela représente 364 macroblocs pour une latence d'environ 1.4 ms.

Dans le cas du transcodage étudié (H.264 vers MPEG-2), il s'agit d'une division par quatre et donc la latence est la plus élevée possible. Il faut ajouter à cette latence le temps d'écriture des macroblocs qui dépend de la résolution du flux à encoder. Pour du MPEG-2, la résolution maximale est de 720x480. Il faut donc 518400 cycles pour écrire toutes les données si on considère un débit d'un mot par cycle d'horloge une fois les premiers macroblocs mémorisés. Le temps d'exécution total maximal du bloc d'adaptation est donc de 6.7 millisecondes.

À partir de ces données, un début d'architecture a été spécifié et est illustré par la figure 4.11. Cette architecture est articulée autour de quatre zones reconfigurables (deux spécifiques aux encodeurs, deux aux décodeurs) connectées à un bus AXI et à deux blocs d'adaptation statiques. La partie droite de ce schéma-bloc fait apparaître le sous-système de configuration. Cette architecture a l'avantage de ne pas utiliser la reconfiguration dynamique pour mutualiser les ressources, mais uniquement pour changer la fonctionnalité du système, c'est-à-dire changer de cas d'utilisation. Ainsi, on est assuré que les contraintes de temps sont toujours respectées, l'architecture étant statique au sein d'un même cas d'utilisation. En contrepartie, l'architecture n'est pas du tout optimisée en termes de surface reconfigurable. Nous proposons d'étudier ici le gain qui pourrait être obtenu en tirant parti de la mutualisation des ressources grâce à la reconfiguration dynamique.

Pour cela, il est nécessaire de s'intéresser au pire cas d'utilisation en termes de temps d'exécution et de besoins en ressources. Dans notre cas, il s'agit d'un transcodage H.264 d'une résolution de 1920x1080 vers un flux MPEG-2 en 720x480. Dans ce cas d'utilisation, les encodages et décodages se font sur les plus hautes résolutions acceptées et la différence de résolution oblige le bloc d'adaptation à mémoriser un maximum de macroblocs avant de pouvoir débuter l'écriture des données de sortie. Les ressources nécessaires à ces tâches sont également parmi les plus élevées, ce qui nous assure de la faisabilité de toutes les solutions si l'architecture est validée pour ce cas d'utilisation.

De la même manière que pour l'application Secure Box, il est maintenant possible de déterminer la taille des zones reconfigurables optimales pour l'application ARDMAHN. Le tableau 4.9 montre les résultats obtenus.

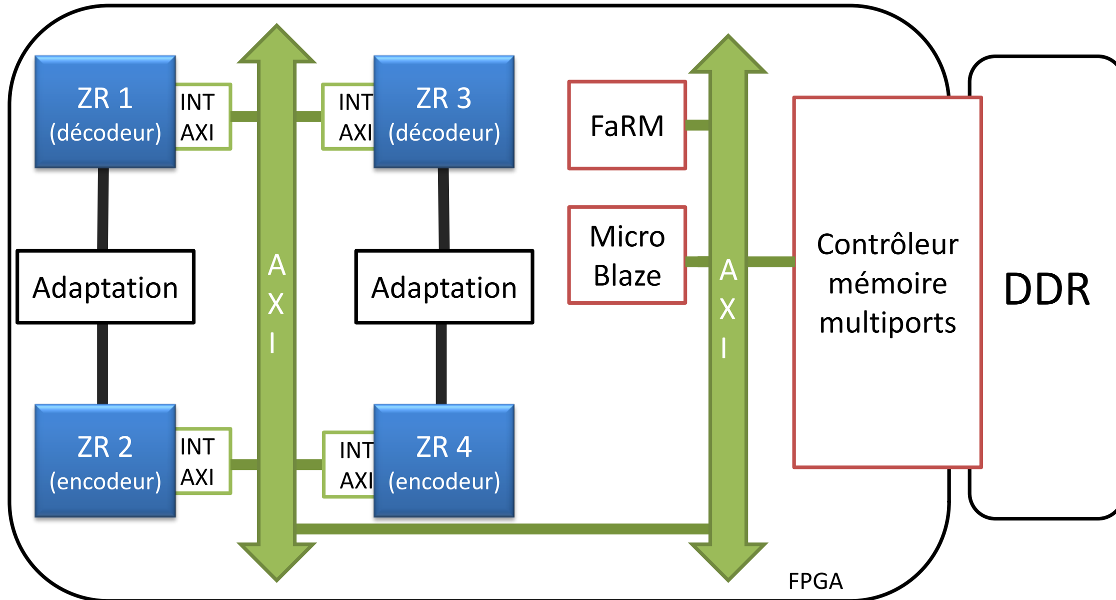


FIGURE 4.11 – *Architecture du FPGA du démonstrateur ARDMAHN*

TABLEAU 4.9 – *Taille des zones reconfigurables optimales des tâches de l'application ARDMAHN pour un FPGA Virtex-6*

Tâche	Colonne(s) de Slice	Colonne(s) de SliceM	Colonne(s) de BRAM	Colonne(s) de DSP
Encodeur MPEG-2	34	1	2	1
Décodeur H.264	36	1	-	-

4.2.2.b Résultats

Nous avons procédé de la même manière que pour la précédente application afin de déduire une architecture possible du FPGA. La première simulation réussie correspond à une architecture à deux zones reconfigurables pouvant chacune abriter toutes les tâches. Le tableau 4.10 montre les résultats en surface obtenus pour le traitement simultané de deux flux sur le FPGA.

TABLEAU 4.10 – *Résultats en surface de l'application ARDMAHN pour un FPGA Virtex-6*

Ressource	Surface statique	Surface reconfigurable (colonnes)	Surface reconfigurable	Gain (%) Brut	Gain (%) Total	Taux d'occupation ICAP (%)
Slice	2688	72	2880	-7.1	-18.9	
BRAM	10	2	16	-60	-60	24.5
DSP	16	1	16	0	0	

Les chiffres présentés ne sont clairement pas en faveur d'une implémentation dynamique-

ment reconfigurable. Toutefois, il est ici impossible de partitionner l'application. En effet, le partitionnement forcerait le placement des deux encodeurs MPEG-2 sur la même zone reconfigurable pour mutualiser les besoins en BRAM et DSP. De la même manière, les deux encodeurs H.264 seraient placés sur la même zone reconfigurable, ce qui ne permettra pas de respecter les contraintes temps-réel de l'application. En effet, il est ici impossible d'exécuter les deux encodeurs, aux temps d'exécution de plus de 20 millisecondes, dans une période de temps de 33.3 millisecondes (pour respecter un flux à 30 images par secondes). Ainsi, cette solution dynamique est la meilleure possible sous ces conditions.

Toutefois, il est nécessaire de tempérer ces propos. Le but du démonstrateur est de fournir un environnement flexible pour permettre de basculer entre les différents cas d'utilisation. Ainsi, pour une version entièrement statique du démonstrateur, il faut implémenter les encodeurs et décodeurs pour chacun des standards utilisés (ici H.264 et MPEG-2). Dans ce cas, une implémentation reconfigurable dynamiquement est plus intéressante et surtout plus flexible et évolutif : il est tout à fait possible d'ajouter la gestion de nouveaux standards très facilement sans être limité par les ressources disponibles sur le FPGA utilisé.

4.3 Perspectives

RecoSim utilise une représentation au niveau transactionnel des communications entre modules afin de diminuer le temps de simulation global et de s'affranchir de détails trop bas niveau et trop proches de l'implémentation qui ne sont pas toujours pertinent lors des premières étapes de la conception. Nos modèles utilisent donc les phases définies par la norme TLM pour représenter notamment les temps de récupération des données, par exemple dans une mémoire partagée. Toutefois, ces temps sont définis comme étant un temps moyen d'accès et peuvent être effectués en parallèle, ce qui ne correspond pas à la réalité où il ne peut y avoir qu'un seul accès à la mémoire à la fois. Ces problèmes de congestion lors des accès à la mémoire, ou plus généralement sur un bus de données, ne sont pas pris en compte par RecoSim et pourraient mener à des cas où la simulation valide l'architecture alors que l'implémentation sur carte ne donne pas de bons résultats. Ainsi, la séquentialisation des accès bus, avec éventuellement la gestion des priorités, devra être pris en compte pour que les paramètres temporels de RecoSim représentent au mieux la réalité.

Nous avons mentionné à plusieurs reprises qu'il était possible de faire de la co-simulation VHDL/SystemC qui serait intéressante au niveau de l'algorithme utilisateur qui représente le comportement de la tâche. Il est alors nécessaire de développer une interface entre la simulation SystemC et le code VHDL de la tâche, appelée adaptateur. Pour le moment, la conception de ces adaptateurs, qui sont donc différents en fonction de l'interface de la tâche, est laissée à la charge du concepteur. Il est envisageable de développer un patron d'adaptateur correspondant à notre approche TLM pour certains types d'interfaces communément utilisées. Par exemple, les outils de la suite Xilinx permettent de générer des patrons d'IP dont le cœur ont une interface de type IPIF (*Intellectual Property Interface*). Cela offrirait une approche plus directe au concepteur souhaitant tester le code d'implémentation VHDL d'une tâche.

4.4 Conclusion

Dans ce chapitre, nous avons étudié la manière dont RecoSim fonctionne pour permettre la simulation d'architectures reconfigurables dynamiquement. Basé sur la librairie open-source SystemC, RecoSim utilise notamment les threads dynamiques ainsi qu'un modèle transactionnel des communications au sein du modèle dans le but de réduire au maximum le temps de simulation. Le modèle est basé sur une approche en Y où la description de l'application et de l'architecture sont séparées puis rassemblées en un modèle alloué. Le simulateur vérifie que les contraintes de temps de l'application sont respectées de manière stricte (temps-réel dur) ou de manière plus tolérante (temps-réel souple).

Notre approche a été testée avec une application de sécurisation de flux vidéo. Nous avons vu que les six modules composant l'application peuvent se partager deux zones reconfigurables tout en respectant les contraintes de temps définies pour garantir une qualité de service optimale avec un traitement de 30 images par secondes. RecoSim nous a également permis d'évaluer des solutions partitionnées, c'est-à-dire séparant les modules en fonction des ressources nécessaires à leur implémentation. Cette solution nous a permis de réaliser un gain très important en surface par rapport à une application statique, économisant plus de 30% des DSP et des CLB et plus de 11 % des mémoires BRAM.

Nous avons vu que pour utiliser RecoSim, il est nécessaire de définir les zones reconfigurables ainsi que les tâches qui leurs sont allouées. Cette étape peut être compliquée à réaliser pour obtenir une solution efficace et requiert un effort important de la part du concepteur. Ce manque est compensé par notre flot de développement baptisé FoRTReSS, inférant automatiquement une architecture viable pour l'application, dont le fonctionnement est décrit dans le chapitre suivant.

Ces travaux ont donné lieu aux publications [81] (publiée) et [82] (acceptée).

CHAPITRE 5

FoRTReSS : un flot d'exploration d'architectures reconfigurables pour des applications temps-réel

Dans ce chapitre, nous introduirons notre méthodologie pour l'exploration d'architectures reconfigurables pour les applications temps-réel basé sur le flot FoRTReSS (*Flow for Reconfigurable archiTectures in Real-time SystemS*). Ce flot de conception, complètement intégré au sein du flot de conception reconfigurable de Xilinx, permet de déterminer l'architecture, en termes de zones reconfigurables, nécessaires à l'implémentation d'une application avec des contraintes de temps strictes. FoRTReSS est également basé sur les contributions FaRM et RecoSim afin de valider l'architecture pour un algorithme d'ordonnancement donné. Nous montrerons également comment FoRTReSS peut être utilisé pour l'exploration d'architectures dans le but de résoudre des problèmes liés aux performances du système. Le flot FoRTReSS est implémenté au sein d'un outil graphique permettant de facilement décrire l'application et paramétrer le flot.

5.1 L'approche FoRTReSS

L'approche utilisée par FoRTReSS est illustrée par la figure 5.1. Intégré dans les flots de conception existant, FoRTReSS se place après l'étape de synthèse, où toutes les tâches reconfigurables sont générées et transformées en netlists, et la phase d'implémentation, où le placement des tâches sur le FPGA est effectué (*floorplanning*). Les entrées nécessaires à FoRTReSS sont les netlists ainsi que les rapports de synthèse des tâches reconfigurables, issues de la phase de synthèse. FoRTReSS a également besoin de connaître l'application, représentée par un diagramme orienté acyclique (de l'anglais *Directed Acyclic Graph*, DAG) ainsi que des informations de temps de l'application (typiquement, les temps d'exécution des tâches ainsi que leurs échéances). Dans cette section, nous présenterons FoRTReSS et les architectures ciblées par les applications reconfigurables dynamiquement.

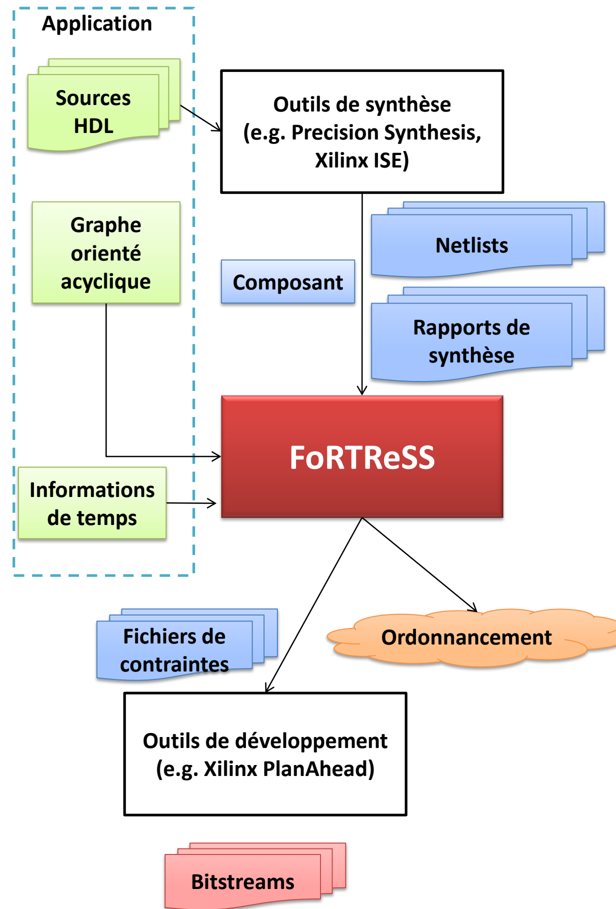


FIGURE 5.1 – *L'approche FoRTReSS*

5.1.1 Le flot FoRTReSS

La figure 5.2 illustre le fonctionnement du flot FoRTReSS. Sa conception modulaire permet au concepteur de facilement modifier les paramètres de la solution, facilitant ainsi l'exploration d'architectures. Les sections suivantes décrivent le fonctionnement de chaque étape ainsi que les différents paramètres modifiables par le concepteur à chaque étape.

