# Déploiement d'applications parallèles sur une architecture distribuée matériellement reconfigurable

Roland Christian Gamom Ngounou Ewo


**THÈSE DE DOCTORAT DE**
**L'UNIVERSITÉ DE CERGY-PONTOISE**

Spécialité

**Science Technique Informatique Communication**

Cergy

Présentée par

**Roland Christian GAMOM NGOUNOU EWO**

Pour obtenir le grade de

**DOCTEUR de l'UNIVERSITÉ CERGY-PONTOISE**

Sujet de la thèse :

# Déploiement d'applications parallèles sur une architecture distribuée matériellement reconfigurable

Soutenue le 22 Juin 2015

devant le jury composé de :

| | | |
|---|---|---|
| M. Bertrand GRANADO | LIP6 (Paris) | Directeur de thèse |
| M. Fabrice MULLER | LEAT (Nice) | Rapporteur |
| M. Samy MEFTALI | LIFL (Lille) | Rapporteur |
| M. Bertrand Hilaire FOTSIN | ETS (Dschang) | Co-directeur |
| M. Emmanuel CHAILLOUX | LIP6 (Paris) | Examinateur |
| M. Andréa PINNA | LIP6 (Paris) | Examinateur |
| M. Benoît MIRAMOND | ETIS (Cergy) | Examinateur |
| M. Sébastien PILLEMENT | IETR (Nantes) | Examinateur |

# Déploiement d'applications parallèles sur une architecture distribuée matériellement reconfigurable

Roland Christian GAMOM NGOUNOU EWO

30 juin 2015

# Résumé

Parmi les cibles architecturales susceptibles d'être utilisées pour réaliser un système de traitement sur puce (SoC), les architectures reconfigurables dynamiquement (ARD) offrent un potentiel de flexibilité et de dynamicité intéressant. Cependant ce potentiel est encore difficile à exploiter pour réaliser des applications massivement parallèles sur puce. Dans nos travaux nous avons recensé et analysé les solutions actuellement proposées pour utiliser les ARDs et nous avons constaté leurs limites parmi lesquelles : l'utilisation d'une technologie particulière ou d'architecture propriétaire, l'absence de prise en compte des applications parallèles, le passage à l'échelle difficile, l'absence de langage adopté par la communauté pour l'utilisation de la flexibilité des ARDs, ....

Pour déployer une application sur une ARD il est nécessaire de considérer l'hétérogénéité et la dynamicité de l'architecture matérielle d'une part et la parallélisation des traitements d'autre part. L'hétérogénéité permet d'avoir une architecture de traitement adaptée aux besoins fonctionnels de l'application. La dynamicité permet de prendre en compte la dépendance des applications au contexte et de la nature des données. Finalement, une application est naturellement parallèle.

Dans nos travaux nous proposons une solution pour le déploiement sur une ARD d'une application parallèle en utilisant les flots de conception standard des SoC. Cette solution est appelée MATIP (*MPI Application Task Integreation Platform*) et utilise des primitives du standard MPI version 2 pour effectuer les communications et reconfigurer l'architecture de traitement. MATIP est une solution de déploiement au niveau de la conception basée plate-forme (PBD).

La plateforme MATIP est modélisée en trois couches : interconnexion, communication et application. Nous avons conçu chaque couche pour que l'ensemble satisfasse les besoins en hétérogénéité et dynamicité des applications parallèles . Pour cela MATIP utilise une architecture à mémoire distribuée et exploite le paradigme de programmation parallèle par passage de message qui favorise le passage à l'échelle de la plateforme.

MATIP facilite le déploiement d'une application parallèle sur puce à travers un *template* en langage Vhdl d'intégration de tâches. L'utilisation des primitives de communication se fait en invoquant des procédures Vhdl.

MATIP libère le concepteur de tous les détails liés à l'interconnexion, à la communication entre les tâches et à la gestion de la reconfiguration dynamique de la cible matérielle. Un démonstrateur de MATIP a été réalisée sur des FPGA Xilinx à travers la mise en œuvre d'une application constituée de deux tâches statiques et deux tâches dynamiques. MATIP offre une bande passante de 2,4 Gb/s et une la latence pour le transfert d'un octet de 3,43 µs ce qui comparée à d'autres plateformes MPI (TMD-MPI, SOC-MPI, MPI HAL) met MATIP à l'état de l'art.

# Remerciements

Cette thèse à bénéficié de nombreuses contributions que je souhaite mentionner ici.

Je remercie :

— MM MULLER Fabrice et MEFTALI Samy pour avoir accepté d'être les rapporteurs de cette thèse. Leurs précieuses observations, questions et remarques ont permis d'enrichir ce travail.
— Tous les membres du jury pour la peine qu'ils se donnée à examiner ce travail.

Ma reconnaissance va à :

— M. MBOUENDA Martin qui m'a formé comme étudiant, m'a coché en tant qu'enseignant puis m'a encouragé dans mes travaux de recherches.
— M. FOTSIN Hilaire Bertrand, mon co-directeur, pour ses encourageants, ses recommandations et son travail de relecture et corrections.
— M. PINNA Andréa pour ses conseils scientifiques, son travail de relecture et corrections.
— M. BOBDA Christophe de l'Université de l'Arkansas (U.S.A) qui a répondu favorablement à ma demande d'informations.

Ma profonde gratitude :

— À M. GRANADO Bertrand, mon Directeur de thèse, pour sa confiance, sa disponibilité, ses conseils scientifiques ainsi que son soutien logistique, matériel et financier sans lesquels cette thèse n'aurait pas été réalisée.
— Aux membres des équipes ASTRE du laboratoire ETIS de Cergy-Pontoise et SYEL du laboratoire LIP6 de Paris pour leur accueil et les moments conviviaux que nous avons passés ensemble.
— Au Ministère de la Coopération française (Campus France) et l'Université de Douala qui ont participé au financement de cette thèse.
— À Mme BEBOA EWO, ma mère, pour son soutien à toute épreuve, ses conseils et orientations littéraires.
— À Coretta GAMOM, mon épouse, pour ses encourageants et l'encadrement solitaire de nos enfants pendant mon absence.

# Table des matières

# Liste des tableaux

# Table des figures

# Liste des abréviations

**AES** . . . . . Advanced Encryption Standard

**ALU** . . . . . Arithmetic Logic Unit

**AMS** . . . . . Analog/Mixed-Signal

**API** . . . . . Application Programming Interface

**ASIC** . . . . . Application-Specific Integrated Circuit

**ASIP** . . . . . Application-Specific Instruction-set Processor

**CPU** . . . . . Central Processing Unit

**CRC** . . . . . Cyclic Redundancy Check

**DMA** . . . . . Direct Memory Access

**DRAM** . . . . Dynamic Random Access Memory

**DSP** . . . . . Digital Signal Processor

**EFSM** . . . . Extended Finite-State Machine

**FAMOUS** . . Flot de modélisation et de conception rapide pour les systèmes dynamiquement reconfigurable

**FIFO** . . . . . First-In-First-Out

**FPGA** . . . . Field Programmable Gate Array

**FSM** . . . . . Finite-State Machine

**GUI** . . . . . Graphical User Interface

**HAL** . . . . . Hardware Abstraction Layer

**HDL** . . . . . Hardware Description Language

**HIBI** . . . . . Heterogeneous IP Block Interconnection

**HW** . . . . . Hardware

**IC** . . . . . Integrated Circuit

**IP** . . . . . Intellectual Property

**IPC** . . . . . Inter-Processor Communication

**ITRS** . . . . . International Technology Roadmap for Semiconductors

**KPN** . . . . . Kahn Process Network

**LAN** . . . . . Local Area Network

**MAC** . . . . . Medium Access Control
**MARTE** . . . Modeling and Analysis of Real-Time and Embedded Systems
**MDA** . . . . . Model Driven Architecture
**MDE** . . . . . Model Driven Engineering
**MoC** . . . . . Model of Computation
**MOPS** . . . . Mega operations per second
**MPSoC** . . . MultiProcessor System-on-Chip
**MP-RSoC** . . Multi Processing Reconfigurable System-on-Chip
**NoC** . . . . . Network-on-Chip
**OCP** . . . . . Open Core Protocol
**OMG** . . . . . Object Management Group
**OS** . . . . . Operating System
**PC** . . . . . Personal Computer
**PBD** . . . . . Platform Based Design
**PE** . . . . . Processing Element
**PN** . . . . . Process Network
**QoS** . . . . . Quality of Service
**RAM** . . . . . Random Access Memory
**RISC** . . . . . Reduced Instruction-Set Computer
**ROM** . . . . . Read-Only Memory
**RTL** . . . . . Register Transfer Level
**RTOS** . . . . Real-Time Operating System
**RSM** . . . . . Repetitive Structure Modeling
**SA** . . . . . Simulated Annealing
**SDF** . . . . . Synchronous Data Flow
**SDL** . . . . . Specification and Description Language
**SoC** . . . . . System-on-Chip
**SRAM** . . . . Static Random Access Memory
**SW** . . . . . Software
**TDMA** . . . . Time Division Multiple Access
**TLM** . . . . . Transaction Level Modeling
**TUT** . . . . . Tampere University of Technology
**UAL** . . . . . Unité Arithmétique et Logique
**UML** . . . . . Unified Modeling Language
**VCC** . . . . . Virtual Component Codesign

**VHDL** . . . . VHSIC Hardware Description Language

**VHSIC** . . . . Very High Speed Integrated Circuit

**VLSI** . . . . . Very Large Scale Integration

**WLAN** . . . . Wireless Local Area Network

# Introduction générale

Les avancées de la technologie des semi-conducteurs favorisent la réalisation des systèmes sur puces (SoC) de plus en plus performants. Ces SoC (*System on Chip*), sont sans cesse améliorés pour augmenter la puissance de traitement, réduire la consommation d'énergie et intégrer plus de fonctionnalités dans des applications. L'augmentation des fonctionnalités est une conséquence de la loi de Moore qui prévoit d'intégrer 24 milliards de transistors à l'horizon 2024 sur une puce d'un $cm^2$ [1]. Pour réaliser le microprocesseur Pentium 4 Willamete, Intel a utilisé 42 millions de transistors, par conséquent il sera possible d'intégrer 500 Pentium IV sur une seule puce en 2024. Certaines unités de calcul telle que l'ARM7 occupent moins de 500 mille transistors ainsi, il serait possible d'intégrer plus de 40 mille ARM7 dans une puce en 2024. Les avancées technologiques et la grande densité d'intégration des transistors impactent la conception des systèmes sur puce en autorisant de nouvelles architectures, l'hétérogénéité et la dynamicité des unités de traitement. L'hétérogénéité peut être réalisée en mêlant des unités de traitement numériques et des unités de traitement analogiques sur une même puce ; mais elle peut aussi être le fruit du mélange de différentes unités de traitements numériques, chacune adaptée à un traitement spécifique de l'application.

Les progrès des technologies de reconfiguration dynamique des circuits logiques permettent au concepteur de systèmes sur puce de disposer de matériels flexibles ayant d'importantes zones reconfigurables. A titre d'exemple, le FPGA Xilinx Virtex 7 XV-2000T possède 6,8 milliards de transistors dont plus de la moitié appartient aux régions reconfigurables du circuit[2]. Une zone reconfigurable dans un FPGA regroupe un ensemble de ressources logiques qui peuvent être configurées à tout moment pour exécuter une tâche ; ainsi, les zones reconfigurables peuvent être assimilées à des unités de traitement au même titre que les processeurs. La fabrication des puces mixtes mélangeant des processeurs et des unités d'exécution matérielles dynamiquement reconfigurables permet la réalisation des systèmes de multi-traitement sur puce dynamiquement reconfigurables que nous désignons sous l'acronyme MP-RSoC(Multi Processing Reconfigurable System On Chip). Le développement d'applications parallèles pour ces MP-RSoC ne peut être efficace que si leur sont associés des outils facilitant le déploiement et la gestion de ces applications à l'instar des outils utilisés dans les HPC (High Performance Computer) pour créer et exploiter des applications parallèles.

Tandis que les outils pour HPC automatisent toutes les phases allant de la compilation d'une application parallèle à son exécution, les outils disponibles pour MP-RSoC n'offrent qu'un support limité du déploiement automatique de l'application parallèle. La conséquence est que le temps passé à mettre au point une application parallèle pour MP-RSoC est très long et les opérations à effectuer pour la déployer sont fastidieuses. Nous nous intéressons dans cette thèse à la réalisation et au déploiement d'application parallèles pour MP-RSoC.

Dans le prochain chapitre, nous revenons sur l'intérêt des MP-RSoC et sur les difficultés rencontrées lors de leur utilisation. Le deuxième chapitre est un état de l'art des outils et plateformes permettant d'utiliser les MP-RSoC. Au troisième chapitre, nous présentons MATIP, la plateforme de déploiement d'applications parallèles pour MP-RSoC que nous avons conçue et nous détaillons la conception de chacune de ses couches. Au chapitre quatre nous étendons la plateforme MATIP en introduisant le support de la reconfiguration dynamique et validons sa mise en œuvre sur un FPGA Xilinx Artix-7. Enfin nous concluons ce travail et annonçons les perspectives.

# Chapitre 1

# Problématique

## Sommaire

## 1.1 Introduction

Les besoins en puissance de traitement ne cessent d'augmenter dans un contexte fortement contraint entre autre par la limitation de l'énergie consommable, la majorité des applications pour SoC étant destinée à des équipements autonomes embarqués. Le tableau 1.1 montre par exemple que les besoins des terminaux mobiles en terme de puissance de calcul mesurée en MOPS [1], croissent de façon exponentielle [3], alors que l'énergie disponible progresse linéairement.
Ce besoin en performances a conduit à la parallélisation des traitements, multipliant ainsi le budget en énergie et en surface des puces. Une piste pour arriver à obtenir les performances induites par l'application tout en satisfaisant les autres contraintes des systèmes embarqués est de tirer partie du caractère dynamique des applications. Plus précisément, il s'agit pour nous d'utiliser la possibilité offerte par la reconfiguration dynamique des circuits électroniques afin d'adapter le système sur puce aux besoins calculatoires tout en limitant le budget en énergie [4] ou le budget en surface [5].

Les circuits électroniques dynamiquement reconfigurables sont les lieux de mise en œuvre de systèmes parallèles sur puce reconfigurables que nous avons nommés MP-RSoC pour *Multi Processing - Reconfigurable System on Chip*. Ces systèmes sur puce sont la plateforme d'exécution des applications parallèles. En analysant les besoins et les contraintes liés à la conception des SoC, nous avons identifié un ensemble de caractéristiques que le concepteur doit prendre en compte pour réaliser un MP-RSoC.