5.1.1.a Détermination des zones reconfigurables pour chaque tâche

La première étape nécessaire à la génération de l'architecture consiste en la détermination de zones reconfigurables capables d'accueillir les tâches de l'application. Le fait de pouvoir accueillir ou non une tâche est déterminé uniquement par rapport aux ressources nécessaires à la tâche et aux ressources fournies par la zone reconfigurable. Cet ensemble de zones reconfigurables de référence est inféré à partir des rapports de synthèse des tâches qui permettent de déduire le nombre de ressources physiques nécessaires à la tâche. Une représentation du FPGA ciblé est parcourue pour en déduire l'ensemble des zones qui fournissent suffisamment de ressources pour implémenter la tâche étudiée. Les zones de l'ensemble de référence sont donc placées sur le FPGA, mais sans prendre en considération un éventuel chevauchement de ces zones.

Les zones sont choisies de manière à ce que les ressources contraintes par la zone doivent

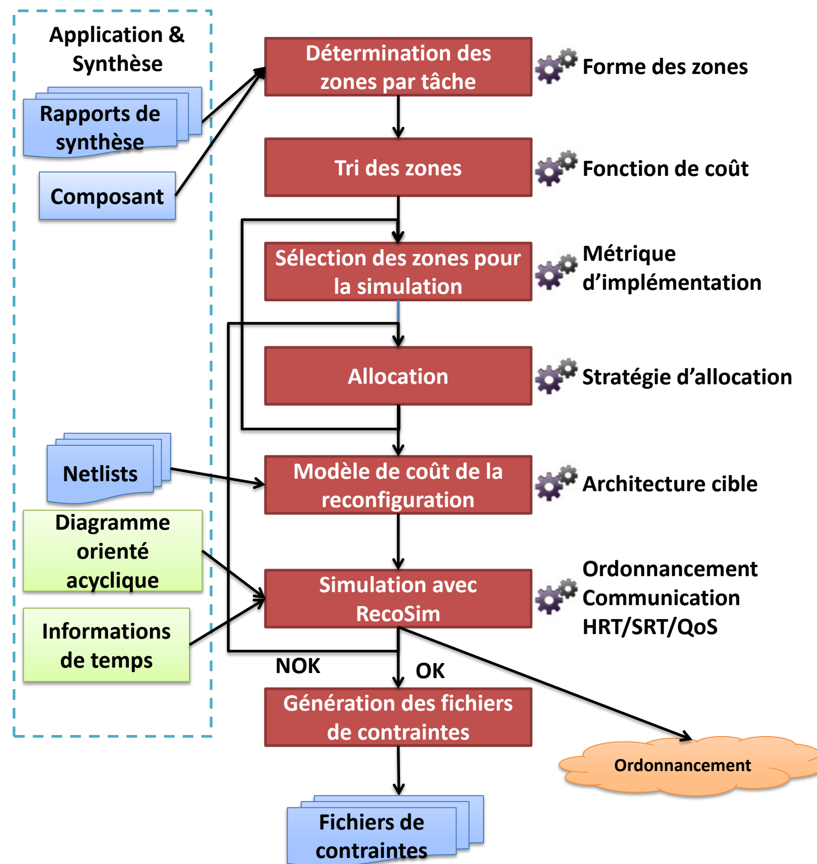


FIGURE 5.2 – *Le flot FoRTReSS*

être le plus proche possible des ressources nécessaires à la tâche. En d'autres termes, l'outil essaye de limiter au maximum la fragmentation interne. Parfois, il est toutefois difficile de ne pas gâcher de ressources au sein d'une zone reconfigurable. C'est typiquement le cas lorsque des ressources de types différents comme BRAM et DSP doivent être présentes dans la zone reconfigurable. Il est alors souvent nécessaire de contraindre plus de colonnes de CLB que nécessaire pour faire la jonction entre ces types de ressources, plus rares au sein des FPGA. De plus, lorsque la zone peut avoir une taille optimale, il est possible que l'implémentation échoue. Cela est dû aux ressources nécessaires au routage qui ne peuvent être inclues dans les rapports de synthèse. Pour cela, nous avons défini un paramètre qui permet de régler la marge de ressources à prendre en compte lors de la définition des zones reconfigurables.

Les formes de zone reconfigurables autorisées lors de la recherche sont illustrées par la figure 5.3. On distingue deux types de zones reconfigurables : les rectangulaires, dont la forme est la plus basique qui peut être définie dans l'outil de floorplanning PlanAhead de Xilinx, et les zones en forme de L, définies ainsi pour optimiser l'utilisation des ressources par la tâche et donc limiter la fragmentation interne et le temps de reconfiguration. Les zones reconfigurables doivent respecter la granularité de reconfiguration : leur hauteur doit être le multiple d'un domaine d'horloge et elles ne peuvent pas être placées à cheval sur plusieurs domaines d'horloge partiellement (cas 2 de la figure). Les zones en forme de L donnent lieu à plusieurs variantes (cas 3 à 5) selon comment est effectué l'ajout de ressources par rapport à une zone rectangulaire. Néanmoins, de telles zones sont générées uniquement

s'il n'est pas possible de créer une zone rectangulaire au même endroit avec le même nombre de ressources. Le cas 6 met en évidence le fait que les deux parties d'une zone en forme de L doivent absolument communiquer entre elles pour éviter au maximum les problèmes liés au routage. Enfin, dans le cas 7, certaines ressources placées au milieu de la zone (et non aux extrémités comme pour les zones en L) ne sont pas complètement contraintes. Cela peut être le cas de mémoires BRAM incluses au milieu de ressources CLB. Nous acceptons ces zones, mais nous considérons que les ressources non utilisées par la zone ne doivent pas servir à d'autres zones reconfigurables, toujours pour des raisons de routage difficile pour des zones imbriquées. Toutefois, ces ressources ne sont pas perdues, mais seulement utilisables par la partie statique du système.

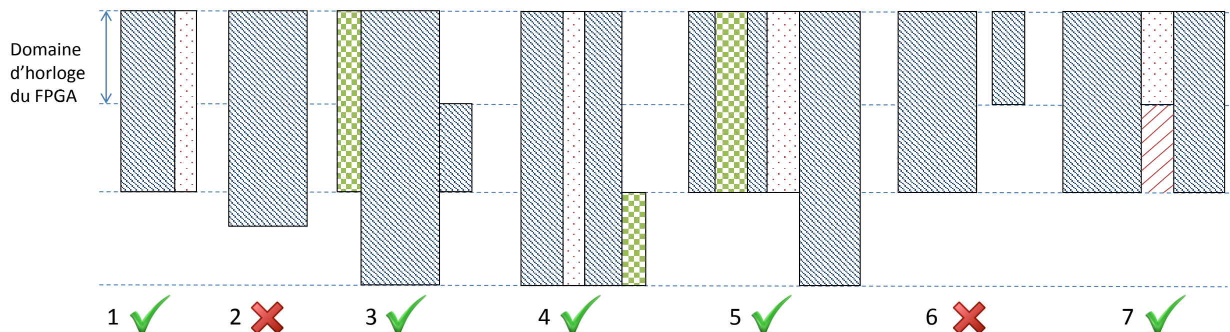


FIGURE 5.3 – *Formes autorisées pour les zones reconfigurables*

La liste des formes de zones acceptées par le flot n'est pas exhaustive et il est tout à fait possible de prendre en compte des formes différentes. Nous pensons en particulier aux FPGA 3D qui émergeront d'ici quelques années. Pour le moment, seule la technologie de *Stacked Silicon Interconnect* (SSI) de Xilinx propose une technologie "2.5D" avec le plus gros FPGA actuel, le Virtex-7 2000T et ses 6,8 milliards de transistors (2 millions de cellules logiques). Le SSI consiste en une interconnexion de dices empilés, mais vus par le concepteur comme un seul composant, ce qui ne fait pas de différence pour l'utilisation de notre modèle.

Notons que le flot garde une trace des ressources pour lesquelles ont déjà été recherchées les zones reconfigurables correspondantes, ce qui permet de gagner en temps d'exécution lorsque des tâches ont des besoins en ressources identiques. L'égalité des ressources est traitée à la granularité de reconfiguration, c'est-à-dire en nombre de colonnes de ressources. En effet, même si deux tâches ont des besoins très légèrement différents en termes de LUT et de registres, elles pourront générer des zones reconfigurables identiques puisqu'il n'est pas possible de contraindre moins qu'une colonne de ressources à la fois.

TABLEAU 5.1 – *Paramètres de la détermination des zones reconfigurables*

Nom du paramètre	Type
Choix de zones non rectangulaires	booléen
Marge de ressources pour le routage	entier positif

5.1.1.b Tri des zones reconfigurables

Une fois que l'ensemble des zones reconfigurables compatibles avec les tâches de l'application est déterminé, il est nécessaire de pouvoir estimer la qualité des zones reconfigurables choisies afin de savoir lesquelles privilégier pour la simulation. Pour cela, nous avons défini une fonction de coût estimant la qualité des zones reconfigurables qui nous permettra ainsi de les trier. La fonction de coût est donnée par l'équation (5.1) :

$$Coût_{zone} = k_1 * Coût_{forme} + k_2 * Coût_{compatibilité} \tag{5.1}$$

Cette fonction de coût est articulée autour de deux composantes. La première pénalise la forme de la zone reconfigurable. En effet, nous avons vu dans la section précédente que nous pouvions gérer différentes formes de zones reconfigurables. Toutefois, plus la forme est complexe, moins la phase de routage lors de l'implémentation aura de chances de réussir. Nous avons déjà limité les probabilités de blocage en ne gérant pas de formes plus complexes que des formes en L. Cette pénalité est réalisée en comptant le nombre d'angles droits que comporte la zone. La valeur minimale de cette composante de coût est donc minimale pour des zones rectangulaires. Si la solution ne propose que des formes complexes pour les zones reconfigurables en dépit de la pénalité qui leur a été infligée, c'est qu'il n'y a pas d'autre solution viable pour cette application sur le composant spécifié. Il est alors peut être nécessaire de considérer des FPGA de plus grande capacité.

La seconde composante de la fonction de coût correspond à la compatibilité de la tâche avec l'application ciblée. En effet, les zones reconfigurables sont initialement déterminées par rapport à une seule tâche et pour laisser le maximum de liberté à l'ordonnanceur durant la phase de simulation, il est nécessaire d'avoir des zones reconfigurables qui peuvent accueillir le plus grand nombre de tâches possible. En d'autres mots, plus la zone est grande, plus elle pourra accueillir de tâche et plus grandes seront les chances de réussite lors de la simulation. Cette composante est donc calculée par rapport au nombre de tâches que l'application ne peut pas accueillir. Ainsi, moins la zone peut accueillir de tâches, plus le coût est grand, ce qui est bien cohérent avec l'évolution souhaitée de la fonction de coût.

Ces deux composantes peuvent être pondérées par les paramètres k_1 et k_2 afin de donner plus d'importance à l'une ou à l'autre des métriques. Nous avons choisi de ne privilégier aucune composante par défaut en leur assignant des poids égaux.

Tableau 5.2 – *Paramètres du tri des zones reconfigurables*

Nom du paramètre	Type
Poids du paramètre *Forme* du coût	entier
Poids du paramètre *Cohérence* du coût	entier

5.1.1.c Sélection des zones reconfigurables pour la simulation

Durant cette étape, FoRTReSS détermine l'ensemble des zones reconfigurables qui feront partie de la simulation. Lors de la première exécution, cet ensemble ne contient qu'une seule zone reconfigurable. À chaque fois qu'une simulation échoue, la taille de cet ensemble est incrémentée. Les zones reconfigurables sont choisies par rapport à leur coût, qui a été calculé lors de l'étape précédente, tout en respectant les contraintes suivantes :

1. Toutes les tâches doivent pouvoir être placées sur au moins une zone reconfigurable.
2. Les zones reconfigurables choisies ne doivent pas se chevaucher.
3. La fragmentation externe doit être limitée.

Les deux premières contraintes doivent absolument être satisfaites alors que la troisième n'est qu'une direction donnée pour estimer la qualité de la solution. En effet, la fragmentation externe représente la distance qui existe entre les zones reconfigurables. En choisissant de réduire au maximum la fragmentation externe de la solution, on réduit la longueur de fils nécessaire à l'interconnexion des zones reconfigurables ce qui se traduit par une amélioration des performances globales du système. La fragmentation externe est calculée comme la somme des distances de Manhattan entre les zones reconfigurables pour représenter le plus fidèlement possible les connexions issues de l'étape de routage de l'implémentation.

Concernant la fragmentation externe, deux points de vue s'opposent : celui où les zones reconfigurables doivent être placées le plus près les unes des autres et celui où il doit subsister un espace entre deux zones voisines. Le second point de vue prend en considération la congestion introduite au niveau des interconnexions lorsque les zones sont trop proches les unes des autres [65]. Nous avons préféré laisser ce choix au concepteur : un des paramètres de cette étape du flot permet de passer de l'un à l'autre des points de vue en autorisant ou en interdisant de coller les zones reconfigurables les unes aux autres.

Enfin, nous avons choisi la fragmentation externe comme représentative de la qualité du placement. Nous considérons ici que toutes les zones reconfigurables communiquent entre elles, par exemple par le biais d'un bus partagé, ce qui n'est pas forcément le cas. En effet, les zones reconfigurables peuvent être connectées en point-à-point. Ainsi, il peut être intéressant de forcer le placement de ces zones l'une à côté de l'autre, peut-être au détriment de la fragmentation externe globale. Toutefois, à cette étape du flot, le placement des tâches sur les zones reconfigurable n'a pas été effectué et on ne connaît donc pas encore les différentes interfaces qui seront nécessaires à la zone ni quelles seront les interactions entre zones. Il s'agit d'un point à améliorer dans les versions futures du flot.

Comme nous l'avons vu au cours dans le chapitre précédent, une solution reconfigurable dynamiquement ne nécessite pas toujours moins de surface qu'une solution statique. C'est très souvent le cas lorsque l'on considère des applications dont les tâches sont hétérogènes, comme c'est le cas pour notre application de test Secure Box. Les tâches ayant des besoins en ressources limités gâcheront une grande partie des ressources proposées par les plus grosses zones reconfigurables (fragmentation interne). En particulier, les zones contenant des BRAM non utilisées par la tâche configurée représente une perte en taille de bitstream (et donc en temps de configuration) très importante, les BRAM étant les ressources avec le plus grand temps de configuration. Afin d'obtenir une solution dynamique plus sobre que la solution

statique, nous avions eu besoin de partitionner l'application. Nous allons réutiliser cette approche au sein du flot FoRTReSS.

Une fois la première solution validée par la simulation avec RecoSim, l'outil recherche un partitionnement possible des tâches dans le but de réduire la surface dynamique de la solution. Pour cela, nous classons les ressources par rapport à leurs besoins en ressources en estimant la taille du bitstream optimal de chaque tâche. FoRTReSS utilise alors deux seuils, définis par rapport à la tâche la plus conséquente (et donc la plus grosse zone reconfigurable de la solution) pour classer les ressources, comme le montre la figure 5.4. Les tâches dont la taille de bitstream est plus petite que le seuil bas défini (sur la figure, ce sont les tâches t_1 et t_2) sont trop petites pour être placées sur les plus grosses zones reconfigurables : cette association est considérée comme inacceptable. Les tâches dont la taille est plus grande que le seuil haut sont considérées comme optimales et ne pourront donc pas être placées sur les zones reconfigurables issues du partitionnement (tâches t_5 et t_6, la tâche la plus gourmande). Les tâches ne rentrant dans aucune des catégories précédentes sont considérées comme tolérables sur toutes les zones reconfigurables (tâches t_3 et t_4).

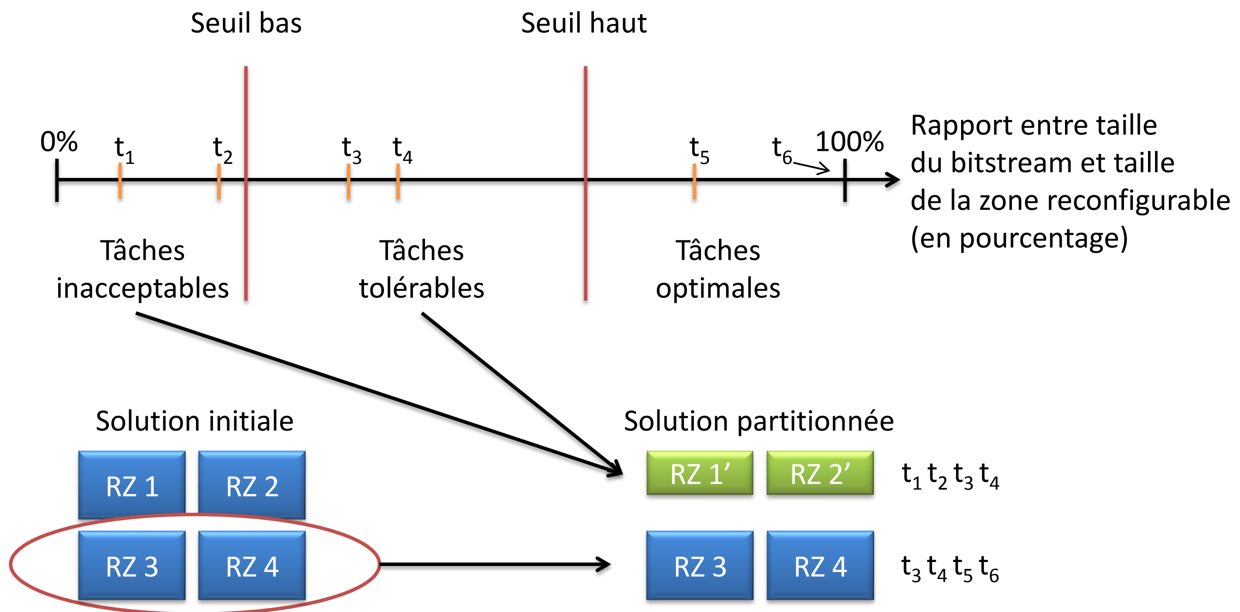


Figure 5.4 – *Principe du partitionnement des tâches avec FoRTReSS*

À partir de ce partitionnement, il est possible de définir de nouvelles zones reconfigurables, formant une solution plus apte à réduire la surface nécessaire pour l'implémentation de l'application. Pour cela, l'outil ne garde que la moitié des zones définies pour satisfaire les besoins des tâches les plus gourmandes. Ces zones seront remplacées par de plus petites zones, définies pour convenir aux besoins des tâches tolérables et inacceptables. Ces dernières ne pourront d'ailleurs plus être instanciées sur les plus grandes zones de la solution.

Actuellement, il est impossible de définir différentes implémentations matérielles pour une tâche. Par exemple, il n'est pas possible de gérer différentes versions d'un accélérateur matériel dont les couples performances contre énergie seraient différents. Nous sommes conscients que

cette limitation bride FoRTReSS et nous étudions la mise en œuvre d'une solution plus souple d'utilisation.

Tableau 5.3 – *Paramètres de la sélection des zones reconfigurables*

Nom du paramètre	Type
Limite haute des tâches inacceptables	naturel [0, 100%]
Limite basse des tâches optimales	naturel [0, 100%]

5.1.1.d Sélection des processeurs pour la simulation

Même si l'utilisation de FoRTReSS est centrée sur la détermination et le placement des zones reconfigurables, il est également possible de concevoir des systèmes mixtes incluant également des unités de calcul logicielles. Le principe de gestion des processeurs est sensiblement différent que pour les zones reconfigurables. En effet, les zones reconfigurables sont définies à partir des tâches de l'application. Pour les processeurs, on préfèrera définir les différentes tâches qui pourront s'exécuter sur l'unité, c'est-à-dire celles dont le concepteur a les exécutables compatibles avec le type de processeur (ARM, MicroBlaze...). Evidemment, il est possible de définir des temps d'exécution différents selon que l'on considère une implémentation logicielle ou une implémentation matérielle de la tâche.

Un des paramètres de FoRTReSS définit le nombre maximum de processeurs qui sont à la disposition du flot pour trouver une solution. Pour chaque solution testée, nous introduirons donc ce nombre de processeurs dans l'architecture en plus des zones reconfigurables. Il n'y a donc pour le moment pas de stratégie de gestion poussée des architectures mixtes au sein de FoRTReSS.

De même que pour les zones reconfigurables, il est impossible de différencier différentes implémentations des tâches pour un même processeur cible. Il est également supposé que tous les processeurs instanciés dans la solution soient homogènes (même type de processeur). Ces limitations devraient être levées dans les prochaines versions du flot.

Tableau 5.4 – *Paramètres de la sélection des processeurs*

Nom du paramètre	Type
Nombre maximal de processeurs	entier

5.1.1.e Allocation

Une fois que toutes les zones reconfigurables qui feront partie de la simulation ont été sélectionnées, il reste à décider quelles tâches pourront accueillir les zones reconfigurables. Cette étape d'allocation est la plus complexe puisque c'est celle qui donne la plus grande liberté au concepteur. En effet, il est rarement acceptable de laisser le maximum de liberté

pour le placement des tâches. Chaque paire tâche/zone reconfigurable générant un bitstream de configuration, le coût mémoire d'une solution explose rapidement. Il est donc dans l'intérêt du concepteur de réduire au maximum les possibilités de migration de tâches tout en respectant les contraintes de temps spécifiées pour l'application.

Nous distinguons deux phases dans la recherche d'une solution dynamique viable, avec deux fonctionnements différents pour l'étape d'allocation. Dans un premier temps, il est nécessaire de trouver le nombre de zones reconfigurables minimal nécessaire au bon fonctionnement de l'application. Durant cette phase, l'étape d'allocation consiste à placer les tâches sur toutes les zones compatibles en termes de ressources. La solution est donc la plus coûteuse en stockage de bitstreams, mais permet de déterminer efficacement le nombre minimal de zones reconfigurables nécessaires. En effet, si la solution la plus coûteuse, qui est aussi celle qui donne le plus de liberté à l'ordonnancement, ne donne pas de bons résultats, il est inutile d'essayer d'optimiser l'allocation sans avoir au préalable augmenté le nombre de zones reconfigurables. Une fois la première solution viable trouvée, FoRTReSS essaye de réduire le coût mémoire de la solution en supprimant une paire tâche/zone reconfigurable à chaque itération. On continue d'optimiser la solution jusqu'à ce que la simulation échoue de nouveau.

Afin de savoir comment doit être effectuée l'optimisation de l'allocation, nous avons mis au point les stratégies suivantes :

- Plus faible taux d'occupation : supprime la tâche ayant le plus faible taux d'occupation parmi toutes les zones reconfigurables.
- Plus grande fragmentation interne : supprime la tâche gâchant le plus les ressources de la zone reconfigurable.
- Plus grand coût mémoire : supprime la tâche ayant le bitstream le plus grand.

Ces stratégies ne fournissent pas toujours les mêmes résultats quant aux gains réalisés en taille mémoire. L'idée est donc de pouvoir exécuter les différentes stratégies afin d'en faire ressortir la stratégie la plus efficace pour une application donnée.

Le phénomène est moins prononcé pour l'allocation des processeurs. En effet, si l'on considère le cas d'un ensemble homogène de processeurs, un seul exécutable est nécessaire pour la configuration d'une tâche car il sera compatible avec toutes les unités. Le problème ne se pose alors que lorsque la solution utilise des processeurs de types différents. Les stratégies d'optimisation utilisées se concentrent alors sur le coût mémoire des exécutables ou sur le taux d'occupation de la tâche sur chacun des processeurs de l'architecture.

Dans cette étape d'allocation, les interfaces des zones reconfigurables ne sont pas du tout prises en compte. En effet, nous considérons l'interface de la zone reconfigurable comme étant un résultat de l'étape d'allocation. Il est actuellement possible de faire cohabiter des tâches qui ont chacune des interfaces différentes (par exemple FIFO, AXI maître, AXI esclave) ce qui est inefficace à la fois en termes de développement (temps passé à développer ces interfaces) qu'en termes de temps de reconfiguration. Une implémentation classique consiste à séparer la partie calculatoire de la tâche (IP core) de la partie communication. La partie reconfigurable dynamiquement doit ainsi être limitée à l'IP core dans le but de restreindre le temps de reconfiguration. Une telle architecture est illustrée par la figure 5.5. La zone reconfigurable n'est définie que pour accueillir le cœur de l'IP. Les signaux de l'interface sont alors routés vers l'interface correspondante.

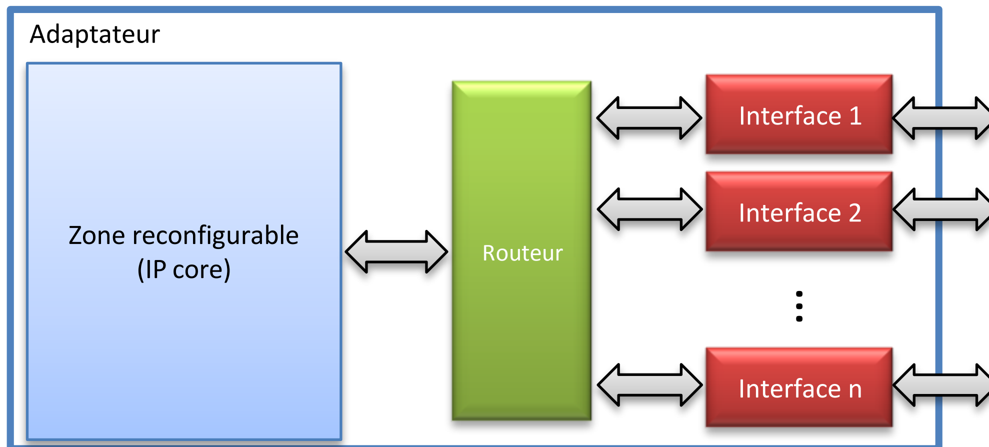


Figure 5.5 – *Gestion des interfaces de la zone reconfigurable*

Des travaux futurs devraient porter sur la prise en compte des interfaces au niveau de l'étape d'allocation, le but étant de limiter au maximum le nombre d'interfaces par zone reconfigurable. Nous comptons implémenter cette fonction comme une nouvelle stratégie d'optimisation qui serait capable d'identifier l'interface la moins utilisée par la zone et tenter de la supprimer. Cette fonctionnalité nécessite toutefois une définition des interfaces des tâches qui n'est pour le moment pas mise en place.

Tableau 5.5 – *Paramètres de l'allocation*

Nom du paramètre	Type
Stratégie d'allocation des tâches sur les zones reconfigurables	énuméré
Stratégie d'allocation des tâches sur les processeurs	énuméré

5.1.1.f Modèle de coût de la reconfiguration dynamique

À cet instant du flot, toutes les composantes de l'architecture sont définies et l'on connaît les allocations possibles pour chaque tâche. Il ne manque plus qu'une information : les temps de reconfiguration pour chaque allocation de tâche. Ces temps de reconfiguration peuvent être estimés de manière très précise à l'aide du modèle de coût développé pour FaRM et présenté en page 42.

Une des caractéristiques majeures de FaRM est la compression des bitstreams afin de réduire les temps d'exécution. Toutefois, l'estimation du taux de compression pour une utilisation dans le modèle de coût n'est pas du tout triviale. En effet, le taux de compression varie beaucoup pour une même tâche sur différentes zones reconfigurables. Nous avons tenté de corréler le taux de compression avec la fragmentation interne de la zone (la différence entre les ressources de la zone et les ressources de la tâche), mais nous n'avons pas obtenu de résultat probant. Les différences observées tiennent au routage effectué à l'intérieur de la zone reconfigurable. Celui-ci n'est pas pris en compte dans le rapport de synthèse de la tâche et varie énormément d'une tâche à l'autre.

La solution que nous proposons pour estimer le taux de compression du bitstream consiste à générer le bitstream avec l'outil de floorplanning PlanAhead de Xilinx, puis le compresser. Le projet utilisé pour la génération de bitstream est vierge et ne contient que la tâche dont le concepteur souhaite estimer le taux de compression. Cette implémentation ne contient donc aucune information sur le système complet pour réduire au maximum le temps de génération du bitstream, qui reste tout de même de l'ordre de cinq minutes pour un Virtex-6 VLX240T. Le bitstream ainsi généré ne sera donc probablement pas identique au bitstream partiel final puisque la zone reconfigurable comportera peut être une partie de routage statique, qui ne peut pas toujours être évité. Toutefois, nous considérons que cette variation du taux de compression pourra être négligée.

Si aucune netlist n'est disponible à ce stade de la conception, le concepteur peut assigner arbitrairement une valeur de taux de compression à chaque tâche. Celle-ci peut être de 26.7%, qui est le taux de compression moyen observé sur une ensemble de tâches hétérogènes, ou une autre valeur qui semble plus représentative pour le concepteur (en ajustant le taux par rapport au nombre de BRAM contraintes, par exemple). Cette approche peut également être utilisée pour obtenir une première idée de la faisabilité d'une solution reconfigurable dynamiquement. Notons également la possibilité de procéder à l'exploration d'architectures avec une estimation du taux de compression afin d'obtenir rapidement une solution dont la validité sera finalement testée avec les vrais taux de compression issus d'une implémentation avec PlanAhead. Dans le cas où la validation échouerait, il serait à la charge du concepteur de revoir à la baisse les taux de compression des différentes tâches (dont il connait maintenant les ordres de grandeur) pour être plus en adéquation avec la réalité.