La prochaine section, rappelle les principales contraintes des systèmes embarqués ; la section 3 montre la réponse du modèle MP-RSoC à la problématique des systèmes embarqués ; la section 4 présente les outils et les types d'architectures reconfigurables dynamiquement pour MP-RSoC ; la section 5 rappelle les partitionnements du MP-RSoC ; la section 6 décrit les modèles parallèles pour MP-RSoC ; la section 7 interroge sur l'utilisation des MP-RSoC et la section 8 est une conclusion.

1. million d'opérations par seconde

Tableau 1.1: Evolution des besoins des SoC dans les terminaux mobiles [3]

| Année | 1995 | 2000 | 2005 | 2010 | 2015 |
|---|---|---|---|---|---|
| Génération | 2G | 2,5-3G | 3,5G | pre-4G | 4G |
| Standard | GSM | GPRS-UMTS | HSPA | HSPA-LTE | LTE/LTE-A |
| Débit Mb/s | 0,01 | 0,1 | 1 | 10 | 100 |
| Capa batteries (Wh) | 1 | 2 | 3 | 4 | 5 |
| Charge de calcul (GOPS | 0,1 | 1 | 10 | 100 | 1000 |
| #Coeurs programmables | 1 | 2 | 4 | 8 | 16 |

## 1.2 Les principales contraintes des systèmes sur puce

**Le système sur puce (SoC)** désigne une plateforme de traitement qui est confinée dans une puce et intégrée dans un équipement pour gérer une partie ou la totalité des traitements automatisés. Il peut contenir des composants numériques, analogiques, mixtes, et souvent les composants de radio-fréquence. Une utilisation typique des SoC relève du domaine des systèmes embarqués. Les systèmes sur puce doivent être capables de satisfaire un certain nombre de contraintes qui sont définies par les besoins de l'application et par leur environnement d'utilisation.

### 1.2.1 Augmenter la puissance de traitement

Les nouvelles applications telles que la réalité augmentée, la biométrie dans les systèmes de sécurité, demandent de disposer d'équipements capables de traitements complexes comme l'incrustation en temps réel d'un objet dans une séquence vidéo ou l'identification automatique par la reconnaissance faciale. Ces applications sont généralement embarquées sur des équipements miniatures comme les *Google glass* [6] et exigent de disposer d'une puissance de calcul de plusieurs centaines de GOPS pour exécuter convenablement les applications.

Dans plusieurs domaines les traitements gagnent en complexité ; le temps est loin où il suffisait d'avoir un capteur de température et des valeurs de consigne dont le franchissement déclenchait une action. Aujourd'hui, les systèmes qui contrôlent la température d'un procédé peuvent en plus, calculer le gradient pour déclencher encore plus d'actions.

Pour améliorer le trafic des informations et la qualité de service sur un réseau, certains routeurs sont capables de compresser plusieurs paquets d'informations ou de donner la priorité à quelques paquets après un sondage du trafic. Dans le domaine médical, les professionnels ont besoin d'appareils d'imagerie avec une meilleure résolution, capable de compresser, de stocker et de transmettre des images sans perte de qualité. Ces besoins impliquent de doter ces appareils de plus de puissance de traitement pour la mise en œuvre d'algorithmes complexes [7] réalisant ces objectifs.

Le tableau 1.1 montre que la dernière génération (4G) de protocole de communication

sans fil requiert 1000 GOPS tandis que la 1G n'en demandait que 0,1 GOPS. Ces applications illustrent le besoin d'une plus grande puissance de calcul dans les systèmes sur puce.

### 1.2.2 La flexibilité du développement

Le temps de mise sur le marché d'une nouvelle application des systèmes sur puce doit être le plus court possible, afin de profiter pleinement des nouvelles idées et de nouveaux concepts. Ceci exige la disponibilité d'outils permettant d'assister les concepteurs dans toutes les phases de développement des nouveaux produits.

### 1.2.3 Économiser les ressources matérielles et l'énergie

Plusieurs facteurs militent en faveur d'une réduction de l'énergie consommée par les SoC. Les industriels souhaitent rallonger la durée de fonctionnement des systèmes alimentés par batterie pour accroître l'autonomie et donner plus de confort aux utilisateurs. Une puce qui contient un grand nombre de transistors en activité consomme de l'énergie et dissipe de la chaleur proportionnellement au nombre de transistors. Il existe alors un risque de surchauffe et de destruction. Contrôler intelligemment le nombre de transistors actifs dans le SoC est un moyen pour réduire ce problème.

La possibilité de reconfigurer dynamiquement une unité de traitement permet de réutiliser les mêmes ressources logiques pour exécuter différentes fonctionnalités, entraînant du même coup une économie d'énergie et une économie des ressources logiques. A plus grande échelle, toute économie d'énergie et de ressources contribue à la réduction de la pollution de l'environnement sur notre planète.

### 1.2.4 Accomplir simultanément plusieurs tâches sur puce

Pour obtenir une grande puissance de calcul, la parallélisation des traitements est une option naturelle, elle permet en plus de réduire la fréquence du circuit en augmentant la capacité de traitement. La parallélisation des traitements peut permettre d'exécuter plusieurs applications sur le même équipement avec de bonnes performances pour chacune d'elles. Les utilisateurs de smartphones, par exemple, peuvent simultanément : recevoir leurs communications, naviguer sur Internet, écouter la musique, suivre leur itinéraire à l'aide du GPS. Pour arriver à intégrer autant de fonctionnalités sur un SoC, les concepteurs ont besoin de nouveaux modèles pour concevoir leurs applications et de nouveaux outils pour les réaliser. En pratique ceci implique de repenser totalement leurs applications pour tirer profit du parallélisme des unités d'exécution.

### 1.2.5 Un processeur sur mesure pour chaque tâche

Les différentes fonctionnalités d'une ou de plusieurs applications sollicitent des traitements très différents les uns des autres. Il est souhaitable que les unités d'exécution intégrées dans le SoC s'adaptent le plus efficacement à chaque type de traitement. Les SoC ainsi créés sont hétérogènes, favorisant la faible consommation d'énergie par l'utilisation d'unités d'exécution adaptées telles que des DSP, des VLIW, des ASIP [2] [8] ou des circuits reconfigurables. Cette contrainte demande que le SoC soit flexible, pour accepter des processeurs et des IP provenant de différentes sources, et puisse les mettre en œuvre à la demande.

### 1.2.6 La convergence

La convergence dénote le fait d'intégrer tout un ensemble d'applications dans un seul et même équipement comme le smartphone, nécessitant la prise en compte de modèles de calculs différents. En effet, ces applications ont des contraintes différentes et requièrent de très fortes capacités de calculs. Ces constats amènent à la définition d'architectures parallèles hétérogènes intégrant différents opérateurs plus ou moins flexibles, mais performants pour un domaine d'application.

### 1.2.7 Le passage à l'échelle

Les nouveaux systèmes doivent supporter le passage à l'échelle. Ceci afin de rentabiliser à long terme les développements en supportant des applications non définies lors de la conception du système. Une solution, est alors de concevoir des systèmes sur puce reconfigurables permettant l'adaptation de l'architecture.

## 1.3 Pourquoi un MP-RSoC

Nous définissons le terme MP-RSoC par rapport au terme MPSoC qui désigne dans la littérature un système sur puce ayant plusieurs unités d'exécution (multicoeurs).

**MP-RSoC** : variante de MPSoC comprenant une application parallèle déployée sur une puce contenant une architecture dynamiquement reconfigurable (ARD). Contrairement au MPSoC, les modules fonctionnels et les chemins de données du MP-RSoC peuvent être modifiés, ajoutés ou supprimés pendant leur fonctionnement. Le MP-RSoC facilite la mise à jour et l'évolution des systèmes sur puce.

Comme exemple de MP-RSoC, on peut citer le Zynq de Xilinx [9] qui est une puce contenant un processeur Cortex A9 dual core et une zone ARD, où il est possible d'exécuter des modules fonctionnels réalisés par programmation à l'aide du processeur tout en utilisant les ressources logiques reconfigurables pour définir des unités d'exécution sur mesure adaptées

2. Application-Specific Instruction set Processors

aux besoins de l'application. Un MP-RSoC permet la convergence et le passage à l'échelle du fait de la flexibilité de l'architecture.

### 1.3.1 Le Coût de fabrication des puces

La diminution de la taille des transistors va de paire avec l'augmentation des coûts de fabrication des puces. Ainsi, alors que les équipements pour la fabrication des masques en technologie 65 nm reviennent à 1,5 million de dollars US, il en coûte le double pour une fabrication en technologie 45 nm et il faudra compter 60 millions de dollars pour ceux permettant la technologie 14 nm [10]. Cette contrainte économique forte, pose le problème de la réutilisation des architectures et des codes. Les MP-RSoC peuvent donner lieu au développement de systèmes personnalisables et flexibles, fournissant des puissances de calculs proches de celles obtenues par un SoC dédié, réalisé à l'aide d'un ASIC (voir figure 1.1). La flexibilité du MP-RSoC peut favoriser la réduction des coûts lorsque le même système sur puce s'adapte à plusieurs domaines applicatifs : téléphonie mobile, automobile, imagerie, avionique, réseaux de communication, cloud, monétique, etc. La flexibilité répond de plus à des demandes spécifiques du marché des SoC comme la dynamicité [11].

### 1.3.2 La dynamicité

La dynamicité est une caractéristique de plusieurs applications, comme la radio logicielle ou la vidéo embarquée, qui nécessitent de supporter des variabilités dans leur exécution en fonction des conditions extérieures telles que les performances du canal de transmission. Ces applications sont intégrées dans des dispositifs connectés en réseau, créant l'*Everyware*[3] [12] qui, par définition, est dynamique. La dynamicité est aussi requise du fait de la vitesse d'évolution des applications et/ou des normes. l'ARD du MP-RSoC permet naturellement au matériel de s'adapter à la dynamicité de l'application embarquée.

### 1.3.3 La disponibilité

Ces dernières années les circuits logiques programmables ont connu un très grand développement et leur prix de revient baisse tandis que leurs performances augmentent, les rendant plus accessibles [13]. Les MP-RSoC sont construits autour des architectures telles que le FPGA[4] qui supporte d'ores et déjà la reconfiguration dynamique. Aujourd'hui il est possible de concevoir un MP-RSoC à la fois flexible au niveau logiciel et au niveau matériel.

3. terme inventée et utilisée par A. Greenfield pour parler des dispositifs de traitement, petits et peu coûteux, connectés en réseau (intégrés et distribués dans les objets de la vie quotidienne), visant à créer un espace quotidien intelligent.

4. Field Programmable Gate Array

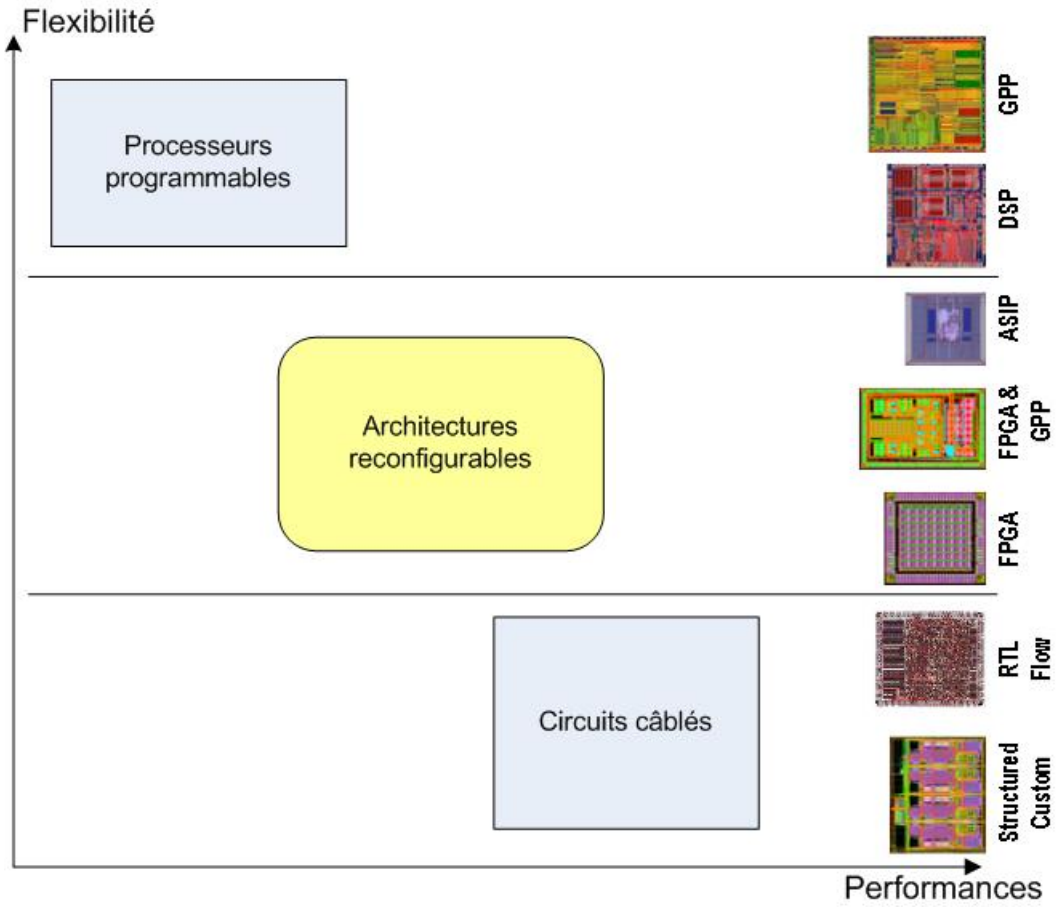



FIGURE 1.1 : Compromis entre performance et flexibilité des architectures reconfigurables

## 1.4 Les types d'ARD et les outils des MP-RSoC

Les architectures matérielles des MP-RSoC rencontrées dans la littérature sont de type grain fin (configurable au niveau du bit), ou grain épais (configurable au niveau fonctionnel) et les outils qui permettent d'y déployer des applications sont spécifiques à chacune d'elle.

**Une plateforme** est une base de travail à partir de laquelle on peut utiliser un ensemble de logiciels afin de développer une application.

### 1.4.1 ARD pour MP-RSoC

Les MP-RSoC intègrent des ARD qui peuvent être à grain fin comme le FPGA qui permet une configuration au niveau d'opérateurs booléens. Elles peuvent aussi être à grain épais comme l'architecture MATRIX [14] et proposer une configuration à base des bancs de registres, d'UAL programmables, de processeurs à jeu d'instructions variables, d'unités d'interconnexion qui opèrent sur des mots de taille fixe ou variable. Autour des années 2000 un grand engouement est observé pour le développement des MP-RSoC. Le tableau 1.2 présente un certain nombre de projets présentant des architectures de traitements originales. Malheureusement, la révolution dans l'utilisation des MP-RSoC n'a pas eu lieu et les solutions

proposées sont en majorité restées dans les cartons ou ont été abandonnées à cause de la relative difficulté pour leur mise en œuvre et de la résistance au changement.