Tableau 5.6 – *Paramètres du modèle de coût*

Nom du paramètre	Type
Génération de bitstream	énuméré
Période du bus de données T_{bus}	naturel
Nombre de mots par accès rafale N_{burst}	entier
Temps d'un accès rafale t_{burst}	naturel
Latence	naturel
Taille de FIFO	entier

5.1.1.g Simulation avec RecoSim

Toutes les informations nécessaires à la simulation de la solution sont connues. Le flot de conception est alors interfacé avec le simulateur RecoSim dont nous avons décrit le fonctionnement dans le chapitre précédent. Le concepteur peut spécifier le temps nécessaire à l'exécution de l'algorithme d'ordonnancement ainsi que la qualité de service (QoS) souhaitée, exprimée en pourcentage de transactions réussies par rapport au nombre de transactions réalisées. Une valeur de 100% indiquera que les tâches devront toujours respecter leurs contraintes de temps (mode temps-réel dur) alors que toute autre valeur laissera un peu de marge à l'ordonnanceur (mode temps-réel souple).

Le choix de certains paramètres du simulateur, par exemple les temps de communication entre modules, est lié à l'architecture ciblée par le concepteur, dont certains concepts seront introduits dans la section 5.1.2.

Tableau 5.7 – *Paramètres de la simulation*

Nom du paramètre	Type
Temps d'exécution de l'ordonnanceur	naturel
Qualité de service	entier [0, 100%]

5.1.1.h Génération du fichier de contraintes

À chaque fin de simulation réussie, FoRTReSS va générer un fichier de contraintes au format UCF (*Unified Constraint Format*, format de contraintes unifié). Ce fichier UCF contient les informations relatives au placement des zones reconfigurables ainsi qu'aux ressources contraintes par la zone. Ce fichier est lu par l'outil PlanAhead qui l'utilise durant la phase d'implémentation du design. Une fois le floorplan défini, le concepteur peut alors générer les bitstreams complets et partiels du système. Le listing 5.1 montre un exemple de fichier UCF généré par FoRTReSS pour une zone reconfigurable contraignant à la fois des CLB, DSP et BRAM.

```
##########################################################################
# Reconfigurable zone constraints
# This file has been automatically generated by FoRTReSS
# on Thu Jul 12 10:25:37 2012
##########################################################################

INST "nom_instance" AREA_GROUP = "pblock_instance_0";
AREA_GROUP "pblock_instance_0" RANGE=SLICE_X34Y120:SLICE_X35Y159;
AREA_GROUP "pblock_instance_0" RANGE=SLICE_X36Y120:SLICE_X37Y159;
AREA_GROUP "pblock_instance_0" RANGE=SLICE_X38Y120:SLICE_X39Y159;
AREA_GROUP "pblock_instance_0" RANGE=RAMB18_X2Y48:RAMB18_X2Y63;
AREA_GROUP "pblock_instance_0" RANGE=RAMB36_X2Y24:RAMB36_X2Y31;
AREA_GROUP "pblock_instance_0" RANGE=SLICE_X40Y120:SLICE_X41Y159;
AREA_GROUP "pblock_instance_0" RANGE=SLICE_X42Y120:SLICE_X43Y159;
AREA_GROUP "pblock_instance_0" RANGE=DSP48_X2Y48:DSP48_X2Y63;
AREA_GROUP "pblock_instance_0" RANGE=SLICE_X44Y120:SLICE_X45Y159;
```

Listing 5.1 – *Exemple de fichier de contraintes UCF*

5.1.2 Architecture cible

Comme nous venons de le voir, FoRTReSS est capable de générer un fichier de contraintes représentant le placement des zones reconfigurables sur le FPGA. Toutefois, il reste encore à définir toute l'architecture nécessaire à la gestion et à la communication des zones reconfigurables entre elles. À notre connaissance, tous les travaux déjà menés sur l'exploration

d'architectures dynamiquement ne tenaient pas compte de l'implémentation derrière le modèle, ou la considérait comme un problème indépendant. Selon nous, ces deux problèmes doivent être traités simultanément afin de représenter au mieux la réalité de l'implémentation au sein du modèle. L'utilisation de la modélisation transactionnelle pour abstraire les communications fournit la flexibilité nécessaire à la prise en charge de différents modes de communication. Notre approche est basée sur la correspondance entre l'application initiale, statique, vers une architecture reconfigurable dynamiquement.

La figure 5.6 illustre deux architectures cibles possible, à base de FIFO ou de mémoire partagée accessible par un bus. Dans les deux cas, nous considérons trois tâches pouvant chacune être implémentées sur deux zones reconfigurables. Toutefois, FoRTReSS n'est pas limité à ces deux types d'architectures, du fait de l'utilisation du TLM. Il est par exemple possible de mélanger l'utilisation de plusieurs architectures (à base de FIFO pour certaines tâches nécessitant un accès point à point, par mémoire partagée pour les tâches moins critiques). Nous avons choisi d'illustrer ces deux cas car nous pensons qu'ils représentent les deux architectures les plus communément utilisées.

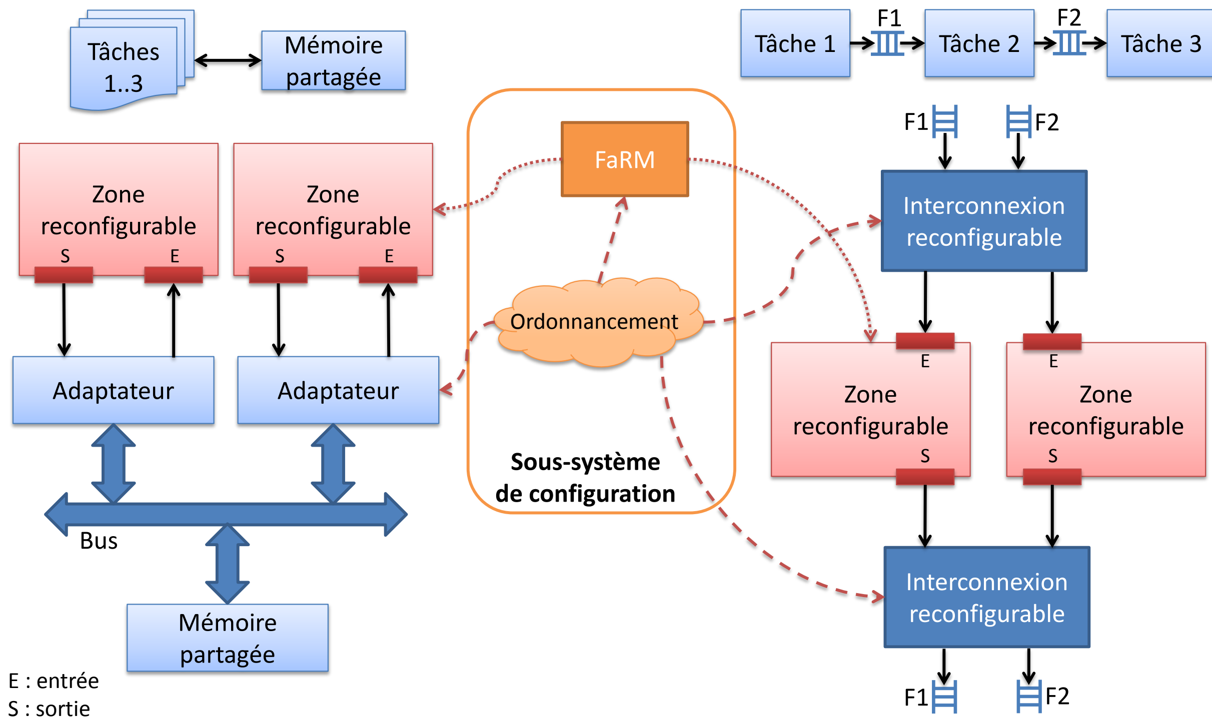


FIGURE 5.6 – *Exemples d'architectures cibles*

5.1.2.a Communications par mémoire partagée sur un bus

Nous nous plaçons ici dans le cas où l'architecture a été pensée initialement pour faire communiquer les différentes tâches par le biais d'une mémoire partagée sur un bus. Dans ce cas, les zones reconfigurables doivent pouvoir accéder au bus. Pour cela, les zones reconfigurables sont connectées à un adaptateur. Comme nous l'avons déjà précisé, il est inutile et même pénalisant de reconfigurer les interfaces. Notons qu'à l'exception des zones reconfigurables, le reste de l'architecture est flexible et peut être adaptée aux besoins de l'application.

Par exemple, il est possible de séparer les bus ou utiliser deux mémoires avec deux contrôleurs différents pour réduire la congestion.

L'outil Platform Studio de Xilinx, permettant une approche système pour la définition de l'architecture, fournissent un bon exemple d'une telle architecture. En effet, Platform Studio permet de générer facilement des patrons pour le développement d'IP, basée sur une interface IPIF (*Intellectual Property Interface*), standard développé par Xilinx et qui peut être interfacé avec des standards de bus tels que l'AXI ou PLB.

Un autre exemple de standard suivant cette architecture est le standard AXI-Stream [25]. Ce standard, faisant partie de la norme AXI. Celui-ci définit un type d'interface pour les IP ayant besoin de transferts de données rapides en mode streaming. Nativement, une IP avec une interface AXI-Stream est connectée à un DMA compatible. On retrouve alors exactement l'architecture définie dans la figure 5.6, où le rôle de l'adaptateur est joué par le DMA. Toutefois, les premiers essais effectués avec ce standard se sont révélés peu concluants, les outils et les IP fournies par Xilinx n'étant pas assez matures et encore en phase de développement.

5.1.2.b Communications par FIFO

Dans cette section, nous traitons des tâches communiquant directement avec les tâches suivantes par le biais de FIFO. Les tâches sont donc couplées de manière très forte avec les FIFO qui leurs sont connectées. Ainsi, lors du passage d'un système statique à une approche reconfigurable dynamiquement, nous avons choisi de conserver le même nombre de FIFO même si, pour des raisons de commodité, ces FIFO sont représentées deux fois dans la figure 5.6. Celles-ci sont connectées aux zones reconfigurables par le biais d'interconnexions configurables par l'ordonnanceur. Ainsi, selon le module instancié sur la zone reconfigurable, le sous-système de configuration, qui centralise l'intelligence du système et donc la connaissance de l'état des zones reconfigurables à tout instant, doit correctement configurer ces interconnexions.

5.1.2.c Sous-système de configuration

Le sous-système de configuration présenté dans la figure 5.6 est le même que celui présenté précédemment dans les chapitres concernant FaRM et RecoSim, permettant ainsi de conserver une unité dans le flot de conception depuis la phase de modélisation jusqu'à la phase d'implémentation sur carte. Le rôle d'ordonnanceur peut être tenu par un contrôleur matériel ou un processeur (MicroBlaze, PowerPC en mode standalone ou avec un système d'exploitation).

À l'heure actuelle, l'implémentation sur carte d'un système complet avec un algorithme d'ordonnancement validé par la simulation n'a pas été testée. En effet, il manque encore un travail sur le contrôle des zones reconfigurables. Par exemple, le simulateur est basé sur la communication entre les zones reconfigurables et le manager, notamment pour notifier à ce dernier la fin de l'exécution de la tâche implémentée sur la zone. Il faut donc prévoir ces signaux lors du codage ou du portage d'IP codée en HDL. Par exemple, une solution pourrait consister en l'utilisation de signaux d'interruptions qui seraient levés à chaque fin d'exécution de tâche. L'implémentation devra donc passer par la spécification des interfaces physiques des

zones reconfigurables à partir des interfaces déjà présentes dans RecoSim. Enfin, l'algorithme d'ordonnancement utilisé en simulation, actuellement un algorithme EDF, devra être porté en langage C pour une utilisation sur un processeur MicroBlaze.

5.1.3 Environnement graphique

Afin de faciliter l'utilisation conjointe du flot d'exploration d'architectures et de RecoSim, nous avons développé une interface graphique permettant de décrire graphiquement le diagramme orienté acyclique de l'application ainsi que de paramétrer chaque module et les paramètres du flot présentés dans la section 5.1.1. Pour cela, nous nous sommes basés sur l'environnement modélisation graphique Eclipse GMF (*Graphical Modeling Framework*) [83] qui permet de créer des interfaces graphiques reposant sur l'éditeur Eclipse et sur EMF (*Eclipse Modeling Framework*). EMF est un framework de modélisation ainsi qu'une infrastructure de génération de code basé sur des modèles de données structurés. Un modèle EMF est donc décrit sous forme de fichier XMI (*XML Metadata Interchange*), un standard de l'OMG dérivé du XML et spécialisé pour l'échange d'informations des métadonnées UML. EMF permet notamment la génération de code Java issu du modèle. Toutefois, nous n'utilisons pas cette fonctionnalité dans l'interface graphique.

La description du graphe orienté acyclique est effectuée dans l'éditeur comme présenté dans la figure 5.7. Ce modèle sera donc transcrit en un fichier XMI. Afin de fournir ces informations à RecoSim et à FoRTReSS, deux options sont envisageables : soit l'interprétation depuis les deux programmes du fichier XMI, soit la génération de code directement depuis l'outil graphique pour générer de nouveaux programmes qui devront être recompilés pour pouvoir être exécutés. Nous avons choisi la seconde solution, qui nous a semblé la plus facile à mettre en œuvre. Pour générer les fichiers sources nécessaires, le fichier XMI doit subir une analyse syntaxique. Nous avons choisi d'utiliser JDOM [84] qui fournit une représentation en Java d'un document XML (et donc *a fortiori* un document XMI) et qui nous permettra de récupérer toutes les informations nécessaires pour la génération du code C++ et SystemC de FoRTReSS et de RecoSim. Une fois le code généré, l'interface graphique permet de lancer la compilation puis le flot directement depuis ses menus. À la suite de l'exécution du flot, il est possible de visualiser les traces de la simulation sous ModelSim ainsi que le floorplan choisi sous PlanAhead.

Avec cette interface graphique, nous proposons aux développeurs d'applications reconfigurables dynamiquement une solution complète depuis la description du graphe orienté acyclique représentant l'application jusqu'à la simulation et la validation d'une solution viable. L'outil est compatible avec les systèmes d'exploitation Linux et Windows.