Le tableau 1.2 précise pour chaque projet : l'architecture d'interconnexion des unités d'exécution, leur granularité, le mode de déploiement de l'application embarquée sur le matériel et le domaine d'utilisation de la plateforme ou de l'environnement proposé. Le type de ressources de base et l'organisation matérielle de l'ARD ont un grand impact sur le développement des outils de déploiement de l'application. Des architectures complexes ou sophistiquées ont difficilement de bons outils qui leurs sont associés. Une solution consiste à avoir une architecture matérielle simple et un outil de déploiement qui peut lui-même configurer l'ARD suivant les besoins de l'application [15].

### 1.4.2 Outils de programmation et de configuration du MP-RSoC

Les structures de programmation pour architectures reconfigurables sont très dépendantes de l'organisation matérielle et de la granularité qui favorise l'utilisation de différents langages. Pour les plateformes MorphoSys, MATRIX, et REMARC le langage utilisé est l'assembleur [15]. La plateforme DART offre un atelier logiciel comprenant un compilateur C, un synthétiseur et un simulateur dédié [16]. D'autres plateformes disposent de flux de conception automatique à partir du langage HDL ou d'un langage de programmation de haut niveau comme le *Native Mapping Language* (NML) [17] pour la plateforme XPP [5].

Les environnements diffèrent par l'approche utilisée pour le mappage technologique, le placement et le routage. Le mappage technologique est relativement simple pour les architectures à grain épais dans lesquelles les opérateurs sont déjà prédéfinis ; il est plus complexe pour les architectures à grain fin telles que les FPGA.

Pour améliorer la performance de l'application, plusieurs projets MP-RSoC ont proposé un nouveau langage de programmation et un nouveau compilateur/synthétiseur pour mieux cibler leur architecture. Ce dernier est chargé d'utiliser la reconfiguration du matériel pour optimiser les unités de traitement comme dans [18] et [19].

Définir un paradigme de programmation pour un MP-RSoC est une tâche difficile. Il n'y a pas une méthode générale qui couvre la programmation de toutes les unités d'exécution. Les unités de type processeurs généralistes peuvent être programmées dans un langage comme le C/C++ mais d'autres unités d'exécution ayant un jeu d'instructions plus spécifiques nécessitent un langage particulier ou une extension spécifique du langage de programmation usuel. La reconfiguration dynamique peut être intégrée dans l'application à l'aide d'un compilateur ou d'un outil de déploiement ou être entièrement laissée sous le contrôle du concepteur de l'application en fonction des tâches à exécuter. Il n'existe pas aujourd'hui un modèle bien établi de déploiement d'application sur un MP-RSoC, chaque projet propose plutôt son propre modèle.

5. eXtreme Processing Platform

Tableau 1.2: Plateformes pour architectures reconfigurables trouvées dans la littérature

| **Project** | **first publ.** | **Source** | **Architecture** | **Granularity** | **Fabrics** | **Mapping** | **Intended target application** | **Language** |
|---|---|---|---|---|---|---|---|---|
| DART | 2003 | [16] | 2-D array | 8 & 16 bit | hierarchical buses | run-time reconfiguration | 3G telecommunication | C |
| PADDI-2 | 1993 | [14] | crossbar | 16 bit | multiple crossbar | routing | DSP and others | |
| DP-FPGA | 1994 | [14] | 2-D array | 1 & 4 bit, multi-granular | inhomogenous routing channels | switchbox routing | regular datapaths | |
| KressArray | 1995 | [14] | 2-D mesh | family : select pathwidth | multiple NN [6] & bus segments | (co-)compilation | (adaptable) | |
| Colt | 1996 | [14] | 2-D array | 1 & 16 bit inhomogenous | (sophisticated) | run time reconfiguration | highly dynamic reconfig. | |
| RaPID | 1996 | [14] | 1-D array | 16 bit | segmented buses | channel routing | pipelining | Rapid-C |
| Matrix | 1996 | [5] | 2-D mesh | 8 bit, multi-granular | 8NN, length 4 & global lines | multi-length | general purpose | Assembler |
| RAW | 1997 | [14] | 2-D mesh | 8 bit, multi-granular | 8NN switched connections | switchbox rout | experimental | |
| Garp | 1997 | [14] | 2-D mesh | 2 bit | global & semi-global lines | heuristic routing | loop acceleration | |
| Pleiades | 1997 | [14] | mesh / crossbar | multi-granular | multiple segmented crossbar | switchbox routing | multimedia | |
| PipeRench | 1998 | [14] | 1-D array | 128 bits | (sophisticated) | scheduling | pipelining | |
| REMARC | 1998 | [20] | 2-D mesh | 16 bit | NN & full length buses | (information not available) | multimedia | Assembler |
| MorphoSys | 1999 | [18] | 2-D mesh | 16 bit | NN, length 2 & 3 global lines | manual P&R | (not disclosed) | Assembler |
| CHESS | 1999 | [14] | hexagon mesh | 4 bit, multi-granular | 8NN and buses | JHDL compilation | multimedia | |
| DReAM | 2000 | [19] | 2-D array | 8 &16 bit | NN, segmented buses | co-compilation | next generation wireless | |
| Chameleon | 2000 | [14] | 2-D array | 32 bit | (not disclosed) | co-compilation | tele- & data communication | |
| MorphICs | 2000 | [14] | 2-D array | (not disclosed) | (not disclosed) | (not disclosed) | next generation wireless | |
| PACT XPP | 2003 | [17] | 2-D mesh | 32 bits | NN, switched connections & global lines | compiler run-time reconf | general purpose | Native NML |

## 1.5 Les options et les possibilités de mise en œuvre des MP-RSoC

Les contraintes de la conception des MP-RSoC placent le concepteur devant un ensemble de possibilités qu'il faut comprendre et analyser pour faire les bons choix.

Le concepteur d'un MP-RSoC doit faire un compromis entre le temps de développement et les performances de l'application, puis entre la souplesse de programmation des différentes tâches de l'application et leur performance. Le concepteur doit également considérer la surface occupée sur la puce par l'application et le coût du développement. Si tous les modules fonctionnels de l'application sont exclusivement réalisés à l'aide des ressources logiques de base, les performances de l'application pourraient être optimales mais le temps de développement qui est lié au coût, et la surface occupée deviennent très vite prohibitifs.

### 1.5.1 Le partitionnement du MP-RSoC

Deux aspects à prendre en compte lors de la mise en œuvre d'une application sur un MP-RSoC : le partitionnement spatio temporel identifie les segments de l'application qui peuvent s'exécuter indépendamment les uns des autres sur une tranche de temps ; le partitionnement architectural définit les tâches réalisables logiciellement et celles réalisables matériellement. Plus précisément Il s'agit d'une part, lors de la conception, de décider si des tâches doivent être réalisées à l'aide de processeurs ou si des tâches doivent être directement implantées matériellement. Il s'agit d'autre part, lors de l'exécution, de décider quand une tâche doit être chargée en mémoire ou configurée. Ce partitionnement dans le cadre d'un MP-RSoC ne peut être intégralement décidé avant l'exécution. Pour exploiter au mieux les caractéristiques dynamiques d'une application, le partitionnement doit avoir lieu durant l'exécution. Il est, dans ce cadre, intéressant de disposer d'un intergiciel qui fournit ces services à l'instar d'un OS comme dans [21] ou bien de disposer d'une bibliothèque de primitives qui donne à chaque tâche de l'application les primitives permettant de gérer ses communications et le chargement dynamique des configurations applicatives pendant le fonctionnement du MP-RSoC.

### 1.5.2 Les modules du MP-RSoC

Le MP-RSoC peut être physiquement modélisé comme un ensemble de modules qui échangent des données pour réaliser une application. Le module est à minima l'unité d'exécution qui accomplit au moins une fonction constitutive d'une tâche de l'application ou à maxima une unité de traitement complexe responsable de plusieurs tâches tel qu'un processeur. Sur la puce, le concepteur de MP-RSoC dispose des modules physiques et des modules virtuels encore appelés blocs IP. Pour faire fonctionner les modules ensemble, il leur associe des interfaces d'interconnexion pour assurer les communications. Différents standards ont

été fournis pour l'interfaçage. Nous pouvons citer : VCI (Virtual component interface) développé par VSIA [22] , OCP (Open Core Protocol) développé par OCP-IP (Open Core Protocol-International Partnership) et IP-XACT développé par le consortium SPIRIT [23]. L'utilisation de ces standards spécifiques augmente la complexité de la conception des MP-RSoC et rend difficile la réutilisation des IP. Dans ce cas de figure, il est nécessaire, lors de la conception, de prendre en compte cet interfaçage et de disposer d'outils qui soient capables de gérer la diversité des standards, ou bien qui imposent une norme commune, pour réaliser un MP-RSoC hétérogène (voir figure 1.2)

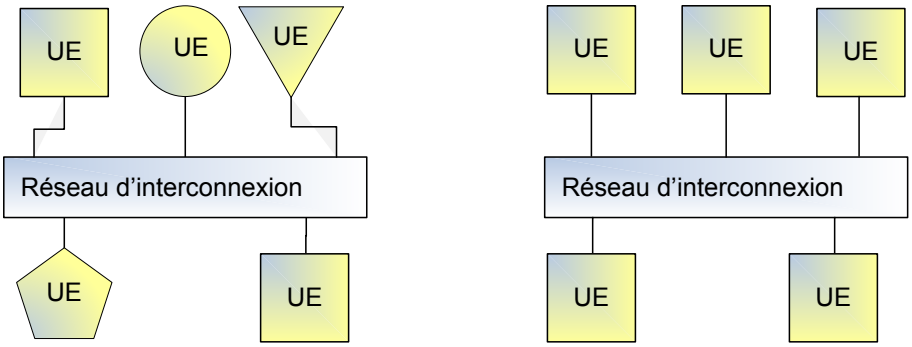



FIGURE 1.2 : Système hétérogème et système homogène [24]

### 1.5.3 Partition statique et partition dynamique du MP-RSoC

Certains modules du MP-RSoC seront déployés en permanence sur la puce, ils constitueront la partition statique du MP-RSoC tandis que d'autres modules seront déployés dynamiquement. Les modules dynamiques sont configurés sur les ressources logiques de l'ARD suivant l'évolution de l'application. La figure 1.3 présente le principe d'organisation d'un MP-RSoC utilisant des régions reconfigurables réservées pour accueillir des modules dynamiques. Ces régions échangent des informations avec la partition statique du MP-RSoC à l'aide des connecteurs internes, les Bus macro de Xilinx [25] par exemple.

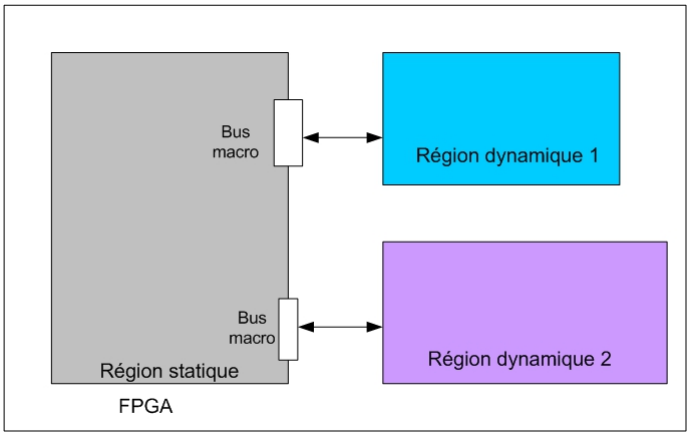



FIGURE 1.3 : Architecture de base d'un système reconfigurable dynamiquement

## 1.6 Le MP-RSoC : un système parallèle

Les opérations logiques sont naturellement concurrentes dans les puces ayant des ARD, ce qui permet de facto de réaliser des systèmes de traitements parallèles. Le MP-RSoC doit être structuré suivant un modèle de système parallèle afin de faciliter le travail du concepteur de l'application. Sans un modèle de structuration de l'application parallèle, le concepteur serait obligé de synchroniser les modules de traitement par un système adhoc alors que l'utilisation d'un modèle donne la possibilité de réaliser des mécanismes de synchronisation standardisés ou mieux, d'utiliser un mécanisme de synchronisation déjà disponible afin de réaliser plus rapidement son application. Nous définissons dans cette section la notion de tâche matérielle et rappelons les architectures des systèmes parallèles.

### 1.6.1 Tâches matérielles et tâches logicielles

**Tâche logicielle** : désigne une instance exécutable sur un processeur d'un algorithme, d'un processus, d'un traitement, ou d'une fonctionnalité quelconque. Une tâche logicielle peut utiliser d'autres primitives ou fonctions (sous-tâches) pour son exécution, elle peut ainsi requérir plusieurs unités d'exécution.

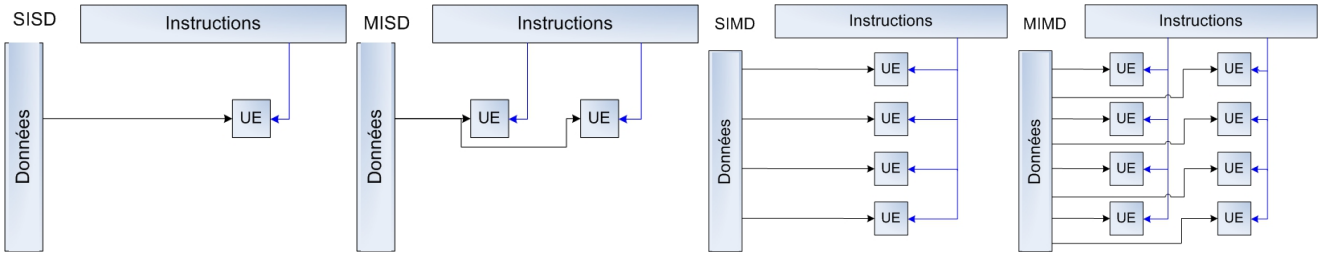
**Tâche matérielle** : désigne la réalisation directe sur les ressources logiques d'une puce, d'un algorithme, d'un processus, d'un traitement ou d'une fonctionnalité quelconque à partir d'un code écrit en langage de description matérielle. Une tâche matérielle peut être décrite en assemblant des IP ou en utilisant des modules IP dans la description.

### 1.6.2 Les architectures de traitements parallèles

Suivant la classification de Flynn [26], une architecture parallèle peut être de type SIMD, MISD ou MIMD (voir figure 1.4). Chaque type décrit l'interaction entre le flux de données, le flux d'instructions et l'unité d'exécution. Dans le type SIMD, il s'agit d'une architecture où les unités d'exécution traitent la même instruction sur des données différentes. Les processeurs vectoriels, les processeurs de traitement graphique sont des exemples d'architectures SIMD.

Dans le type MISD, les différentes unités d'exécution traitent en parallèle le même jeu de données par des instructions potentiellement différentes, les architectures pipeline appartiennent à ce type. Dans le type MIMD, les unités d'exécution traitent différents ensembles de données à l'aide de différents ensembles d'instructions ; c'est le cas des systèmes multiprocesseurs. Le type SISD de la figure 1.4 désigne une architecture mono traitement : l'unité d'exécution réalise un traitement sur une donnée à l'aide d'une instruction.

La structure MIMD est celle qui permet la mise en œuvre du parallélisme des tâches dans un MP-RSoC, chaque tâche matérielle peut indépendamment traiter un ensemble particulier de données.




(a) SISD (b) MISD (c) SIMD (d) MIMD

FIGURE 1.4 : Taxonomie de Flynn

### 1.6.3 Modèles mémoire de l'application parallèle

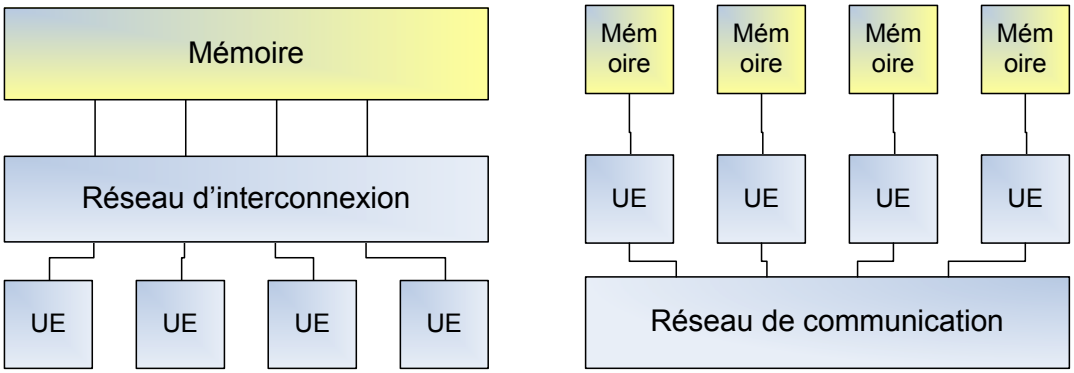
Dans la littérature, on distingue suivant la répartition du stockage des données deux modèles de systèmes parallèles : les systèmes parallèles à mémoire partagée (voir figure 1.5a), et les systèmes parallèles à mémoire distribuée (voir figure 1.5b). Chaque modèle influence directement la conception matérielle ainsi que le modèle de déploiement d'application du MP-RSoC. La figure 1.5 montre une architecture à mémoire partagée et une architecture à mémoire distribuée.

#### 1.6.3.1 Systèmes parallèles à mémoire partagée

Dans les systèmes à mémoire partagée, toutes les unités d'exécution (UE) partagent les ressources de la même mémoire ; par conséquent, tous les changements faits par une UE à un emplacement mémoire deviennent visibles à toutes les autres UE. D'un point de vue conceptuel, les systèmes à mémoire partagée sont peu évolutifs. Ceci est dû au fait que la bande passante d'accès à la mémoire du système s'amenuise avec la croissance du nombre d'UE. Dans un système à mémoire partagée, les différents processus échangent des informations à travers les variables partagées. Les systèmes à mémoire partagée exigent des mécanismes de synchronisation et de protection des variables partagées en mémoire tels que les sémaphores, les barrières et les serrures. POSIX threads [27] et OpenMP [28] permettent de mettre en œuvre une application parallèle dans un système multiprocesseurs à mémoire partagée.

#### 1.6.3.2 Systèmes parallèles à mémoire distribuée

Dans les systèmes à mémoire distribuée, chaque unité d'exécution a sa mémoire privée (voir figure 1.5b). Les UE communiquent de manière explicite et utilisent un protocole de communication particulier. Les systèmes à mémoires distribuées sont potentiellement plus évolutifs, car les supports de communication sont partagés entre les unités d'exécution [24]. Les systèmes à mémoire distribuée exigent des mécanismes pour supporter les communications explicites entre les instances des tâches. Habituellement une bibliothèque de primitives est utilisée pour permettre d'écrire sur les canaux de communications. La bibliothèque MPI




(a) mémoire partagée (b) mémoire distribuée

FIGURE 1.5 : Système à mémoire partagée et système à mémoire distribuée [24]

(Message Passing Interface) est le standard le plus utilisé [29]. Le principal inconvénient de l'architecture à mémoire distribuée est qu'elle nécessite un protocole de communication et mobilise plus de ressources pour sa réalisation que le partage de la mémoire entre les UE. Une application parallèle dans un système multiprocesseurs à mémoire distribuée est plus complexe à mettre en œuvre que dans un système multiprocesseurs à mémoire partagée du fait de la spécification explicite du parallélisme.

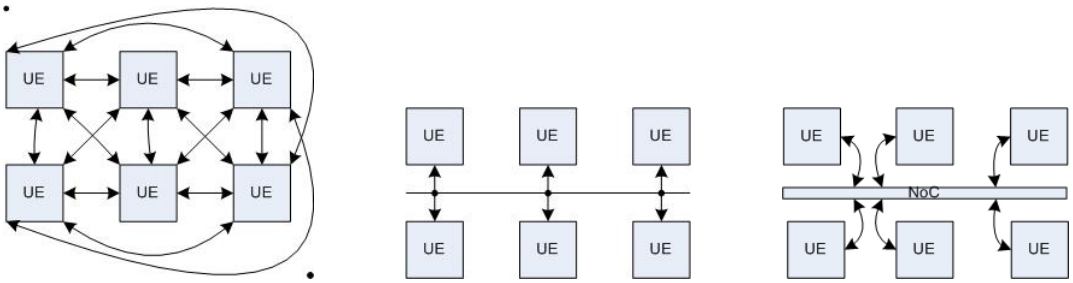
### 1.6.4 L'interconnexion des systèmes parallèles

Pour la mise en œuvre d'un MP-RSoC, il est nécessaire de choisir une topologie physique des connexions qui permette aux UE de communiquer dans la puce. A ce niveau on distingue trois grandes orientations.

i Point à point (représenté en figure 1.6a) : les noeuds possèdent entre eux des liaisons directes ce qui permet d'avoir une grande bande passante car le support de communication n'est pas partagé. Néanmoins il n'est pas facile de faire évoluer le système à cause du coût silicium induit.

ii Bus partagé (représenté en figure 1.6b) : c'est une approche traditionnelle qui dérive des systèmes mono-processeur. Cette approche est assez répandue pour la communication entre les UE ; mais elle induit une latence qui pénalise les performances car le bus ne peut être contrôlé que par une UE à la fois.

iii L'approche réseau (représenté en figure 1.6c) : elle pallie au manquement des approches précédentes. Son principe est d'interconnecter les UE à travers un réseau géré par arbitrage. Les méthodes d'arbitrage sont inspirées des méthodes d'arbitrage du trafic dans les réseaux informatiques. Cette approche offre le meilleur compromis encombrement et performance lorsque le nombre d'UE devient important.




(a) Point à point (b) Bus (c) NoC

FIGURE 1.6 : interconnexion des systèmes parallèles

### 1.6.5 Application parallèle et MP-RSoC

Nous avons montré qu'un MP-RSoC est un système parallèle pour lequel des choix de modèles d'organisations sont à faire. Un autre choix à faire assez tôt dans la phase de conception est celui de l'architecture qui permettra de déployer la reconfiguration dynamique du matériel. Le choix des outils pour concevoir et réaliser le MP-RSoC est lié à l'ARD retenue. Nous retenons que ces outils doivent gérer deux classes de problèmes : la gestion d'une application parallèle dans un système sur puce et la reconfiguration dynamique de l'ARD. Pour appliquer le flot de reconfiguration dynamique à une plateforme MP-RSoC, il faut très bien comprendre et maîtriser la technologie et la structure interne de ce MP-RSoC, compréhension et maîtrise à acquérir à chaque évolution technologique. Les avancées technologiques créent une certaine instabilité en rendant assez vite obsolètes certaines plateformes MP-RSoC et poussant le concepteur à réviser sa conception. La gestion de l'application parallèle va se faire généralement à l'aide de composants logiciels ou matériels additionnels qui mettent en œuvre une interconnexion et un protocole de communication entre les modules fonctionnels.

## 1.7 Concevoir et utiliser les MP-RSoC

Compte tenu de ce qui précède, une question s'impose : comment concevoir et déployer une application parallèle sur un MP-RSoC en minimisant le temps de développement et en maximisant l'efficacité du MP-RSoC ?

Aujourd'hui pour la conception, deux approches existent (voir sur la figure 1.7). L'une dite *"Top down"* suppose une exploration architecturale complète pour aboutir à la réalisation de l'application sur une cible et l'autre dite *"Meet in the middle"* propose de raffiner l'application pour définir ses besoins architecturaux puis d'utiliser une plate-forme prédéfinie de MP-RSoC pour mapper l'application (voir figure 1.8). L'utilisation d'une plate-forme MP-RSoC a pour objectif de réduire le temps de conception de l'application parallèle. La flexibilité de la plate-forme MP-RSoC favorise la réussite de la transformation des traitements du système initial en une application parallèle.

La méthode basée sur une plate-forme permet de simplifier le déploiement d'une appli-

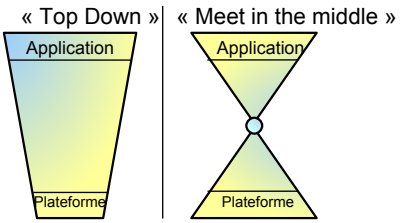



FIGURE 1.7 : Méthodes de raffinement de la conception

cation en utilisant un modèle "*Meet in the Middle*" où la plate-forme est réalisée par une équipe distincte de l'équipe qui réalise l'application.

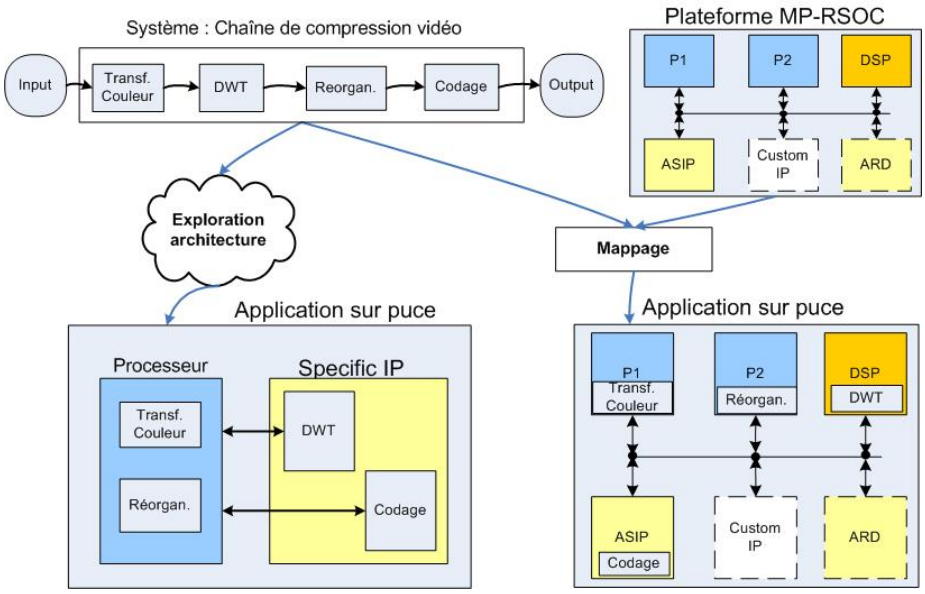



FIGURE 1.8 : Exemple de déploiement d'une application de décompression vidéo sur un MP-RSoC

## 1.8 Conclusion

Nous venons de définir la problématique de ce travail de thèse : la conception et le déploiement d'application parallèles sur des MP-RSoC. Le MP-RSoC répond à des besoins que nous avons identifiés, parmi lesquels : traiter un grand volume de données en un temps minimum, consommer moins d'énergie, être modulaire, être dynamique, être hétérogène, supporter la montée en charge. Nous avons rappelé que les performances et la flexibilité de l'architecture matérielle de la plateforme dépendent du réseau d'interconnexion des UE et du paradigme de parallélisation utilisé. Les outils, les flots de conception et de déploiement pour mettre en œuvre des applications parallèles sur des plateformes reconfigurables dynamiquement représentent aujourd'hui un verrou limitant leur utilisation. Le prochain chapitre de ce travail est consacré à l'état de l'art ; il répond aux questions : Quels sont les plateformes MP-RSoC existantes et quels sont les outils mis à la disposition des concepteurs pour le déploiement de leurs applications ?

# Chapitre 2

# État de l'art

## Sommaire

## 2.1 Introduction

La conception d'une application à déployer sur un MP-RSoC suit une approche descendante du modèle fonctionnel au modèle RTL [1] et parcourt les phases suivantes :

- la phase de collecte des besoins : rédiger un cahier de charges, exprimer un ensemble de besoins ou de fonctionnalités (généralement par le client ou un utilisateur) ;
- la phase conceptuelle : identifier les traitements principaux et faire les choix méthodologiques et technologiques, établir les graphes conceptuels, le plan de conception, définir les modules fonctionnels, faire le choix du matériel, ébaucher la stratégie de déploiement de l'application ;
- la phase de développement : réaliser les modules fonctionnels et modéliser les différentes transactions et interactions ;
- la phase de validation conceptuelle : évaluer la conception suivant des critères standardisés pour garantir la fonctionnalité de l'application finale ;
- la phase de simulation : vérifier par simulation fonctionnelle les modules individuellement puis l'ensemble de l'application ;
- la phase d'exploration : rechercher l'architecture de la plateforme d'exécution de chaque module fonctionnel de l'application ;
- la phase de déploiement : mettre en œuvre les modules de l'application sur la plateforme matérielle d'exécution ;
- la phase d'exploitation : transférer la plateforme dans son environnement d'exécution et la soumettre aux traitements en situation réelle ;
- la phase de maintenance : faire des ajustements et des corrections dans la plateforme pour corriger les fonctionnalités défaillantes ou améliorer les performances.

Ces différentes phases de conception sont réalisées à l'aide d'un ensemble d'outils permettant de modéliser les différents niveaux d'abstraction d'une application. Il est possible de modéliser séparément les modules fonctionnels ou unités de traitements et les interconnexions entre ces modules fonctionnels.