5.2 Application

L'application retenue pour mettre à l'épreuve notre méthodologie est la même que pour la validation de RecoSim, à savoir l'application Secure Box (décrite en page 67 et représentée dans l'outil en figure 5.7). Ainsi, nous pourrons comparer les résultats obtenus avec l'utilisation de RecoSim seule avec les résultats issus de FoRTReSS. Afin de bien mettre en avant les

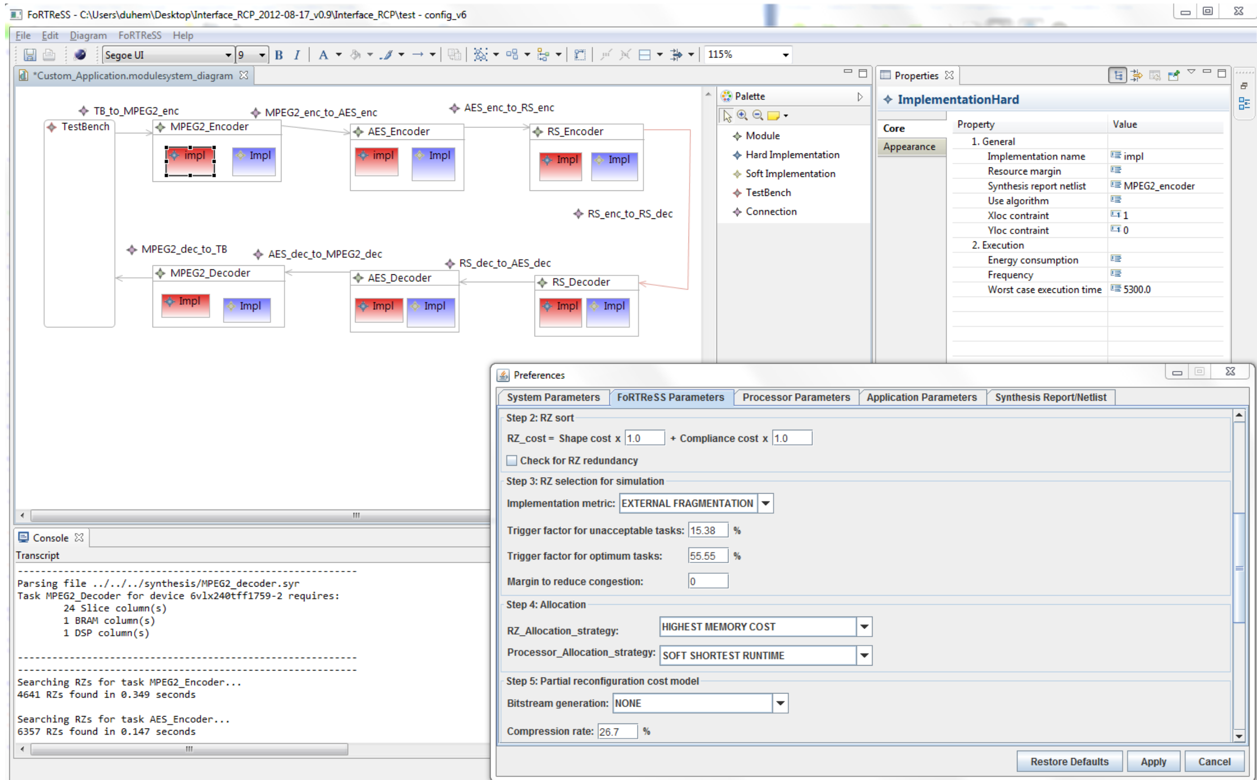


FIGURE 5.7 – *Interface graphique de FoRTReSS*

bénéfices qui peuvent être tirés de l'utilisation de FoRTReSS, nous considérerons également des cas d'utilisation où plusieurs applications Secure Box devront s'exécuter simultanément sur le FPGA. On multiplie ainsi le nombre de tâches devant s'exécuter sur le FPGA, augmentant de cette manière la complexité de définition d'une architecture et d'un ordonnancement si cela devait être fait manuellement. L'outil FoRTReSS intègre directement la possibilité de dupliquer les applications et il n'est donc pas nécessaire de décrire manuellement plusieurs fois la même application.

Comme nous l'avons déjà mentionné dans la section 4.2.1, les reconfigurations s'effectuent après chaque traitement d'image afin que les temps d'exécution des tâches soient du même ordre de grandeur que les temps de reconfiguration. Ainsi, il est nécessaire de pouvoir stocker cette image entre les modules. Une image décompressée pesant près de 760 ko, cibler une implémentation par FIFO n'est pas raisonnable. Nous préférons donc cibler une implémentation par mémoire partagée sur un bus. Les temps de récupération et de sauvegarde des données en mémoire sont cachés par l'exécution de la tâche sur les données précédentes déjà récupérées. La mémoire partagée introduit donc uniquement une latence qui peut être négligée par rapport aux temps d'exécution constatés pour les tâches de l'ordre de quelques millisecondes.

Les paramètres utilisés pour la configuration de FoRTReSS sont présentés dans le tableau 5.8. Nous nous somme concentrés sur une solution uniquement matérielle, sans processeurs dans le système autre que dans le sous-système de configuration. Les valeurs utilisées pour les seuils des tâches optimales et inacceptables durant la phase de partitionnement n'ont pas été choisies au hasard. En effet, ces valeurs permettent d'aboutir à un partitionnement

du système similaire à celui réalisé manuellement lors du chapitre précédent. Ces valeurs sont très dépendantes de l'application et il appartient au concepteur de trouver un couple de valeur permettant un partitionnement efficace. Si ces valeurs sont mal choisies, le partitionnement peut échouer ou simplement aboutir à une solution similaire à celle qui serait obtenue sans partitionnement, et donc potentiellement inefficace en termes de ressources comme nous l'avons expérimenté dans la partie résultats du chapitre précédent. Les valeurs utilisées pour le sous-système de configuration représentent une configuration classique s'exécutant à une fréquence de 100 MHz comme pour les tests de RecoSim. Nous utiliserons également une estimation du taux de compression correspondant à la valeur moyenne obtenue lors des bancs de test de l'algorithme O-RLE, à savoir 26.7%. Enfin, la solution déterminée par FoRTReSS sera validée avec les taux de compression réels récupérés grâce à la génération de bitstreams partiels avec PlanAhead.

Tableau 5.8 – *Paramétrage de FoRTReSS pour l'application Secure Box*

Etape	Paramètre	Valeur
1	Choix de zones non rectangulaires	Oui
	Marge de ressources pour le routage	5%
2	Poids du paramètre *Forme* du coût	1
	Poids du paramètre *Cohérence* du coût	1
3.a	Limite haute des tâches inacceptables	16%
	Limite basse des tâches optimales	55%
3.b	Nombre maximal de processeurs	0
4	Stratégie d'allocation	Plus grand coût mémoire
	Stratégie d'allocation des processeurs	–
5	Génération de bitstream	Validation finale uniquement
	T_{bus}	10 ns
	T_{icap}	10 ns
	N_{burst}	16 mots de 32 bits
	t_{burst}	49.2 cycles
	Latence	10 cycles
	Taille de FIFO	512 mots de 32 bits
6	Temps d'exécution de l'ordonnanceur	500 ns
	Qualité de service	100%

5.3 Résultats

Dans cette section, nous présenterons les solutions proposées par FoRTReSS pour l'application précédemment décrite. Nous montrerons également comment l'exploration d'architectures est facilitée par le flot pour permettre de résoudre des problèmes issus des premiers résultats obtenus. Ce floorplan sera validé par une ultime phase d'implémentation avec PlanAhead. Nous comparerons les résultats obtenus pour les différentes stratégies d'optimisation de l'allocation existantes et nous présenterons les temps d'exécution du flot.

5.3.1 Floorplanning et résultats de simulation

Les rapports de synthèse sont automatiquement interprétés par FoRTReSS. Les résultats obtenus, notamment concernant les tailles des zones optimales, sont évidemment les mêmes que pour la partie RecoSim et nous ne rappellerons pas ces résultats ici. Le tableau 5.9 montre les résultats en surface issus de l'exécution de FoRTReSS pour une, deux ou trois applications s'exécutant simultanément sur un FPGA.

Tableau 5.9 – *Résultats en surface issus de FoRTReSS pour un FPGA Virtex-6 LX240T*

Nombre d'applications	Type de ressource	Surface statique	Nombre de zones	Surface reconfigurable (colonnes)	Surface reconfigurable	Gain (%)		Taux d'occupation ICAP (%)
						Brut	Total	
	Slice	3750		56	2240	40.3	31.8	
1	BRAM	18	2	2	16	11.1	11.1	23
	DSP	26		1	16	38.5	38.5	
	Slice	7500		112	4480	40.3	36	
2	BRAM	36	4	4	32	11.1	11.1	47
	DSP	52		2	32	38.5	38.5	
	Slice	11250		324	12960	-15.2	-18	
3	BRAM	54	9	18	144	-166.7	-166.7	88
	DSP	78		9	144	-84.6	-84.6	

Les résultats obtenus pour l'exécution d'une seule application sont exactement les mêmes que ceux obtenus manuellement en utilisant seulement RecoSim. En effet, FoRTReSS est basé sur les concepts utilisés lors de la définition manuelle de l'architecture. Lorsque deux applications doivent être exécutées simultanément sur le FPGA, les résultats sont également très favorables à une utilisation de la reconfiguration dynamique, avec des gains similaires à ceux obtenus pour une seule application (36% de gains en ressources de type CLB).

Le floorplan correspondant est illustré par la figure 5.8. Celui-ci est composé de deux paires de zones reconfigurables. Les zones 1 et 2 sont les zones les plus grandes et accueillent les tâches optimales et acceptables à la suite du partitionnement (encodeurs et décodeurs MPEG-2 et AES). Les zones 3 et 4 sont plus petites car elles ne s'accommodent qu'aux tâches considérées avant le partitionnement comme acceptables et inacceptables (encodeurs et décodeurs AES et Reed-Solomon). Nous pouvons observer que l'allocation a été optimisée (c'est-à-dire des bitstreams ont été supprimés afin de réduire le coût mémoire de la solution) car la zone reconfigurable 1 ne peut accueillir que les tâches la première application alors que la zone 2 ne peut accueillir que les tâches de la seconde application. On a ainsi supprimé huit bitstream de la solution (deux encodeurs et deux décodeurs pour chacune des zones). Les taux d'occupation sont exprimés en pourcentage du temps de simulation et sont relativement similaires pour les zones de même type. Le temps d'exécution des tâches est bien plus important pour les petites zones, ce qui s'explique par les temps d'exécution plus importants de l'encodeur et du décodeur Reed-Solomon par rapport aux tâches MPEG-2. Les zones où ont été placées les tâches MPEG-2 sont donc libérées plus vite et peuvent ainsi passer plus rapidement soit dans l'état *Idle* soit dans l'état reconfiguration.

Notons que la solution pour deux applications est en fait la même que pour une application, mais dupliquée. Ce résultat est assez intuitif et le concepteur seul aurait sûrement fait

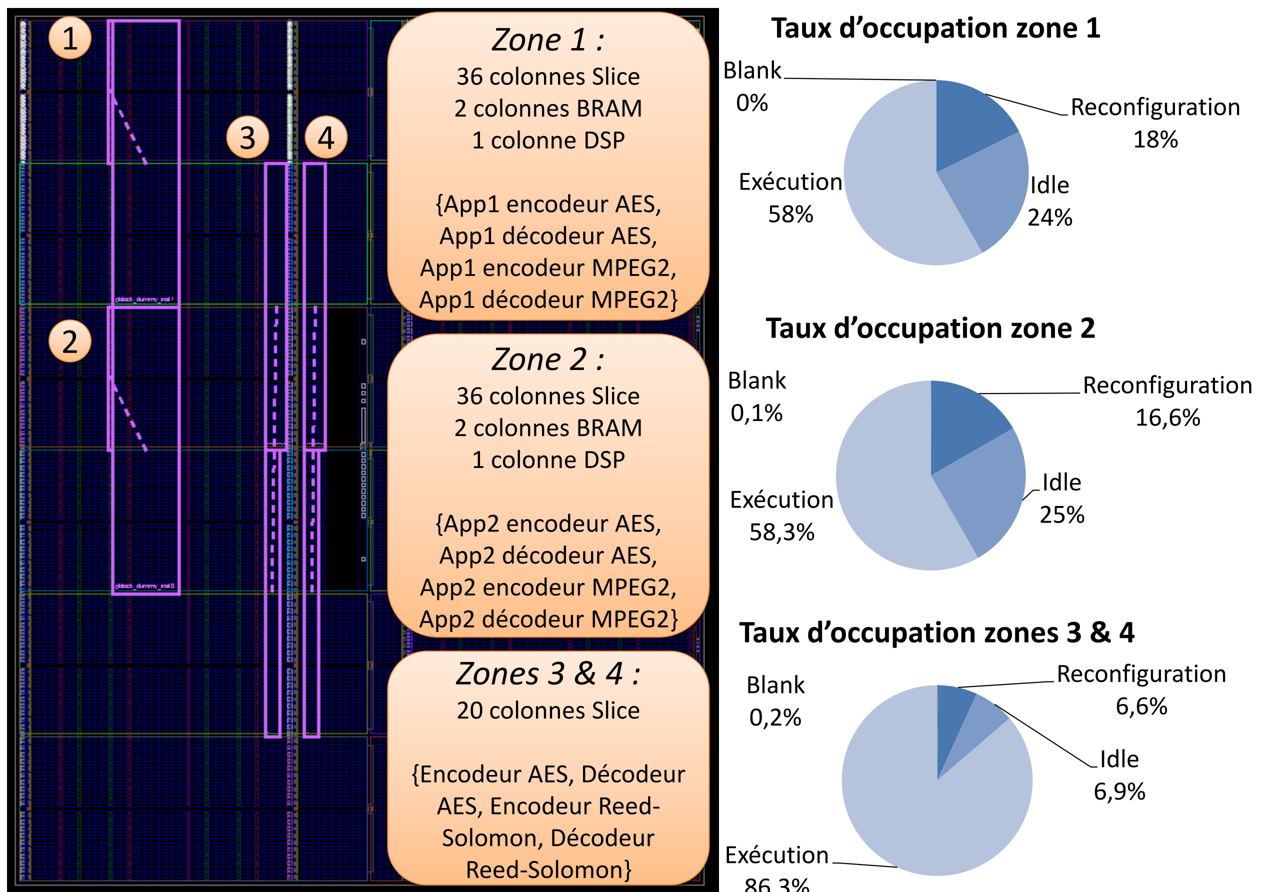


Figure 5.8 – *Floorplan issu de FoRTReSS pour deux chaînes en parallèle*

le même choix. L'approche FoRTReSS permet toutefois de réduire drastiquement le temps de conception de l'application, notamment pour un plus grand nombre de tâches ou pour la gestion de plusieurs applications hétérogènes.