Quelques outils de modélisation sont capables de générer automatiquement le code exécutable à partir du modèle fonctionnel. Cependant, la génération de code aujourd'hui est inefficace voir impossible lorsqu'elle est demandée pour un système très complexe ou très hé-

1. Register Transfer Logic

térogène. Une solution consiste à modéliser puis réaliser chaque partie du système avec l'outil le mieux adapté et à assembler toutes les parties du système avec des outils de conception de plus bas niveau.

Différents outils de conception permettent la réalisation d'un MP-RSoC à partir d'une spécification abstraite de haut niveau ou à partir d'une spécification concrète de bas niveau de chaque module fonctionnel de l'application.

Des outils de haut niveau parmi lesquels Matlab Simulink pour la conception des modules fonctionnels dédiés au traitement de signal comme dans [30] et [31]. Xilinx System Gererator associé à Matlab Simulink, automatise la génération du code RTL et transforme des modèles abstraits en des modèles fonctionnels exécutables sur FPGA Xilinx [32].

Des outils de niveau intermédiaire comme SystemC qui est une extension du langage C++, plus précisément une bibliothèque de classes C++ contenant des modèles matériels, ainsi que toutes les briques de base pour modéliser un système matériel. SystemC a pour objectif de simuler des systèmes numériques matériels et logiciels à l'aide de C++ [33].

Les outils de bas niveau comme les langages HDL (Verilog, VHDL) sont majoritairement utilisés pour décrire le modèle RTL d'un MP-RSoC.

Pour utiliser un MP-RSoC, il est nécessaire de disposer d'outils qui permettent de déployer des applications avec facilité, afin de profiter des avantages de la gestion spatio-temporelle qu'offre la technologie de la reconfiguration dynamique [34] [35].

Nous présentons dans ce chapitre un état de l'art de l'offre disponible afin de souligner les besoins qui ne sont pas encore couverts par ces solutions.

Dans la prochaine section nous présentons les possibilités et les limites du langage VHDL ; dans la section 3 nous introduisons les outils pour MP-RSoC ; dans les sections 4 à 7 nous présentons les outils et plateformes de déploiement de MP-RSoC suivant leur niveau d'abstraction ; la section 8 est une conclusion.

## 2.2 VHDL et MP-RSoC : le langage et ses limites

Le langage VHDL [2] est un langage de description de systèmes électroniques numériques. Ce langage est né du programme VHSIC [3] initié par le gouvernement des États-Unis. Dans le cadre de ce programme, il est devenu clair qu'il existait un besoin d'un langage standard pour décrire la structure et la fonction des circuits intégrés. VHDL a ensuite été développé sous les auspices de l'*Institute of Electrical and Electronics Engineers* (IEEE) et adopté en 1987 sous la forme de la norme IEEE 1076. Comme toutes les normes IEEE, la norme VHDL est soumise à des évolutions. Les commentaires et suggestions des utilisateurs de la norme 1987 ont été analysés par le groupe de travail IEEE responsable de VHDL, et en 1992 une version révisée de la norme a été proposée. Elle fut finalement adoptée en 1993, introduisant

2. VHSIC Hardware Description Language
3. Very High Speed Integrated Circuits

VHDL-93. Une deuxième étape de la révision de la norme a été lancée en 1998. Ce processus s'est achevé en 2001, conclu par VHDL-2002. Après cela, le développement a eu lieu dans le groupe de travail IEEE et d'un comité technique d'une organisation, Accellera, dont la charte est de promouvoir des normes pour la conception électronique. Ces efforts ont abouti à la version actuelle du langage : VHDL-2008 [36].

### 2.2.1 Les objectifs du VHDL

VHDL est conçu pour satisfaire un certain nombre de besoins dans le processus de conception. Ce langage permet :

- de décrire la structure d'un système, en sous-systèmes et la façon dont ces sous-systèmes sont reliés entre eux ;
- la spécification de la fonction d'un système à l'aide des formes de langage de programmation familier ;
- de simuler un système à concevoir avant de le fabriquer, de sorte que les concepteurs peuvent rapidement comparer les solutions de rechange et de test pour l'exactitude sans retard et frais de prototypage du matériel ;
- de synthétiser la structure détaillée d'une conception à partir d'une spécification plus abstraite, permettant aux concepteurs de réduire les délais de fabrication.

### 2.2.2 La dynamicité en VHDL

Nous pouvons classer les modèles des systèmes sur puce suivant trois domaines : le fonctionnel, le structurel et le géométrique.

Le domaine fonctionnel concerne les opérations effectuées par le système. Dans un sens, c'est le domaine le plus abstrait de description, car il ne précise pas comment la fonction est mise en œuvre.

Le domaine structurel traite de la manière dont le système est composé de sous-systèmes interconnectés. Le domaine géométrique traite de la manière dont le système est disposé dans l'espace physique, ici la surface par exemple d'un FPGA ou d'un ASIC. Le VHDL permet de décrire des modèles fonctionnels et structurels.

Le langage VHDL comprend des instructions pour modéliser la structure et la fonction d'un système du niveau le plus abstrait jusqu'au niveau de la porte. Il fournit également un mécanisme d'attributs qui peut être utilisé pour annoter un modèle fonctionnel ou structurel avec des informations permettant de construire le modèle géométrique du système.

Le VHDL est tout d'abord un langage de modélisation, de simulation et de spécification. Nous pouvons également l'utiliser pour la synthèse de matériel si nous utilisons uniquement les instructions du langage réservées pour décrire les modèles RTL [37].

Toutefois, aucun mécanisme standard du langage VHDL ne permet en 2014 de décrire une

architecture dynamique [37]. Le VASG [4] œuvre pour donner plus de dynamicité au langage depuis la version *VHDL IEEE 1076 : 2008*, mais les nouvelles extensions proposées par [38] et [39] ne sont pas mises en œuvre à ce jour, soit parce que les constructions à adopter ne font pas encore l'unanimité soit à cause de l'immaturité des outils pour la reconfiguration dynamique des MP-RSoC.

Le VHDL ne suffit pas à lui seul pour concevoir et déployer une application sur un MP-RSoC. Il est nécessaire de définir des méthodes et outils pour la reconfiguration dynamique. Nous en présentons certains dans la section suivante.

## 2.3 Outils de modélisation et niveau d'abstraction du MP-RSoC

Il existe actuellement des outils et des plateformes MP-RSoC qui peuvent être classés suivant leur niveau d'abstraction comme sur la figure 2.1. Le niveau MDE/MDA [5] (Ingénierie dirigée par les modèles) est le plus haut niveau. Il permet de décrire une application par des objets abstraits et vise à la transformation automatique de cette description en une application exécutable. Le niveau PBD [6] favorise l'utilisation des abstractions appropriées, améliorant la compréhension d'un système. Il permet de renforcer la probabilité de réussite d'une mise en œuvre efficiente de la fonctionnalité du système [40]. Le niveau algorithmique utilise des langages informatiques pour modéliser l'application et décrire tous les modules fonctionnels. Le niveau RTL modélise au cycle près et au bit près chaque fonctionnalité de l'application à l'aide de primitives matérielles ou de langage HDL. Des recherches se sont focalisées sur la conception d'un outil pour faciliter la reconfiguration tandis que d'autres ont proposé des plateformes complètes.

Nous nous intéressons dans ces travaux au déploiement d'applications parallèles sur un MP-RSoC. Certains projets de recherche se sont intéressés à la mise en œuvre de la reconfiguration dynamique partielle dans les FPGA. Ces projets ont pour originalité de proposer un concept et une mise en œuvre plus performants que ce que les fabricants, Xilinx majoritairement, proposent en standard. Nous allons en présenter quelques-uns dans les paragraphes qui suivent afin d'illustrer cette tendance.

## 2.4 Outils niveau RTL

Pour déployer des applications sur un MP-RSOC, les outils les plus connus sont les outils de niveau RTL comme Xilinx ISE et Altera Quartus II. Ces outils proposent de modéliser

4. VHDL Analysis and Standardization Group
5. Model Driven Engineering/Model driven Architecture
6. Plateform Based Design : conception orientée plateforme

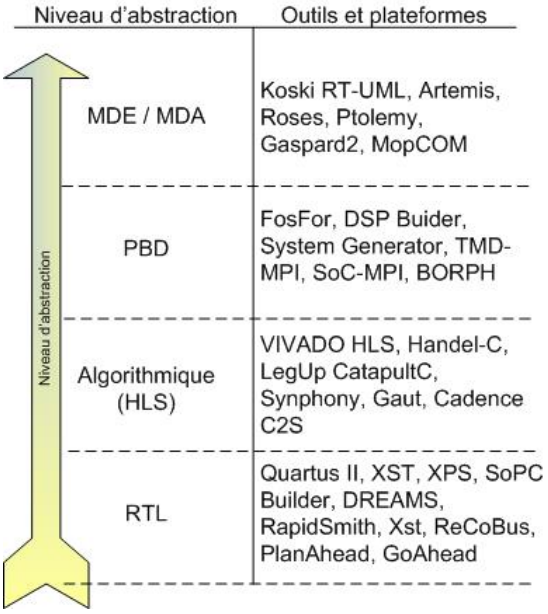



FIGURE 2.1 : Outils de conception des MP-RSoC selon leur niveau d'abstraction

entièrement le MP-RSoC et l'application qui y sera déployée à partir d'un langage HDL (Verilog/VHDL). Pour accélérer la conception d'un système complexe les fabricants proposent une deuxième catégorie d'outils qui permettent le déploiement à l'aide d'une interface graphique et d'une bibliothèque de processeurs softcores, de bus système et d'IP. Chez Xilinx l'outil se nomme Xilinx Platform Studio (XPS), et chez Altera il se nomme SoPC Builder. Ces outils ne sont pas réellement adaptés pour réaliser des applications dynamiques et massivement parallèles car ils utilisent un modèle plutôt statique organisé autour d'un processeur au cœur du MP-RSoC, d'accélérateurs matériels souvent figés et d'un bus hiérarchique d'interconnexion. D'autres outils, utilisant ou non ces solutions propriétaires, ont été développés pour dépasser leurs limites.

### 2.4.1 Dreams, RapidSmith

Pour une meilleure efficacité de la reconfiguration dynamique, il est intéressant de reloger un *bitstream* sur une autre région du FPGA. Ceci a pour avantage, entre autre, de réduire la taille de la mémoire occupée par les *bitstream*. L'outil de déploiement PlanAhead, du fabricant de FPGA Xilinx, ne permet pas de créer des bitstreams relogeables. Le projet DREAMS [41] qui utilise le framework RapidSMITH [42] est un exemple d'outil qui permet sous certaines contraintes de reloger un *bitstream* partiel dans un FPGA de Xilinx.

### 2.4.2 cPCAP

Le projet cPCAP (compressed Parallel Configuration Access Port) met en œuvre un softcore qui peut reconfigurer le Xilinx Spartan 3 à travers le port Select Map car ce dernier ne possède aucun port interne pour l'auto reconfiguration (voir figure 2.2). Le bitstream

partiel est généré à l'aide de la commande `bitgen -r` de l'outil de Xilinx puis converti en un fichier de coefficients (.coe)[7], compressé et stocké dans les blocs RAM internes du FPGA. D'après Bayar S. et Yurdakul A. [43], le softcore cPCAP peut être utilisé avec les FPGA Virtex-II, Virtex-4, Virtex-5.

L'inconvénient avec cPCAP est que la taille du bitstream partiel compressé doit être logeable dans les blocs RAM du FPGA à reconfigurer partiellement ce qui limite la complexité des systèmes qui peuvent être stockés sous forme de bitstream dans les blocs RAM du FPGA.

Le sotfcore cPCAP est limité à la fonction de déploiement des modules reconfigurables dans un FPGA, il n'est pas un outil de conception et de déploiement d'application parallèle pour MP-RSoC.

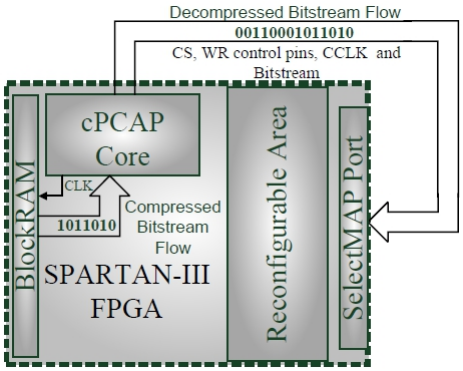



FIGURE 2.2 : Architecture de la plateforme cPCAP[43]

### 2.4.3 RecoBus builder

Le projet ReCoBus [44] propose une architecture de communication optimisée pour les FPGA Xilinx membres des familles Spartan-3, Virtex-II et Virtex-Pro, qui permet de créer un bus de connexion et jusqu'à 246 *slots*[8] dans le FPGA VIRTEX II pour accueillir des modules fonctionnels reconfigurables. Le bus de connexion est mis en œuvre directement dans la matrice de routage du FPGA et chaque slot coûte 7 *slices*[9] pour la connexion au bus. Les modules fonctionnels peuvent être déployés sur une tranche du FPGA d'une taille de 66x4 *slices*. Il est possible pour un module d'occuper plusieurs tranches contiguës. Les modules reconfigurables peuvent être connectés dynamiquement sur le bus ; ils communiquent à l'aide d'un protocole défini. Les modules peuvent être déplacés d'un slot vers un autre. ReCobus est très lié à l'architecture des FPGA pour lesquels il est conçu car il suppose une certaine homogénéité dans la répartition des ressources logiques du FPGA et il utilise les bus macro de Xilinx. ReCoBus favorise la mise en œuvre des applications reconfigurables dont les modules fonctionnels sont organisés suivant une architecture maître-esclaves.

7. format standard des fichiers d'initialisation des blocs RAM pris en charge par les outils de Xilinx au moment de la configuration initiale du FPGA
8. des ports de connexion
9. tranche logique contenant deux CLB

Sur la figure 2.3 nous voyons le flot de conception pour l'architecture ReCoBus. L'outil ReCoBus builder assiste le concepteur pour le placement et le routage de l'application basée sur ReCoBus. Le projet RecoBus builder [44] permet de reconfigurer dynamiquement le FPGA en :

— relocalisant le module dynamique,
— configurant différentes régions du FPGA avec le même bitstream partiel,
— partageant la même région reconfigurable avec plusieurs modules différents.

. Les principales limitations du projet ReCoBus :

— il impose au concepteur une architecture de type maître-esclaves pour le MP-RSoC, ceci est une restriction pour la communication entre les tâches matérielles et les communications ne peuvent se faire que par l'intermédiaire d'une tâche maître dans la zone statique ;
— pour interfacer les modules reconfigurables avec le bus ReCobus, des tuiles précises sont à configurer ;
— l'application de configuration du bus ReCoBus builder utilise des macros qui sont mises en œuvre à partir de LUT et qui ne sont pas supportées par toutes les versions des outils de synthèse de Xilinx ;
— il n'offre pas au concepteur un modèle de communication standard entre tâches matérielles, mais il utilise un protocole dédié.