Nous avons également estimé les différents algorithmes d'optimisation de l'allocation que nous avons développé dans FoRTReSS. Pour cette application, c'est l'algorithme qui supprime les pairs tâche/zone reconfigurable en fonction de leur temps d'exécution qui fournit le meilleur résultat avec plus de 53% d'amélioration du coût mémoire. Cette amélioration est calculée par rapport à la première simulation réussie, correspondant à une allocation où les tâches peuvent aller sur toutes les zones reconfigurables. Les deux autres algorithmes donnent des résultats très similaires aux alentours de 35% d'amélioration.

5.3.2 Exploration d'architectures pour trois applications

Un autre résultat issu du tableau 5.9 est que l'utilisation de la reconfiguration dynamique est à proscrire lorsque l'on a besoin de trois applications Secure Box simultanément sur le FPGA. Nous pouvions nous attendre à une amélioration constante de la surface grâce aux résultats pour une et deux applications, mais la solution reconfigurable dynamiquement est beaucoup plus gourmande en ressources que la version statique à cause de l'utilisation de neuf zones reconfigurables (à la place de deux et quatre zones respectivement pour une et

deux applications). Le problème est lié au mécanisme de reconfiguration du FPGA : en effet, il ne peut y avoir qu'un seul accès à la fois à la mémoire de configuration du FPGA. Ainsi, si plusieurs demandes de reconfiguration sont effectuées simultanément ou sont rapprochées dans le temps, la première demande est satisfaite tandis que les reconfigurations suivantes sont placées en file d'attente. C'est ce phénomène qui est mis en évidence dans ce cas. Le taux d'occupation de l'ICAP est de presque 90%, contre 23 et 40% pour une et deux applications. Le système est pénalisé par la latence avant le début de la reconfiguration effective de la tâche et ne peut donc pas respecter les échéances des tâches.

Afin de réduire le taux d'occupation de l'ICAP, deux solutions s'offrent au concepteur :

1. Augmenter la fréquence de fonctionnement du sous-système de reconfiguration afin de réduire le temps de reconfiguration. Notre contrôleur de reconfiguration, FaRM, permet de faire fonctionner l'ICAP à une fréquence de 200 MHz alors que le bus AXI peut fonctionner à 150 MHz. Dans ce cas, le facteur limitant est clairement les accès sur le bus comme nous l'avons déjà démontré et il n'est donc pas utile de monter la fréquence de l'ICAP à plus de 150 MHz. Nous nous plaçons dans le cas où l'algorithme d'ordonnancement ne peut pas prédire les configurations à l'avance, rendant inutile l'utilisation du mode de préchargement de bitstream. Enfin, il est également possible d'augmenter le nombre de mots transférés à chaque accès rafale, la transmission ayant un débit d'un mot par coup d'horloge après la phase d'initialisation. En AXI, la taille des accès rafale est limitée à 256 mots.
2. Réduire le nombre de reconfigurations en changeant la granularité d'exécution du système. Nous avions considéré jusqu'à présent qu'une reconfiguration pouvait être effectuée après chaque image traitée. En multipliant par n le nombre d'images traitées avant de pouvoir effectuer une reconfiguration, on réduit artificiellement le nombre de reconfigurations et donc le temps de reconfiguration global du système d'un facteur n. Ce gain en performances se fait aux dépends de l'empreinte mémoire du système. Toutefois, dans le cas d'une mémoire partagée externe au FPGA (par exemple une DDR3 de 512 Mo), il est tout à fait possible de stocker un nombre plus important d'images.

Ces deux solutions peuvent permettre au concepteur de satisfaire les contraintes de temps de l'application, mais cela au prix d'une consommation plus élevée, d'une fréquence d'ICAP en dehors de sa spécification ainsi que de besoins mémoire plus importants. Encore une fois, il s'agit d'un compromis à trouver entre performances et coût de la solution qui reste à la charge du concepteur : l'outil ne peut que proposer une solution allant dans le sens des directives qui lui sont fournies.

Le tableau 5.10 met en évidence les résultats obtenus en appliquant les deux solutions proposées. Dans le cas où nous changeons la granularité de reconfiguration (avec $n = 2$), une tâche traite deux images avant de pouvoir être reconfigurée. En utilisant l'une ou l'autre des stratégies, les résultats obtenus sont les mêmes : malgré une amélioration notable par rapport aux premiers résultats obtenus, on constate toujours une importante perte au niveau des blocs mémoires BRAM. Les résultats obtenus en combinant les deux approches sont enfin à la faveur de l'utilisation de la reconfiguration dynamique. Dans ce cas, FoRTReSS réussit à partitionner l'application ce qui se traduit par une nette réduction des besoins en

surface. Nous pouvons toutefois noter que le taux d'occupation de l'ICAP est assez faible (12%) ce qui laisse à penser qu'un meilleur compromis est possible en diminuant la fréquence de fonctionnement du sous-système de configuration.

Tableau 5.10 – *Résultats en surface pour trois applications sur un FPGA Virtex-6 LX240T*

Solution choisie	Type de ressource	Surface statique	Nombre de zones	Surface reconfigurable (colonnes)	Surface reconfigurable	Gain (%) Brut	Gain (%) Total	Taux d'occupation ICAP (%)
	Slice	11250		180	7200	36	33.2	
Stratégie 1 **ou** 2	BRAM	54	5	10	80	-48.1	-48.1	38
	DSP	78		5	80	-2.6	-2.6	
	Slice	11250		148	5920	47.4	44.6	
Stratégie 1 **et** 2	BRAM	54	5	6	48	11.1	11.1	12
	DSP	78		3	48	38.5	38.5	

Ces deux méthodes d'optimisation auraient également pu être utilisées dans les cas d'utilisation avec une ou deux applications. Toutefois, l'architecture proposée est la même avec ou sans optimisation. Le seul bénéfice tiré de ces optimisations est une empreinte mémoire plus faible, c'est-à-dire qu'il y a moins de paires tâche/zone reconfigurable possible et donc moins de bitstreams partiels. Ce gain en mémoire parait néanmoins bien faible en comparaison du coût lié au stockage de deux fois plus de données entre chaque module.

5.3.3 Validation finale

Afin de réduire le temps d'exécution du flot, nous avons choisi de n'utiliser qu'une estimation du taux de compression des bitstreams à la place d'une valeur réelle qui pourrait être calculée en utilisant le logiciel PlanAhead. Une fois le floorplan inféré par l'outil, il est possible et même recommandé d'effectuer une dernière vérification, cette fois en utilisant les valeurs réelles des taux de compression en générant les bitstreams nécessaires avec PlanAhead. Le tableau 5.11 montre les taux de compression obtenus pour chaque tâche sur les deux types de zones reconfigurables du floorplan. En effet, indépendamment du nombre d'applications considérées, il n'y a toujours que deux types de zones reconfigurables dans les solutions proposées par FoRTReSS. Les zones de type 1 correspondent au zones les plus larges, capables d'accueillir notamment les tâches MPEG-2, alors que les zones de type 2 sont celles qui ont été créées après le partitionnement pour n'accueillir que les tâches considérées comme acceptables et inacceptables avant de partitionner l'application. Certains taux de compression ne sont pas rapportés dans le tableau, soit parce que la zone ne peut pas accueillir la tâche (par exemple les tâches MPEG-2 ne peuvent physiquement pas être placées sur les zones de type 2), soit parce que la zone ne doit pas accueillir la tâche (après partitionnement, les zones de type 1 ne doivent plus accueillir les tâches inacceptables comme Reed-Solomon).

Les taux de compression obtenus sont globalement plus élevés que la valeur moyenne de 26,7% à l'exception de l'encodeur et du décodeur AES sur les zones de type 2. Toutefois, cette différence est compensée par la très forte compression des tâches Reed-Solomon à plus de 90% de taux de compression, puisque les besoins en ressources des tâches Reed-Solomon sont très faibles en comparaison des ressources fournies par la zone reconfigurable. Finalement,

Tableau 5.11 – *Taux de compression réels (en pourcentage de la taille initiale)*

Tâche	Zone de type 1	Zone de type 2
Encodeur MPEG-2	35.1	-
Décodeur MPEG-2	32.7	-
Encodeur AES	45.3	13.1
Décodeur AES	49.2	7
Encodeur Reed-Solomon	-	97.3
Décodeur Reed-Solomon	-	94.1
Moyenne géométrique	40	30.3

les taux de compression moyens obtenus par zone reconfigurable sont supérieurs à la valeur utilisée. La dernière simulation avec les taux de compression réels est un succès, validant ainsi la solution proposée par FoRTReSS. Les taux d'occupation obtenus sont très proches des valeurs reportées dans la figure 5.8 avec des variations de seulement 1 ou 2%.

Dans le cas où cette simulation échouerait, le concepteur aurait alors la possibilité de relancer le flot avec une valeur de taux de compression plus faible et sûrement plus en phase avec l'application considérée. Le concepteur a également la possibilité de toujours prendre les taux de compression réels en effectuant une implémentation PlanAhead à chaque nouveau couple tâche/zone reconfigurable. Cette solution est beaucoup plus longue, puisqu'une implémentation PlanAhead et la génération de bitstream dure environ 10 à 15 minutes, mais assure que la solution proposée sera physiquement réalisable.

5.3.4 Temps d'exécution de FoRTReSS

Le temps d'exécution de FoRTReSS peut être séparé en deux parties : la phase de détermination des zones reconfigurables (voir figure 5.2) et la phase d'exploration d'architectures. La détermination des zones reconfigurables est effectuée une seule fois au début du flot et dépend, pour un composant donné, à la fois du nombre de tâches et de la complexité de celles-ci (différents types de ressources nécessaires), tout en n'excédant jamais plus d'une demi-seconde de temps de recherche par tâche. Pour notre application, cette première phase de recherche et de classement des zones reconfigurables dure 4 secondes.

La partie exploration d'architectures regroupe toutes les autres étapes du flot FoRTReSS ainsi que les phases de simulation. Ainsi, ce temps d'exécution varie beaucoup selon les cas d'utilisation (5, 19 et 25 secondes respectivement pour une, deux ou trois applications), notamment à cause du nombre grandissant de simulations SystemC nécessaires à l'obtention de la solution, chacune durant quelques secondes. Il est également compliqué d'estimer le temps d'exécution de FoRTReSS puisqu'il dépend aussi du nombre d'optimisations qui seront effectuées lors de la phase d'allocation, ce qui ne peut être estimé avant l'exécution.

Les temps d'exécution mentionnés ci-dessus n'incluent pas le temps nécessaire à la validation finale de la solution avec les véritables taux de compression issus de PlanAhead. Comme nous l'avons déjà mentionné, chaque implémentation et génération de bitstream prend entre 10 et 15 minutes sur un FPGA Virtex-6 LX240T avec PlanAhead 13.3. Le temps nécessaire

à la génération de tous les bitstreams, près de 4 heures pour le cas d'utilisation avec deux applications, est très grand mais est nécessaire pour s'assurer de la validité de la solution.

Les mesures ont été effectuées sur un processeur Core 2 Quad Q9650 fonctionnant à 3 GHz avec 8 Go de RAM sans utiliser les capacités de parallélisation offertes par les multiples cœurs du processeur.

5.4 Perspectives

FoRTReSS est encore un outil de développement très jeune auquel de nombreuses fonctionnalités devront être ajoutées par la suite pour prendre en compte les besoins des concepteurs d'architectures reconfigurables dynamiquement. En particulier, il serait pertinent d'intégrer les problématiques de consommation énergétique dans FoRTReSS. Nous avons vu dans l'analyse de l'existant que des modèles de consommation existent et pourraient être utilisés pour estimer la consommation introduite par la reconfiguration dynamique. Ainsi, en cumulant cette information avec les valeurs de consommation en exécution et en état *Idle* des tâches qui composent l'application, il serait possible de quantifier la consommation globale du système.

De la même manière, il serait serait intéressant d'ajouter une nouvelle stratégie pour la sélection des zones reconfigurables sur le FPGA. Actuellement, la seule stratégie disponible consiste à grouper les zones au maximum afin de limiter la fragmentation externe du circuit. Ces considérations nous mènent à une répartition assez aléatoire des zones sur le FPGA, alors qu'il est parfois nécessaire de conserver une certaine organisation dans le placement des zones. Typiquement, si plusieurs zones doivent être connectées à un réseau sur puce (*Network-on-Chip*, NoC), il est préférable que le sommet des zones reconfigurables soient placées sur le même domaine d'horloge afin que le placement des entrées/sorties soit facilité. En allant plus loin dans ce raisonnement, il serait possible de déterminer une architecture de type matricielle, tout en tenant compte de l'hétérogénéité du composant.

A l'instar des contraintes de temps, il est possible que les spécifications du système définissent une valeur maximale de la consommation d'énergie. Nous proposons de prendre en compte la consommation une fois qu'une première architecture respectant les contraintes de temps a été trouvée. Si le système ne respecte pas les contraintes énergétiques, une solution pourrait être de résoudre le nombre de reconfigurations. Pour un algorithme d'ordonnancement donné, il s'agit de laisser plus de liberté à l'ordonnanceur en augmentant le nombre de zones reconfigurables. On perd alors en surface sur le FPGA et c'est donc un nouveau compromis à réaliser entre surface et consommation.

Toujours dans le but d'intégrer des considérations énergétiques au sein de FoRTReSS, un second mode de fonctionnement pourrait être ajouté. En effet, pour une architecture donnée, il peut être intéressant de figer l'architecture et d'explorer les différentes possibilités d'ordonnancement pour trouver un ensemble de couples consommation/qualité de service. L'exploration de l'ordonnancement correspond d'une part à différents algorithmes d'ordonnancement mais aussi à différentes implémentations (matérielles ou logicielles) d'une même tâche. Par exemple, dans le cas de tâches matérielles, plusieurs implémentations plus ou moins performantes et donc plus ou moins consommatrices d'énergie peuvent être proposées

et utilisées selon les besoins du système à un instant donné. L'exploration de cet espace de solution fournit un nuage de point montrant le compromis à faire entre consommation et qualité de service pour une architecture donnée.

Nous avons parlé dans le chapitre précédent de la nouvelle suite Xilinx, baptisée Vivado, qui intègre un outil de synthèse de haut niveau à partir de langages comme le C, le C++ ou SystemC. Il pourrait être intéressant d'intégrer complètement la suite Vivado au sein de FoRTReSS. À partir d'un code haut niveau écrit par le concepteur, il serait possible de générer le code VHDL correspondant qui pourrait être ensuite utilisé par les outils de synthèse pour générer les netlists de chaque tâche. La simulation peut alors être effectuée directement avec le code haut niveau pour accélérer le flot tout en garantissant la précision à la fois de l'algorithme simulé et des ressources correspondantes.