### 2.4.4 GoAhead

Dans [46] Beckhoff et al. présentent le projet GoAhead qui définit une méthodologie et propose un environnement logiciel pour déployer une application ayant des modules reconfigurables. L'outil possède une interface graphique qui facilite la définition des régions reconfigurables du FPGA et qui appelle les commandes de Xilinx de façon transparente pour créer les bitstreams partiels et vérifier les contraintes temporelles de l'application. L'interconnexion entre les régions reconfigurables et la région statique est assurée à travers des bus macros. Il est possible d'utiliser le même bitstream pour reconfigurer plusieurs régions distinctes avec GoAhead. La figure 2.4 présente le flot de déploiement de l'application avec GoAhead. L'outil propose un template Vhdl qui permet de configurer, à l'aide du port ICAP [10], le Spartan 6 avec des données reçues depuis une interface série de type RS232C. Le concepteur fournit les fichiers RTL de la partie statique et du module partiel puis l'outil génère les contraintes pour le routage, et produit les bitstreams partiels qui permettront de configurer le FPGA. Le flot de GoAhead permet de déployer dynamiquement des modules reconfigurables sur un FPGA Xilinx Spartan 6.

Comme le projet ReCoBus, l'outil GoAhead permet au concepteur d'intégrer des modules reconfigurables dans un FPGA Xilinx d'entrée de gamme mais ne propose pas une stratégie

10. Internat configuration Access Port

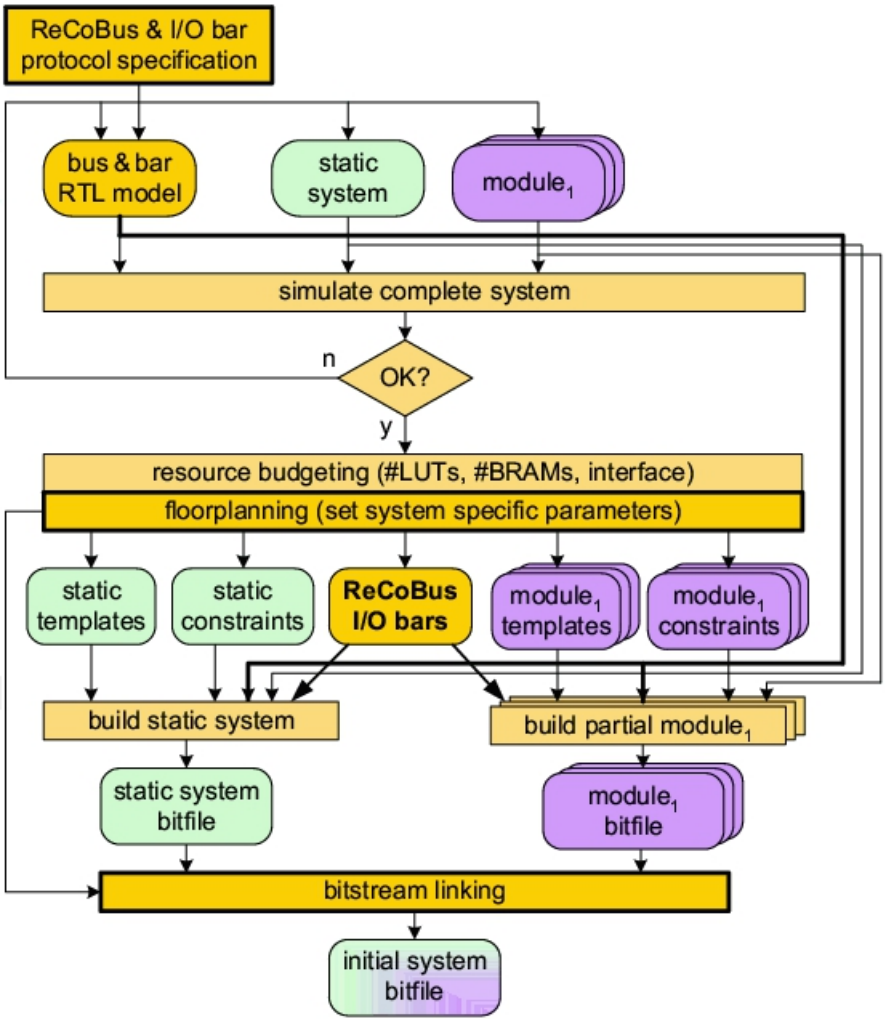



FIGURE 2.3 : Flot de conception à l'aide de l'outil RecoBus Builder [45]

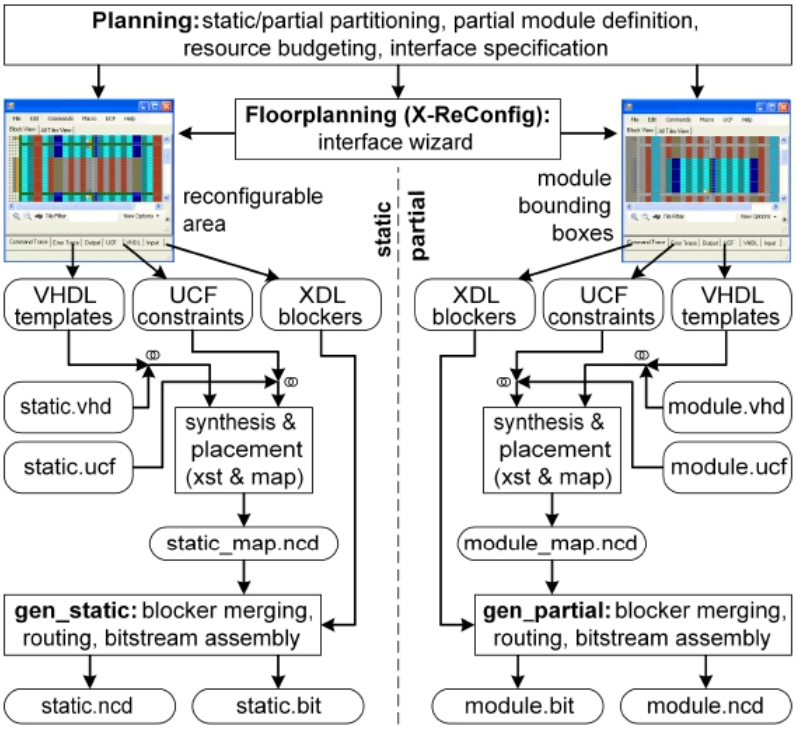



FIGURE 2.4 : Flot de déploiement d'une application ayant un module reconfigurable avec GoAhead sur FPGA Xilinx Spartan 6 [46]

ou une interface standard pour gérer les communications entre les différents modules du systèmes sur puce. Or lorsque le nombre de modules reconfigurables augmente, il est difficile de les faire communiquer sans disposer d'une infrastructure d'interconnexion et d'une méthode standard de communication entre les modules. Avec GoAhead, le développement de cette infrastructure est laissé à la charge du concepteur.

### 2.4.5 Plateforme FPX et l'outil PARBIT

Dans [47] Horta et al. présentent la plateforme `Field programmable Port Extender` qui est réalisée à l'aide de deux FPGA XILINX VIRTEX E : le NID (Network Interface Device) qui est un XCV600E [11] et le RAD (Reprogrammable Application Device) qui est un XCV2000E [12]). L'outil PARBIT [48] a été développé pour transformer et restructurer le bitstream créé par les outils de synthèse standard en des bitstreams partiels permettant de configurer une portion du FPGA. Cet outil a été utilisé sur la plateforme FPX (voir figure 2.5) pour déployer des modules reconfigurables (DHP [13]) sans perturber le reste du système. Avec cette plateforme il est possible de déplacer un module DHP d'une région reconfigurable initiale vers une autre ayant les mêmes ressources logiques.

PARBIT vient enrichir l'offre des outils permettant de gérer la reconfiguration dynamique

11. Circuit de 1999 en technologie 180 nm ayant 15552 cellules logiques
12. circuit de 1999 en technologie 180 nm ayant 43200 cellules logiques
13. Dynamic Hardware Plug in

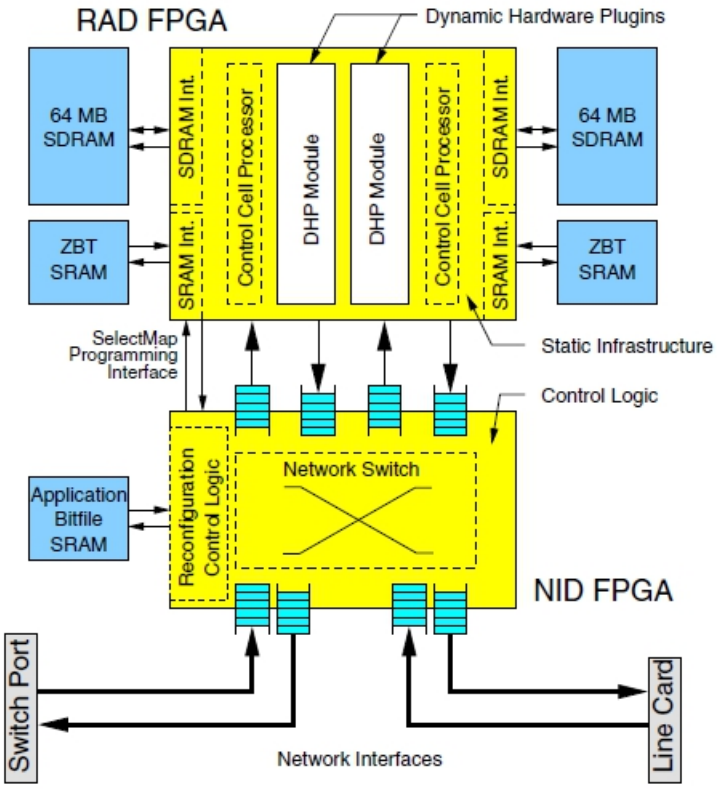



FIGURE 2.5 : La plateforme FPX [47]

de certains FPGA de Xilinx. La plateforme PFX nous présente une approche de déploiement multi FPGA du système sur puce.

Toutefois, comme nous l'avons dit précédemment, cette seule opération ne suffit pas à faciliter la conception du MP-RSoC.

D'autres outils, de plus haut niveau, existent qui facilitent notamment la conception des modules matériels à partir de langage informatique ; ces outils sont présentés dans la section suivante.

## 2.5 Outils de niveau Algorithmique (HLS)

Cette catégorie d'outils réalise la synthèse d'un système à partir d'une description algorithmique en utilisant un langage de type C/C++.

### 2.5.1 LegUP

L'outil de synthèse algorithmique LegUp [49] accepte un programme C standard en entrée et compile le programme pour une architecture hybride, réalisée sur cible FPGA, contenant un processeur softcore MIPS 32 bits et des accélérateurs matériels personnalisés qui communiquent via une interface de bus standard Avalon [14] [50]. Dans l'architecture hybride

14. Bus propriétaire du fabricant ALTERA pour réaliser l'interconnexion des composants

processeur / accélérateurs, les segments de programme qui sont impropres à la mise en œuvre matérielle peuvent s'exécuter sous forme logicielle sur le processeur. LegUp peut synthétiser la plupart du langage C au matériel, y compris les tableaux multi-dimensionnels de taille fixe, les structures, les variables globales et l'arithmétique des pointeurs. La figure 2.6 montre le flot de conception avec l'outil LegUp. Un point important de cet outil est qu'il est fourni sous forme *Open Source*, ainsi les concepteurs et les chercheurs peuvent l'adapter pour leurs besoins particuliers.

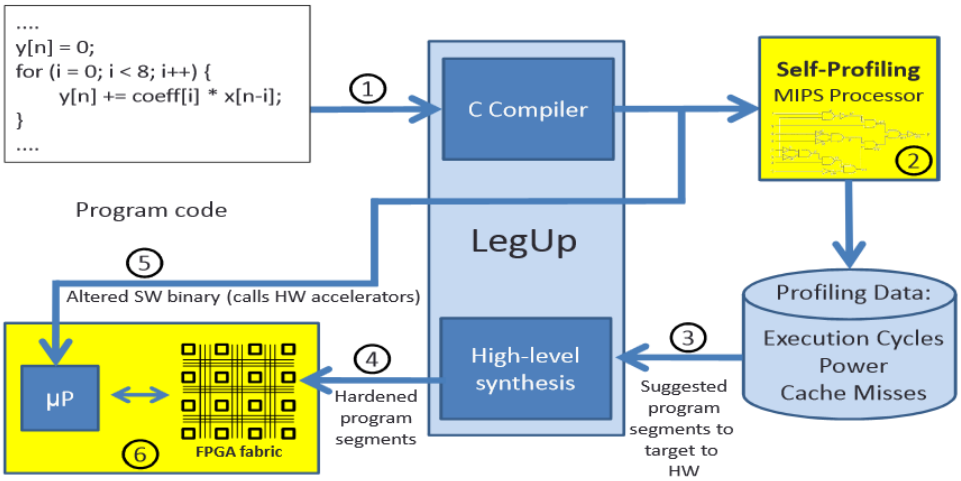



FIGURE 2.6 : Flot standard de conception LegUp [49]

### 2.5.2 Synphony HLS

Synphony HLS est une suite d'outils proposés par la société Synopsys. Ces outils permettent de créer rapidement des algorithmes complexes qui sont synthétisables dans les FPGA optimisés ou des mises en œuvre ASIC. *Synphony C compiler* prend en charge la synthèse de haut niveau à partir des langages C/C ++, et *Synphony Model Compiler* fournit la synthèse de haut niveau à partir de l'environnement de modélisation Matlab-Simulink qui est fondé sur les modèles graphiques appelés *blockset* pour représenter et simuler des systèmes (Voir la figure 2.7).

### 2.5.3 CatapultC

Cet outil HLS commercialisé dès 2004 par Mentor Graphics Inc. a été racheté depuis août 2011 par Calypto Design System. CatapultC est capable de synthétiser le langage ANSI C/C++ sans extensions propriétaires. Le support du langage C/C++ inclut des pointeurs, des classes, des templates et la surcharge d'opérateur.

Catapult C prend en charge la génération de modèle SystemC destiné aux plateformes virtuelles et un environnement de vérification SystemC pour comparer le RTL généré au

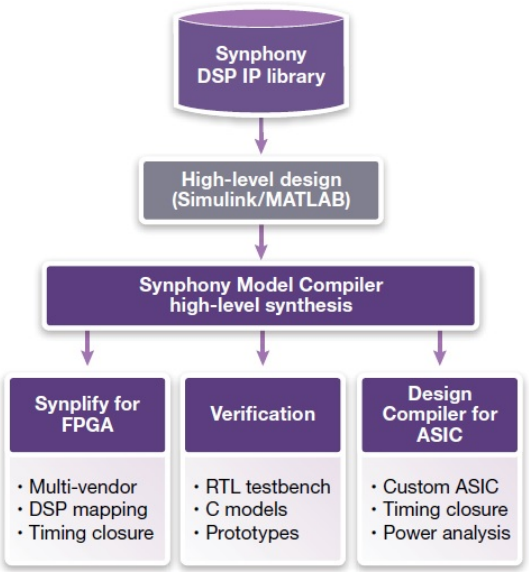



FIGURE 2.7 : Flot de conception avec Synphony Model Compiler [51]

code C ++ d'origine en utilisant le banc d'essai C ++ d'origine. Sa spécificité est de synthétiser avec efficacité les traitements mathématiques de type traitement du signal. La figure 2.8, présente le flot de conception intégrée de Catapult-C qui vise à la fois l'efficacité architecturale et la productivité pour la modélisation, le partitionnement et la synthèse de l'ensemble du système. La sortie RTL est générée en fonction des styles de codage C/C++ utilisés. Les ressources logicielles, telles que les registres et les tableaux, sont mappées à des ressources matérielles équivalentes et des machines à états finis sont générées pour accéder à ces ressources. CatapultC alloue les ressources de traitement à chaque portion de l'algorithme à synthétiser en fonction des contraintes de temps et de surfaces qui sont données. Les ressources de traitements peuvent être des unités fonctionnelles, des mémoires, des bus de communications etc. Les unités fonctionnelles peuvent être réutilisées pendant un cycle de calcul suivant le parallélisme dans l'algorithme.

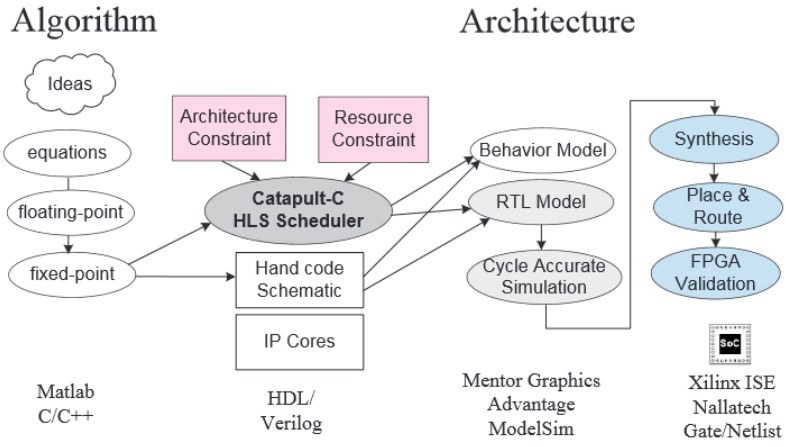



FIGURE 2.8 : Flot de conception HLS avec CatapultC [52]

L'outil CatapultC propose une interface utilisateur graphique et dispose de 3 types de simulations : par cycle, niveau RTL, et niveau porte-logique.

### 2.5.4 L'outil VIVADO HLS

La suite logicielle Xilinx Vivado High Level Synthesis (HLS) permet l'utilisation des langages C, C++, SystemC, ou Open Computing Language (OpenCL) pour définir un circuit logique synthétisable. La suite logicielle Xilinx Vivado HLS offre à l'utilisateur un environnement graphique intégré de production rapide des IP et d'assemblage de systèmes sur puce [53]. Les IP réalisées peuvent inclure des circuits logiques, des softcores, des modules de traitement de signal ou des descriptions algorithmiques rédigées en langage C. les IP tiers qui supportent le protocole IP-XACT peuvent être ajoutées à la bibliothèque *IP Catalog* de Vivado. La Bibliothèque Vivado IP Catalog permet au concepteur de sélectionner rapidement des IP, de les instancier et de les ajouter à son système sur puce ou à sa nouvelle IP.
Xilinx propose le protocole AXI4 pour gérer les interconnexions. Les IP déjà existantes peuvent intégrer le système soit sous forme RTL ou sous forme de netlist (voir la figure 2.9). Dans [54] Syed Z. A. et al. ont comparé deux IP obtenues par transformation en HDL du même algorithme de traitement d'image initialement rédigé en langage C. Les résultats ont montré que le concepteur gagne de cinq à dix fois plus de temps en utilisant Vivado HLS pour produire le RTL qu'en réalisant manuellement la transformation et que les ressources consommées par les deux IP étaient équivalentes. Seule la latence de l'IP faite à la main était meilleure que celle de l'IP générée automatiquement. Cette différence est due, selon les auteurs, à l'utilisation du Bus AXI4 et aux choix de mise en œuvre de la mémoire par l'outil de synthèse de Vivado HLS. Syed. Z. A. et al. concluent que la suite Vivado est bénéfique pour le concepteur qui veut tester une portion de son système en utilisant un bus d'interconnexion prêt à l'emploi. Ainsi Vivado HLS peut contribuer à réaliser en peu de temps un MP-RSoC ayant des softcores et des IP basées sur l'un des protocoles supportés comme dans [55], sans utiliser les langages HDL traditionnels. Il est recommandé au concepteur de bien définir ses contraintes au moment de la synthèse et de comprendre comment chaque construction en langage C est transformée en circuit logique pour réaliser des algorithmes en C qui seront synthétisés efficacement. La figure 2.9 présente le flot de conception standard de Xilinx Vivado HLS.

### 2.5.5 Limites à l'intégration des conceptions HLS dans les MP-RSoC

La difficulté actuelle pour intégrer des conceptions de niveau HLS dans le MP-RSoC est que les outils de transformation HLS fonctionnent très bien pour l'algorithme et le chemin de données, mais d'autres composants du système comme les interconnexions, les interfaces, les processeurs et les contrôleurs de mémoires sont uniquement disponibles dans le langage de

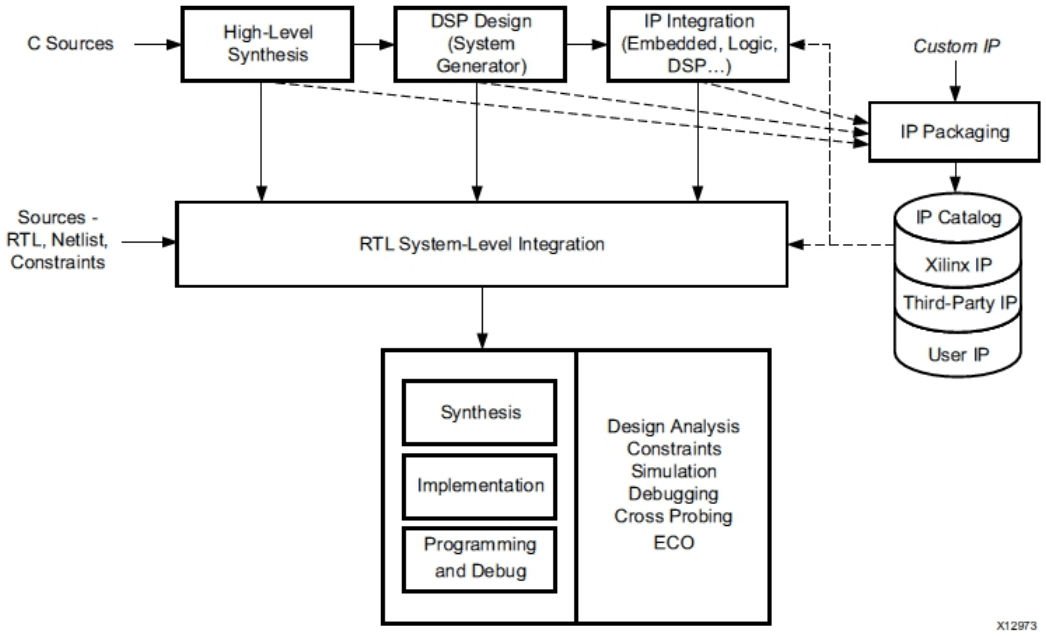



FIGURE 2.9 : Flot standard de conception avec VIVADO HLS [53]

description du matériel (HDL) comme Verilog ou VHDL puis transcrits au modèle RTL. Par conséquent, les concepteurs doivent connecter manuellement une interface RTL standard au modèle RTL de la conception HLS, puis modifier les vecteurs de test, les scripts de simulation pour ajuster les différences. Dans [56] le lecteur pourra trouver un état de l'art plus complet des outils HLS et dans [57] il est montré comment ces outils accélèrent le développement des SoC. Ces outils HLS ne disposent pas à l'heure actuelle de méthode pour gérer la reconfiguration dynamique, les rendant peu utilisables en l'état pour les MP-RSoC.

## 2.6 Les outils de niveau PBD

Pour dépasser les limites des outils de niveau HLS, il existe des outils qui proposent des plateformes pour la mise en œuvre de systèmes. Ces plateformes éliminent le besoin d'une exploration architecturale, toutefois elles sont souvent figées et ne conviennent qu'à une catégorie d'applications.

### 2.6.1 Projet FOSFOR (Flexible Operating System For Reconfigurable Platform)

Ce projet ANR a duré de 2008 à 2011, il a été mené par les laboratoires ETIS, LEAT et Thalès RT. Le but de FOSFOR est de proposer une plateforme hétérogène qui permette de mettre en œuvre des applications de traitement d'images composées de tâches logicielles et matérielles. La plateforme est intégrée dans un FPGA. Elle est composée de processeurs et de zone reconfigurables. Sur chaque processeur un OS de type AMP (Asymmetric Multi

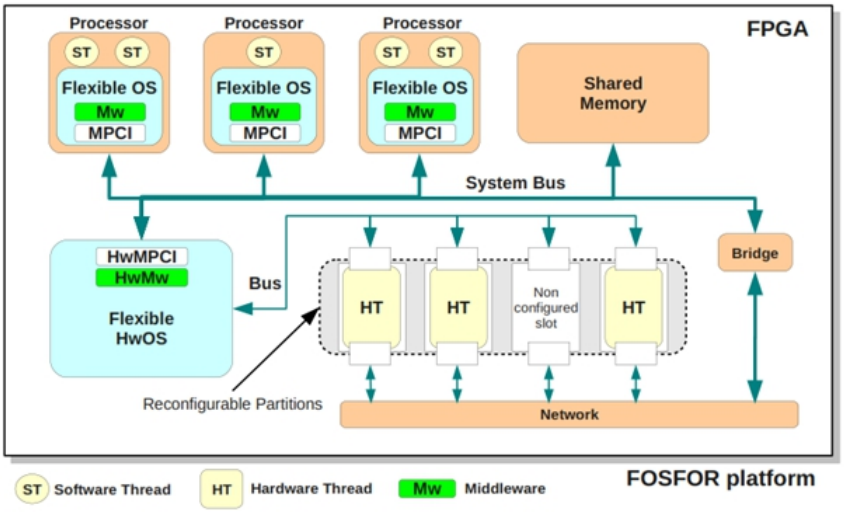



FIGURE 2.10 : plateforme MP-RSoC du projet FOSFOR [58]

Processor) est instancié, il s'agit de RTEMS [15] [59].
RTEMS propose une solution de services distribués aux UE de la plateforme FOSFOR.
Une version matérielle d'un OS basée sur RTEMs a été créée dans le projet FOSFOR [58]. Elle permet aux tâches matérielles d'accéder aux mêmes services que les tâches logicielles. La communication entre les UE logicielles ou matérielles se fait via la mémoire partagée. Les SoftCores sont des Leon3 [60].

Une interruption permet de solliciter l'UE concernée par un appel de service après avoir renseigné les paramètres de l'appel dans la mémoire partagée.

Les tâches matérielles ont accès à un NoC dédié pour communiquer entre elles ou pour accéder à la mémoire partagée afin de communiquer avec les autres UE.

Il y a quatre zones reconfigurables et l'OS matériel est capable de reconfigurer chaque zone. Le LEAT a réalisé une IP nommée FARM qui permet de gérer et d'accélérer la reconfiguration du FPGA. la reconfiguration dynamique a été testée avec des tâches qui accédaient au service sémaphore et avec une application de traitements d'images dans le cadre des travaux de L. Gantel [61].

Le projet FOSFOR illustre la plateforme MP-RSoC complète avec des modules statiques et des modules reconfigurables qui communiquent entre eux. L'utilisation d'un système d'exploitation pour gérer le système sur puce limite la plateforme à des modules logiciels et matériels qui sont compatibles avec cet OS et oblige le concepteur à maîtriser cet OS pour pouvoir l'utiliser. La plateforme FOSFOR est limitée à trois processeurs et quatre zones reconfigurables. Son passage à l'échelle pour gérer des milliers de modules parallèles sans détériorer les performances est difficile en raison de l'utilisation d'une mémoire partagée qui limite de facto le nombre de modules qu'il est possible de faire fonctionner en parallèle.

15. Real-Time Executive for Multiprocessor Systems

### 2.6.2 Le projet TMD-MPI

En 2006 Saldana et al. décrivent, dans le cadre du projet TMD-MPI [62], une architecture hétérogène reconfigurable pour le calcul à haut rendement (HPRC[16]) comprenant un processeur Intel Pentium 4, des hardcores IBM PowerPC, et des softcores Microblaze embarqués. Cette plateforme est réalisée à l'aide de cartes de développement Amirix AP1100 PCI utilisant des FPGA Xilinx Virtex 2 Pro 2VP100. L'architecture de TMD-MPI utilise une mise en œuvre de la bibliothèque MPI de programmation d'applications parallèles à mémoire distribuée et un réseau ad-hoc de type point à point. Cette plateforme a permis de résoudre l'équation de la distribution de la chaleur en temps réel par la méthode de Jacobi. Avec cette architecture, les auteurs proposent un modèle d'abstraction uniforme pour toutes les tâches logicielles et les accélérateurs ou tâches matérielles de l'application. Jusqu'à quarante cinq processeurs ont été interconnectés, montrant la capacité de MPI pour supporter un grand nombre de processeurs. Ce projet a permis de construire un système multiprocesseurs spécifique et performant en utilisant le standard ouvert et répandu MPI pour faciliter sa programmation par les ingénieurs d'architectures matérielles et les ingénieurs de développement logiciels.

Nous notons que la plateforme TMD-MPI n'est pas destinée à la conception de MP-RSoC monopuce, elle est plutôt adaptée à l'extension sur FPGA d'environnements HPC car l'initialisation des traitements est obligatoirement effectuée par un processeur Intel Pentium 4. La plateforme, dans sa version initiale, n'est pas flexible car elle ne prévoit pas de modules reconfigurables dynamiquement.