5.5 Conclusion

Dans ce chapitre, nous avons présenté FoRTReSS, notre méthodologie de conception d'architectures reconfigurables dynamiquement pour les applications temps-réel. FoRTReSS est parfaitement intégré dans les flots de conception existants comme celui de Xilinx et propose une architecture reconfigurable, tâche qui est actuellement entièrement à la charge du concepteur. Basé sur nos deux autres contributions FaRM et RecoSim, FoRTReSS détermine automatiquement un ensemble de zones reconfigurables à partir des rapports de synthèse de chacune des tâches de l'application. Un sous-ensemble est alors choisi pour la simulation à partir de métriques telles que la fragmentation interne ou la forme des zones reconfigurables. Ce sous-ensemble sera alors fourni au simulateur RecoSim afin de vérifier si la solution satisfait bien les contraintes de temps de l'application pour une qualité de service donnée.

Avant chaque simulation, une étape d'allocation est effectuée afin de déterminer les différents placements autorisés pour les tâches. En effet, afin de réduire le coût mémoire de la solution, FoRTReSS tente de réduire le nombre de bitstreams nécessaires à l'implémentation physique de la solution en accord avec des politiques d'optimisation prédéfinies. L'application est également partitionnée afin de réduire au maximum la surface reconfigurable nécessaire et ainsi représenter une réelle différence par rapport à une implémentation statique dans le cas d'applications hétérogènes.

FoRTReSS utilise également les modèles de coût définis pour FaRM afin d'estimer automatiquement les temps de reconfiguration pour chaque paire tâche/zone reconfigurable définie dans la solution. Les taux de compression, nécessaires à l'utilisation du modèle, peuvent être estimés en utilisant les valeurs moyennes constatées pour un ensemble hétérogène de tâches ou en utilisant l'outil d'implémentation PlanAhead afin de générer les bitstreams correspondant pour déduire les taux de compression réels. Cette solution, coûteuse en temps, permet toutefois d'assurer la validité d'une solution.

Nous avons également défini des architectures cibles possibles en vue de l'implémentation de la solution sur FPGA. À l'inverse de nombreuses autres approches, nous pensons qu'il est nécessaire de générer l'architecture et de simuler cette solution en respectant certaines contraintes introduites par l'architecture cible, notamment des contraintes temporelles. Cette

architecture se base sur le sous-système de configuration utilisant FaRM proposé dans les chapitres précédents.

Enfin, notre approche a été validée à l'aide d'une application de transmission sécurisée de flux vidéo. Nous avons montré qu'une solution reconfigurable dynamiquement avantageuse par rapport à une version statique est proposée par FoRTReSS en moins d'une minute. Les gains en surface réalisés diffèrent en fonction des ressources considérées, allant de 11.1% pour les blocs mémoire BRAM jusqu'à 38.5% pour les cellules logiques CLB. Nous avons également montré comment FoRTReSS peut être utilisé à des fins d'exploration d'architectures, facilitée par une grande liberté dans le paramétrage du flot.

Ces travaux ont donné lieu à la publication [85] (en cours de revue).

CHAPITRE 6

Conclusion générale et perspectives

Afin de faire face aux besoins grandissants en ressources de calcul des systèmes embarqués actuels, Xilinx a développé la technologie de reconfiguration dynamique qui permet de modifier le comportement d'une partie du FPGA pendant que le reste du circuit continue de s'exécuter normalement. Malgré des caractéristiques intéressantes, cette technologie peine à s'imposer en environnement industriel à la fois pour des raisons de performances et de la difficulté à valider une architecture dynamique en comparaison d'une approche statique.

Dans le cadre de cette thèse, nous avons proposé une méthodologie de conception d'architectures reconfigurables dynamiquement pour des applications temps-réel baptisée FoRTReSS qui répond aux problématiques énoncées. Ce flot s'articule autour de deux contributions majeures : FaRM et Recosim. La première contribution est un contrôleur de reconfiguration atteignant les limites du port de configuration afin de réduire au maximum le temps de reconfiguration d'une tâche. Pour cela, FaRM se base sur une architecture optimisée avec notamment une interface maître pour la récupération autonome des bitstreams et une FIFO permettant le préchargement des bitstreams. FaRM utilise également la compression des bitstreams pour réduire efficacement les temps de transferts entre la mémoire de stockage et le contrôleur. L'algorithme de compression utilisé est une amélioration du codage par plages RLE qui est efficace pour le type de données que l'on compresse avec un taux de compression moyen de 26,7% tout en conservant un débit de décompression efficace (car elle doit être effectuée à la volée sur le FPGA) d'un mot par cycle d'horloge. FaRM permet ainsi d'atteindre le débit maximal théorique du port de configuration qui est de 400 Mo/s à 100MHz et même 800 Mo/s en overclockant l'ICAP à une fréquence de 200 MHz. Les différents modes de fonctionnement de FaRM (incluant une fonctionnalité de relecture) font de FaRM une solution efficace et facile à utiliser pour optimiser les performances d'un système reconfigurable dynamiquement. Nous avons aussi développé un modèle de coût de FaRM qui permet d'estimer très précisément le temps de reconfiguration d'une tâche en fonction de la taille de la zone reconfigurable et de différents paramètres du sous-système de reconfiguration comme la fréquence de l'ICAP ou la fréquence du bus de données. Cette information est essentielle pour

pouvoir modéliser précisément le comportement d'un système reconfigurable dynamiquement et elle sera utilisée par notre seconde contribution, RecoSim.

RecoSim est un simulateur d'architectures reconfigurables pour des applications aux contraintes temps-réel. Basé sur la librairie open-source SystemC, il permet de vérifier si les contraintes de temps d'une application respectent bien une certaine qualité de service pour une architecture définie en termes de zones reconfigurables. RecoSim utilise des threads dynamiques pour représenter les modifications du comportement d'une tâche au fil du temps ainsi que la modélisation transactionnelle (TLM) des communications entre les modules. L'utilisation du TLM permet de s'affranchir de détails proches de l'implémentation et trop bas niveau lors de la conception à un haut niveau d'abstraction. Ainsi, l'architecture simulée est très flexible et facilite l'exploration d'un espace de conception très vaste durant les premières étapes du processus de conception d'un système. Nous avons également introduit un manager de reconfiguration qui centralise l'intelligence du système et prend les décisions concernant l'ordonnancement des tâches matérielles. Nous avons vu que l'utilisation de RecoSim nécessite une grande interaction avec le concepteur qui doit définir lui-même l'architecture à simuler et l'adapter aux besoins de l'application. Cette étape est entièrement automatisée dans le flot FoRTReSS.

Le flot FoRTReSS automatise l'exploration d'architectures pour des systèmes reconfigurables dynamiquement. À partir des rapports de synthèse des tâches composant l'application et d'un diagramme orienté acyclique, FoRTReSS détermine un ensemble de zones reconfigurables qui pourront être choisies pour la simulation. L'application est également partitionnée en fonction des ressources nécessaires à chaque tâche afin d'assurer une utilisation minimale des ressources du FPGA et de proposer une architecture la plus compacte possible. Les tâches sont alors placées sur les ressources disponibles pendant la phase d'allocation. Le système est alors simulé à l'aide de RecoSim en prenant en compte les temps de reconfiguration estimés à l'aide du modèle de coût de FaRM. Une fois une solution validée par la simulation, FoRTReSS va optimiser cette architecture en réduisant son coût mémoire. La solution finale générée par FoRTReSS consiste en un fichier de contraintes utilisables par les outils d'implémentation ainsi qu'en un algorithme d'ordonnancement qui assure que le système satisfait les contraintes de temps de l'application. La conception modulaire de FoRTReSS fournit au concepteur une grande liberté de paramétrage en ce qui concerne l'architecture ciblée ou les contraintes à satisfaire (par exemple, des contraintes énergétiques pourraient être prises en compte). FoRTReSS s'appuie également sur les flots de conception existants (en l'occurrence celui de Xilinx) afin de permettre au concepteur d'estimer précisément les taux de compression des bitstreams générés et donc assurer le bon fonctionnement du système lors de la phase d'implémentation. Nous avons montré qu'une solution était générée en moins d'une minute, permettant ainsi au concepteur de lancer plusieurs simulations afin de trouver un bon compromis entre les spécifications du système et l'architecture implémentée dans le FPGA. Nous avons également montré comment les résultats issus du flot FoRTReSS pouvaient être interprétés par le concepteur pour résoudre des problèmes d'infaisabilité d'une solution reconfigurable dynamiquement (ou en tout cas lorsqu'une solution dynamique n'est pas meilleure qu'une version statique en termes d'utilisation de ressources).

FoRTReSS propose ainsi au concepteur un moyen d'évaluer rapidement et très tôt dans

le processus de développement la faisabilité d'une solution reconfigurable dynamiquement et le cas échéant les bénéfices par rapport à une architecture statique plus classique mais dont la conception est maitrisée. Comme nous l'avons déjà vu au cours des chapitres de ce mémoire, de nombreuses fonctionnalités peuvent être ajoutées à FaRM, RecoSim ou FoRTReSS pour avoir une solution encore plus complète. C'est par exemple le cas de la génération par FoRTReSS d'une solution matricielle, ou au moins régulière pour le placement des zones reocnfigurables. En dehors de l'aspect fonctionnalités qui peut toujours être amélioré, nous pensons qu'il est important de prendre en compte les différents vendeurs de FPGA qui proposent la reconfiguration dynamique. À l'heure actuelle, seuls les FPGA Xilinx sont capables de reconfiguration dynamique mais Altera va également bientôt se lancer dans cette technologie. A terme, FoRTReSS devrait être capable de gérer les deux flots de conception et pourquoi pas permettre une comparaison des résultats pour plusieurs types de FPGA (non seulement des FPGA de différents constructeurs, mais pourquoi pas des FPGA de gammes différentes).

Nous avons également vu que les interfaces des tâches et des zones reconfigurables n'étaient pas encore complètement gérées par le flot et représente clairement une lacune du flot qui devrait être rapidement corrigée. Il reste toutefois à définir comment sera utilisée cette information au sein du flot : pour améliorer la solution en simplifiant le nombre d'interfaces de chaque zone reconfigurable, ou au contraire ajouter une contrainte pour le partitionnement de l'application et le placement des tâches sur les zones reconfigurables. De notre point de vue, l'interface d'une zone reconfigurable fait partie du résultat de l'étape de floorplanning et ainsi ne peut pas être vue comme une contrainte supplémentaire mais plutôt comme une recommandation et une métrique pour l'estimation de la qualité d'une solution.

Actuellement, FoRTReSS propose une solution mais l'implémentation reste majoritairement à définir. Même si nous avons proposé des exemples d'architectures cibles qui montrent la facilité d'adaptation de notre méthodologie, il reste à définir un protocole de communication entre les zones reconfigurables et le gestionnaire de reconfiguration. Une fois l'implémentation réalisée, ou au moins assistée, grâce à FoRTReSS, nous bénéficierons alors d'une solution complète pour la conception, la validation et l'implémentation d'architectures reconfigurables dynamiquement.

La technologie reconfigurable évolue très rapidement depuis ces quelques dernières années tout en changeant l'expérience utilisateur pour la conception des systèmes. FoRTReSS est compatible avec les tous derniers FPGA de la série 7 de Xilinx (Artix, Kintex et Virtex) qui ne proposent pas d'évolution architecturale majeure en comparaison des FPGA de la série 6 (Spartan et Virtex). Toutefois, cette architecture est susceptible de changer d'une génération à l'autre. Nous pensons en particulier aux FPGA 3D qui devraient émerger d'ici quelques années et qui modifieront très certainement le mode de conception d'architectures reconfigurables. FoRTReSS devra donc être modifié en conséquence pour supporter de nouveaux paradigmes de conception.

Enfin, nous pensons que l'ajout de considérations énergétiques dans FoRTReSS est très important pour en faire un outil complet pour la conception d'architectures reconfigurables utilisable dans différents domaines d'expertise. En effet, de nombreux outils ont été développés dans un but précis (estimation de consommation, floorplanning, étude de l'ordon-

nancement) mais à notre connaissance, il n'existe pas d'outils centralisant ces différentes problématiques et suffisamment exhaustif pour subvenir à tous les besoins des concepteurs de systèmes reconfigurables dynamiquement, quelle que soit leur préoccupation principale (énergie ou performances) et nous pensons que la première version de FoRTReSS telle que décrite dans ce manuscrit représente une réelle avancée pour la conception d'un outil.

Bibliographie

[1] Xilinx Inc. *Logicore IP AXI HWICAP v2.02a*, 2012.

[2] Ming Liu, Wolfgang Kuehn, Zhonghai Lu, and Axel Jantsch. Run-time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, August 2009.

[3] K. Siozios, G. Koutroumpezis, K. Tatas, D. Soudris, and A. Thanailakis. DAGGER : A Novel Generic Methodology for FPGA Bitstream Generation and Its Software Tool Implementation. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 165b – 165b, April 2005.

[4] Dirk Koch, Christian Beckhoff, and Jürgen Teich. Bitstream Decompression for High Speed FPGA Configuration from Slow Memories. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 161 –168, December 2007.

[5] Robin Bonamy, Hung-Manh Pham, Sébastien Pillement, and Daniel Chillet. Uparc - ultra-fast power-aware reconfiguration controller. In Wolfgang Rosenstiel and Lothar Thiele, editors, *DATE*, pages 1373–1378. IEEE, 2012.

[6] Imran Rafiq Quadri, Samy Meftali, and Jean-Luc Dekeyser. MARTE based modeling approach for Partial Dynamic Reconfigurable FPGAs. In *Sixth IEEE Workshop on Embedded Systems for Real-time Multimedia (ESTIMedia 2008)*, Atlanta États-Unis, 10 2008.

[7] Ali Koudri, Joël de Champeau, Denis Aulagnier, and Didier Vojtisek. Processus de développement UML/MARTE pour le codesign. 2009.

[8] David C. Black and Jack Donovan. *SystemC : From the Ground Up*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[9] Andreas Raabe. *Describing and Simulating Dynamic Reconfiguration in SystemC Exemplified by a Dedicated 3D Collision Detection Hardware*. PhD thesis, Bonn University, 2008.

[10] Kenji Asano, Junji Kitamichi, and Kuroda Kenichi. Dynamic Module Library for System Level Modeling and Simulation of Dynamically Reconfigurable Systems. *Journal of Computers*, 3 :55–62, feb 2008.

[11] Lingkan Gong and Oliver Diessel. ReSim : A reusable library for RTL simulation of dynamic partial reconfiguration. In *FPT'11*, pages 1–8, 2011.

[12] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. 3-d floorplanning : Simulated annealing and greedy placement methods for reconfigurable computing systems. In *Proceedings of the Tenth IEEE International Workshop on Rapid System Prototyping*, RSP '99, pages 38–, Washington, DC, USA, 1999. IEEE Computer Society.

[13] Kizheppatt Vipin and Suhaib A. Fahmy. Efficient region allocation for adaptive partial reconfiguration. In Tessier [86], pages 1–6.

[14] Chris Conger, Ross Hymel, Mike Rewak, Alan D. George, and Herman Lam. FPGA Design Framework for Dynamic Partial Reconfiguration. 2008.

[15] Xilinx Inc. *Virtex-5 Configuration User Guide*, 2010.

[16] Open SystemC Initiative (OSCI). *OSCI TLM-2.0 Language Reference Manual*, July 2009.

[17] Cindy Kao. Benefits of Partial Reconfiguration. *Xcell Journal*, 55 :65–67, 2005.

[18] Katarina Paulsson, Michael Hübner, Salih Bayar, and Jürgen Becker. Exploitation of Run-Time Partial Reconfiguration for Dynamic Power Management in Xilinx Spartan III-based Systems. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 699–700, September 2008.

[19] Philippe Manet, Daniel Maufroid, Leonardo Tosi, Gregory Gailliard, Olivier Mulertt, Marco Di Ciano, Jean-Didier Legat, Denis Aulagnier, Christian Gamrat, Raffaele Liberati, Vincenzo La Barba, Pol Cuvelier, Bertrand Rousseau, and Paul Gelineau. An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications. *EURASIP J. Embedded Syst.*, 2008 :1 :1–1 :11, January 2008.

[20] Katherine Compton and Scott Hauck. Reconfigurable computing : a survey of systems and software. *ACM Comput. Surv.*, 34 :171–210, June 2002.

[21] Russell Tessier and Wayne Burleson. Reconfigurable computing for digital signal processing : A survey. *J. VLSI Signal Process. Syst.*, 28 :7–27, May 2001.

[22] ARDMAHN consortium. ARDMAHN project. `http://ARDMAHN.org/`.

[23] Altera. *Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs*, July 2010.

[24] Ali Asgar Sohanghpurwala, Peter Athanas, Tannous Frangieh, and Aaron Wood. Openpr : An open-source partial-reconfiguration toolkit for xilinx fpgas. In *IPDPS Workshops*, pages 228–235. IEEE, 2011.

[25] Xilinx. *AXI Reference Guide*, March 2011.

[26] Pierre Bomel, Jeremie Crenne, Linfeng Ye, Jean-Philippe Diguet, and Guy Gogniat. Ultra-Fast Downloading of Partial Bitstreams through Ethernet. In *Proceedings of the*

22nd International Conference on Architecture of Computing Systems, ARCS '09, pages 72–83, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] Ming Liu, Zhonghai Lu, Wolfgang Kuehn, and Axel Jantsch. Reducing FPGA Reconfiguration Time Overhead using Virtual Configurations. *ReCoSoC*, 2010.

[28] Yitzhak Birk and Evgeny Fiksman. Dynamic reconfiguration architectures for multi-context fpgas. *Computers & Electrical Engineering*, 35(6) :878 – 903, 2009. High Performance Computing Architectures (HPCA).

[29] Y. Nakatani, M. Hariyama, and M. Kameyama. Architecture of a multi-context fpga using a hybrid multiple-valued/binary context switching signal. *Parallel and Distributed Processing Symposium, International*, 0 :210, 2006.

[30] Julien Lallet, Sebastien Pillement, and Olivier Sentieys. Efficient dynamic reconfiguration for multi-context embedded fpga. In *Proceedings of the 21st annual symposium on Integrated circuits and system design*, SBCCI '08, pages 210–215, New York, NY, USA, 2008. ACM.

[31] José Luis Núñez and Simon Jones. Gbit/s lossless data compression hardware. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(3) :499–510, June 2003.

[32] Dirk Koch, Christian Beckhoff, and Torresen Jim. Fine-grained Partial Runtime Reconfiguration on Virtex-5 FPGAs. In *18th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2010)*, pages 69–72. IEEE Computer Society, May 2010.

[33] Christoph Steiger, Herbert Walder, Marco Platzner, and Lothar Thiele. Online scheduling and placement of real-time tasks to partially reconfigurable devices. In *In : Proceedings of the 24th International Real-Time Systems Symposium, Cancun*, pages 224–235, 2003.

[34] Ikbel Belaid, Fabrice Muller, and Maher Benjemaa. New Three-level Resource Management Enhancing Quality of Off-line Hardware Task Placement on FPGA. *International Journal of Reconfigurable Computing (IJRC)*, pages 65–67, 2010.

[35] Juan Antonio Clemente, Carlos Gonzalez, Javier Resano, and Daniel Mozos. A hardware task-graph scheduler for reconfigurable multi-tasking systems. *Reconfigurable Computing and FPGAs, International Conference on*, 0 :79–84, 2008.

[36] Francesco Redaelli, Marco D. Santambrogio, and Donatella Sciuto. Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 519–522, New York, NY, USA, 2008. ACM.

[37] Object Management Group (OMG). Unified Modeling Language (UML). `http://www.uml.org/`.

[38] Object Management Group (OMG). MARTE profile. `http://www.omgmarte.org/`.

[39] Imran Rafiq Quadri, Samy Meftali, and Jean-Luc Dekeyser. From MARTE to dynamically reconfigurable FPGAs : Introduction of a control extension in a model based design flow. Research Report RR-6862, INRIA, 2009.

[40] Sebastien Guillet, Florent de Lamotte, Eric Rutten, Guy Gogniat, and Jean-Philippe Diguet. Modeling and formal control of partial dynamic reconfiguration. In *Proceedings of the 2010 International Conference on Reconfigurable Computing and FPGAs*, RECONFIG '10, pages 31–36, Washington, DC, USA, 2010. IEEE Computer Society.

[41] Jorgiano Vidal, Florent de Lamotte, Guy Gogniat, Jean-Philippe Diguet, and Philippe Soulard. UML design for dynamically reconfigurable multiprocessor embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1195–1200, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[42] Open SystemC Initiative (OSCI). SystemC home. `http://www.systemc.org`.

[43] Jerry Gipper. SystemC : the SoC system-level modeling language. *Embedded Computing Design*, may 2007.

[44] Antti Pelkonen, Kostas Masselos, and Miroslav Cupḱ. System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. *Parallel and Distributed Processing Symposium, International*, 0 :174b, 2003.

[45] Yang Qu, Kari Tiensyrjä, and Kostas Masselos. System-Level Modeling of Dynamically Reconfigurable Co-processors. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *Field Programmable Logic and Application*, volume 3203 of *Lecture Notes in Computer Science*, pages 881–885. Springer Berlin / Heidelberg, 2004.

[46] Andreas Schallenberg, Wolfgang Nebel, Andreas Herrholz, Philipp A. Hartmann, and Frank Oppenheimer. OSSS+R : a framework for application level modelling and synthesis of reconfigurable systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 970–975, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[47] R&D Division Embedded Hardware-/Software-Systems - System Design Methodology Group. *OSSS - A Library for Synthesisable System Level Models in SystemC : A Tutorial for OSSS 2.0*, 2007.

[48] Alisson Vasconcelos Brito, George Silveira, and Elmar Uwe Kurt Melcher. A methodology for modelling and simulation of dynamic and partially reconfigurable systems. *Dynamic Modelling*, 2010.

[49] Lukai Cai and Daniel Gajski. Transaction level modeling : an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '03, pages 19–24, New York, NY, USA, 2003. ACM.

[50] Alon Wintergreen. Using TLM virtual system prototype for hardware and software validation. June 2009.

[51] Wolfgang Ecker, Volkan Esen, Robert Schwencker, Thomas Steininger, and Michael Velten. TLM+ modeling of embedded HW/SW systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 75–80, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[52] Mentor Graphics. Veloce Emulation Systems. `http://www.mentor.com/products/fv/emulation-systems/`.

[53] Cadence. Cadence Incisive Palladium III Shortens Sharp's System Design and Verification Cycle, February 2009.

[54] David Robinson and Patrick Lysaght. Methods of exploiting simulation technology for simulating the timing of dynamically reconfigurable logic, 2000.

[55] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. Performance of partial reconfiguration in fpga systems : A survey and a cost model. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4) :36 :1–36 :24, December 2011.

[56] Juergen Becker, Michael Huebner, and Michael Ullmann. Power estimation and power measurement of xilinx virtex fpgas : Trade-offs and limitations. In *Proceedings of the 16th symposium on Integrated circuits and systems design*, SBCCI '03, pages 283–, Washington, DC, USA, 2003. IEEE Computer Society.

[57] Shaoshan Liu, Richard Neil Pittman, and Alessandro Forin. Energy reduction with runtime partial reconfiguration (abstract only). In *Proceedings of the 18th annual ACM/-SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 292–292, New York, NY, USA, 2010. ACM.

[58] Klaus Danne, Roland Mühlenbernd, and Marco Platzner. Server-based execution of periodic tasks on dynamically reconfigurable hardware. *IET Computers & Digital Techniques*, 1(4) :295–302, 2007.

[59] F. E. Sandnes and G. M. Megson. Improved static multiprocessor scheduling using cyclic task graphs : A genetic approach. In *Parallel Computing : Fundamentals, Applications and New Directions, North-Holland*, pages 703–710. Elsevier, North-Holland, 1997.

[60] Yasmina Abdeddaïm, Abdelkarim Kerbaa, and Oded Maler. Task graph scheduling using timed automata. In *Proc. FMPPTA'03*, 2003.

[61] Adam Flynn, Ann Gordon-Ross, and Alan D. George. Bitstream relocation with local clock domains for partially reconfigurable fpgas. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 300–303, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[62] Simone Corbetta, Massimo Morandi, Marco Novati, Marco Domenico Santambrogio, Donatella Sciuto, and Paola Spoletini. Internal and external bitstream relocation for partial dynamic reconfiguration. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(11) :1650–1654, November 2009.

[63] Ikbel Belaid, Fabrice Muller, and Maher Benjemaa. New three-level resource management enhancing quality of off-line hardware task placement on fpga. *EURASIP International Journal of Reconfigurable Computing (IJRC)*, April 2010.

[64] Ikbel Belaid, Fabrice Muller, and Maher Benjemaa. Static scheduling of periodic hardware tasks with precedence and deadline constraints on reconfigurable hardware devices. *Special Issue in EURASIP International Journal of Reconfigurable Computing (IJRC)*, February 2011.

[65] L. Singhal and E. Bozorgzadeh. Multi-layer floorplanning for reconfigurable designs. *IET Computers & Digital Techniques*, 1(4) :276–294, 2007.

[66] Oliver Diessel and Hossam ElGindy. Run-time compaction of fpga designs. Technical report, 1997.

[67] A. Montone, M. D. Santambrogio, F. Redaelli, and D. Sciuto. Floorplacement for partial reconfigurable fpga-based systems. *Int. J. Reconfig. Comput.*, 2011 :2 :1–2 :12, January 2011.

[68] Pritha Banerjee, Megha Sangtani, and Susmita Sur-Kolay. Floorplanning for partial reconfiguration in fpgas. In *Proceedings of the 2009 22nd International Conference on VLSI Design*, VLSID '09, pages 125–130, Washington, DC, USA, 2009. IEEE Computer Society.

[69] Kizheppatt Vipin and Suhaib A. Fahmy. Architecture-Aware Reconfiguration-Centric Floorplanning for Partial Reconfiguration. In Oliver C. S. Choy, Ray C. C. Cheung, Peter M. Athanas, and Kentaro Sano, editors, *ARC*, volume 7199 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 2012.

[70] Thilo Pionteck, Roman Koch, Carsten Albrecht, and Erik Maehle. A design technique for adapting number and boundaries of reconfigurable modules at runtime. *Int. J. Reconfig. Comp.*, 2009, 2009.

[71] Rohit Kumar and Ann Gordon-Ross. Formulation-level design space exploration for partially reconfigurable fpgas. In Tessier [86], pages 1–6.

[72] Xilinx Inc. *Virtex-6 Family Overview*, 2012.

[73] Clément Foucher, Fabrice Muller, and Alain Giulieri. Fast Integration of Hardware Accelerators for Dynamically Reconfigurable Architecture. 2012.

[74] François Duhem, Fabrice Muller, and Philippe Lorenzini. FaRM : Fast Reconfiguration Manager for Reducing Reconfiguration Time Overhead on FPGA. In Andreas Koch, Ram Krishnamurthy, John McAllister, Roger Woods, and Tarek El-Ghazawi, editors, *Reconfigurable Computing : Architectures, Tools and Applications*, volume 6578 of *Lecture Notes in Computer Science*, pages 253–260. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-19475-7_26.

[75] François Duhem, Fabrice Muller, and Philippe Lorenzini. Reconfiguration time overhead on field programmable gate arrays : reduction and cost model. *IET Computers & Digital Techniques*, 6(2) :105–113, 2012.

[76] Audrey Marchand and Silly-Chetto Maryline. Simulation et évaluation d'algorithmes d'ordonnancement temps-réel sous des contraintes de QoS. Technical report, September 2004.

[77] Xilinx, Inc. Vivado Hgih-Level Synthesis Tool. `http://www.xilinx.com/products/design-tools/vivado/index.htm`.

[78] Inc. Berkeley Design Technology. *The AutoESL AutoPilot High-Level Synthesis Tool*, 2010.

[79] Viotech. Viotech Home. `http://www.viotech.net/`.

[80] OpenCores. OpenCores. `http://opencores.org/`.

[81] François Duhem, Fabrice Muller, and Philippe Lorenzini. Methodology for designing partially reconfigurable systems using transaction-level modeling. In Jari Nurmi and Tapani Ahonen, editors, *DASIP*, pages 316–322. IEEE, 2011.

[82] François Duhem, Nicolas Marques, Fabrice Muller, Hassan Rabah, Serge Weber, and Philippe Lorenzini. Dynamically Reconfigurable Multi-Standard Video Adaptation Using FaRM.

[83] The Eclipse Foundation. Eclipse Graphical Modeling Framework (GMF). `http://www.eclipse.org/modeling/gmp/`.

[84] JDOM Project. JDOM. `http://www.jdom.org/`.

[85] François Duhem, Fabrice Muller, Willy Aubry, Bertrand Le Gal, Daniel Négru, and Philippe Lorenzini. Design Space Exploration for Partially Reconfigurable Architectures in Real-Time Systems.

[86] Russell Tessier, editor. *2011 International Conference on Field-Programmable Technology, FPT 2011, New Delhi, India, December 12-14, 2011*. IEEE, 2011.