### 2.6.3 La plateforme SoC-MPI

Dans [63] Mahr P. et Al. présentent la plateforme SoC-MPI qui est une mise en œuvre des fonctions de la bibliothèque MPI destinée aux FPGA de Xilinx. SoC-MPI propose d'interconnecter des MicroBlazes à travers un choix de trois réseaux : point à point, étoile et bus. La plateforme ne requiert aucun système d'exploitation et se configure suivant les besoins de l'application. Cette plateforme permet de déployer l'approche communication par passage de message dans un FPGA mais elle est destinée au processeur MicroBlaze et ne prévoit ni l'hétérogénéité ni la reconfiguration dynamique. Le fait que SoC-MPI soit codée en C lui confère une performance inférieure à celle qu'on pourrait obtenir en utilisant une bibliothèque directement codée en Vhdl.

### 2.6.4 le projet BORPH

le projet BORPH[17] [64] facilite l'utilisation des FPGA en étendant les commandes du système d'exploitation Unix aux architectures matérielles reconfigurables. Ce projet a dé-

16. High Performance Reconfigurable Computer
17. Berkeley Operating system for ReProgrammable Hardware

fini une extension de fichier pour identifier les processus qui s'exécuteront sur un FPGA. Le système d'exploitation BORPH reconfigure partiellement le FPGA à travers un appel système effectué par une application Unix. Les tâches matérielles et les processus logiciels communiquent en utilisant les appels système d'entrée/sortie standard du système Unix. Le système d'exploitation BORPH réalise une abstraction du matériel du point de vue de l'application en utilisant le système d'exploitation standard Unix qui sert à la fois d'intergiciel sur les processeurs et sur le FPGA reconfigurable. Le co-design matériel et logiciel pour les applications mixtes s'en trouve accéléré et les fonctionnalités de l'application peuvent être efficacement déployées à travers la plateforme [65].

BORPH utilise un OS qui ajoute un niveau de complexité à la plateforme et nous montre surtout comment utiliser un FPGA pour étendre dynamiquement l'environnement d'exécution. Chaque tâche matérielle s'en refère à l'ordinateur hôte pour lire ou écrire des données ; cette approche peut fortement dégrader les performances de l'ensemble de l'application si le nombre de tâches matérielles qui communiquent simultanément avec l'OS devient important à un instant donné. BORPH est davantage destiné à accélérer des traitements sur une plateforme FPGA plutôt que de mettre en relation différents modules hétérogènes qui fonctionnent en parallèle sur la même puce pour réaliser une application.

### 2.6.5 le projet SPoRE

La plateforme SPoRE(Simple Parallel platform for Reconfigurable Environment) propose une architecture permettant de déployer sur multi-FPGA des PEs logiciels (SHP) et des PEs matériels (HSDP)[66].

La version SHP de SPoRE [67] utilise MPI et elle est composée de nœuds matérialisés dans l'implémentation par la carte de développement de type Xilinx ML507 qui comprend le FPGA Virtex 5 fx70t avec un processeur PPC 440. chaque nœud est organisé en cellule hôte et cellules de calculs. la cellule hôte exécute un OS µC/Linux pour gérer la plateforme. MPICH qui est l'une des plus populaires implémentations du standard MPI, a été portée sur cette plateforme. Les cellules de calculs sont construites autour du softcore Microblaze, il est possible d'instancier au maximum 8 Microblaze sur chaque nœud. La communication entre les cellules de calculs d'un même nœud est effectuée par mémoire partagée. Chaque cellule de calcul comprend un environnement d'exécution capable de prendre en charge les communications avec la cellule hôte via une MailBox.

La version HSDP [68] de SPoRE exploite la même architecture que la version SHP, avec en plus le support de la reconfiguration dynamique partielle à travers l'IP FaRM. Cette version n'exploite plus MPI pour les communications mais propose un environnement d'exécution utilisant un langage XML pour décrire et gérer les tâches matérielles de l'application parallèle.

La version HSDP de SPoRE est la plus aboutie en terme de performances et de fonc-

tionnalités que la version SHP car elle prend en compte la RDP pour les tâches matérielles. Toutefois, nous notons qu'elle est basée sur un langage de description non standard et par conséquent demanderait plus d'effort au concepteur pour la réutilisation des IPs. La plateforme matérielle est assez complexe et l'utilisation de la mémoire partagée compromet le passage à l'echelle des noeuds de la plateforme. L'utilisation d'un OS sur chaque nœud est source d'augmentation des ressources utilisées dans le MP-RSOC.

## 2.7 Les outils de niveau MDE/MDA

Certains outils se basent sur un modèle encore plus abstrait que ceux de niveau PBD. La communauté de l'EDA (*Electronic Desing Automation)* a vu se développer cette dernière décennie avec beaucoup d'intérêt des outils dits MDE (*Model Driven Engineering* ou MDA (*Model Driven Architecture*) qui proposent de déployer une application sur MP-RSoC en ayant recours à la modélisation abstraite d'un système. Le but recherché est de réduire le temps de conception. Le principe des MDE est de partir d'un modèle de l'application, un modèle de déploiement et un modèle de la plateforme matérielle et d'obtenir après plusieurs raffinements un modèle exécutable de l'application sur une cible. Les outils MDE / MDA utilisent préférentiellement le langage graphique UML (Unified Modeling Language) pour décrire les modèles.

### 2.7.1 Koski

T. Kangas et al. proposent dans [69] un profil UML 2.0 nommé TUT-profile permettant de modéliser une application parallèle. Ils proposent aussi une suite d'outils logiciels nommée Koski qui permet de réaliser une exploration architecturale en vue de générer les parties matérielles et logicielles automatiquement à partir d'un ensemble de contraintes et de librairies pré-conçues. Le résultat de ce processus est une configuration fonctionnant sur un FPGA. La figure 2.11 décrit le flot de conception en partant de la spécification des besoins, à la modélisation UML, puis à l'exploration architecturale jusqu'à l'implémentation physique. L'architecture matérielle produite à partir du profil UML TUT-profil et du framework Koski est réalisée à partir des librairies RTL qui intègrent le softcore NIOSII comme processeur embarqué et le bus HIBI pour la couche d'interconnexion. La partie logicielle est gérée par un RTOS [18] et des API pour la communication entre les processus. Koski exploite de nombreux outils externes pour générer le RTL et les exécutables.

18. Real Time Operating System

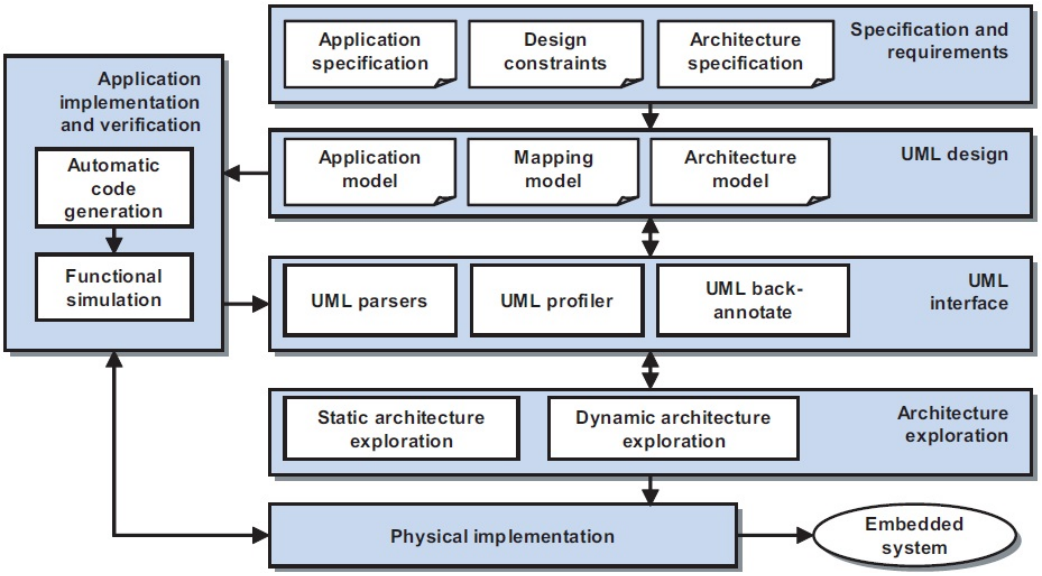



FIGURE 2.11 : Méthodologie de conception avec les outils Koski [70]

### 2.7.2 Gaspard

La méthodologie Gaspard [19] [71] adresse des applications de traitement intensif du signal mis en œuvre sur des grilles de processeurs, représentant un type particulier de plateforme qui comprend des structures répétitives. Gaspard favorise la co-modélisation application et plateforme ainsi que le co-design. Les modèles sont décrits en utilisant le profil UML MARTE (Modeling and Analysis of Real-Time and Embedded systems) de OMG [72].

Le modèle HRM (Hardware Resource Model) de MARTE permet de décrire la partie matérielle du système.

Le modèle RSM (Repetitive Structure Modeling) permet de décrire les structures répétitives. Enfin le modèle de composants génériques (GCM) est utilisé comme base pour modéliser les composants logiciels de l'application.

La figure 2.12 montre le flot de conception de Gaspard. On peut remarquer que la conception subit plusieurs raffinements et que de nombreux modèles et outils différents sont nécessaires pour arriver à un modèle RTL exécutable sur FPGA.

### 2.7.3 FAMOUS

Le projet FAMOUS présente un flot de conception inspiré de Gaspard, il prend en compte la reconfiguration dynamique du matériel à un haut niveau d'abstraction et propose les mécanismes nécessaires pour exploiter ses possibilités pendant l'exécution d'une application constituée de plusieurs tâches parallèles (voir figure 2.13). Le contrôleur de reconfigurations est modélisé par un ensemble d'automates de modes en utilisant les concepts de MARTE [73]. Chaque automate de mode représente les différents modes d'une tâche de l'application.

19. Graphical Array Specification for PARallel and Distributed computing

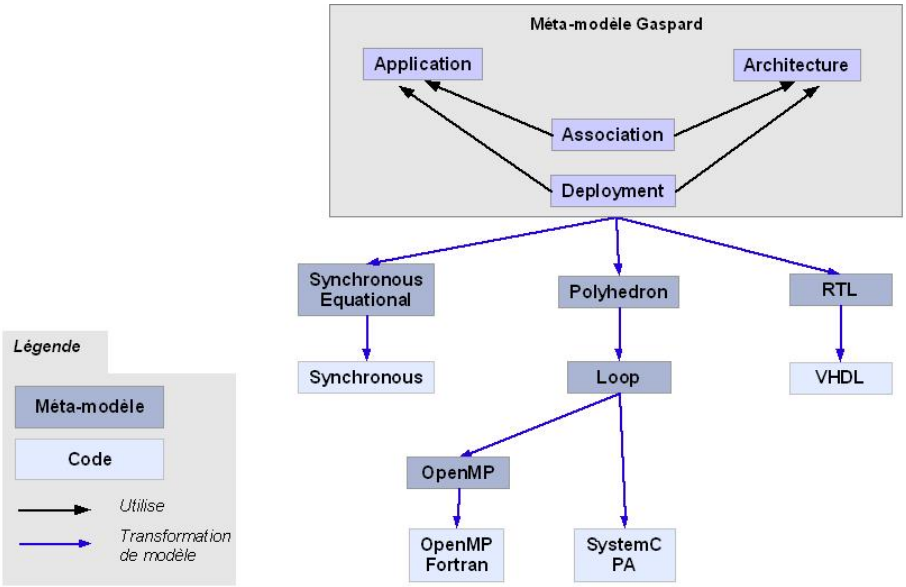



FIGURE 2.12 : Méthodologie de conception avec Gaspard [71]

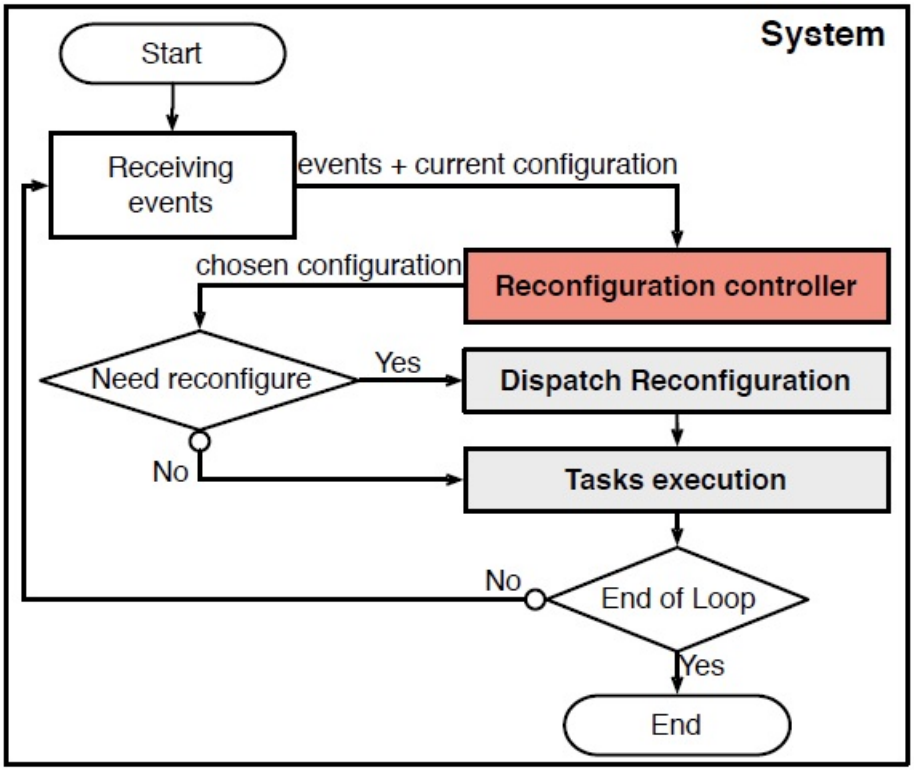



FIGURE 2.13 : Modèle d'exécution d'une application construite à l'aide de FAMOUS [73]

Les contraintes à respecter par l'ensemble des modes actifs des automates sont spécifiées sous la forme de contraintes «nfpConstraint» de MARTE. FAMOUS utilise l'outil SIGALI qui applique la technique DCS (Discrete Controller Synthesis) pour synthétiser automatiquement un contrôleur en langage C qui sera intégré à un projet XPS (Xilinx Platform Studio). Le rôle de ce contrôleur est de gérer automatiquement les valeurs des variables contrôlables à travers le resolver (voir figure 2.14) afin d'assurer le respect des contraintes pour toute évolution du système. FAMOUS est un projet qui vise spécifiquement le contrôle de la dynamicité des tâches constituant une application parallèle dans un environnement MP-RSoC. La communication entre les tâches est peu développée. Toute nouvelle variante de tâche entraîne automatiquement la génération d'un nouveau contrôleur à cause des nouvelles contraintes ce qui limite la souplesse du déploiement de l'application.

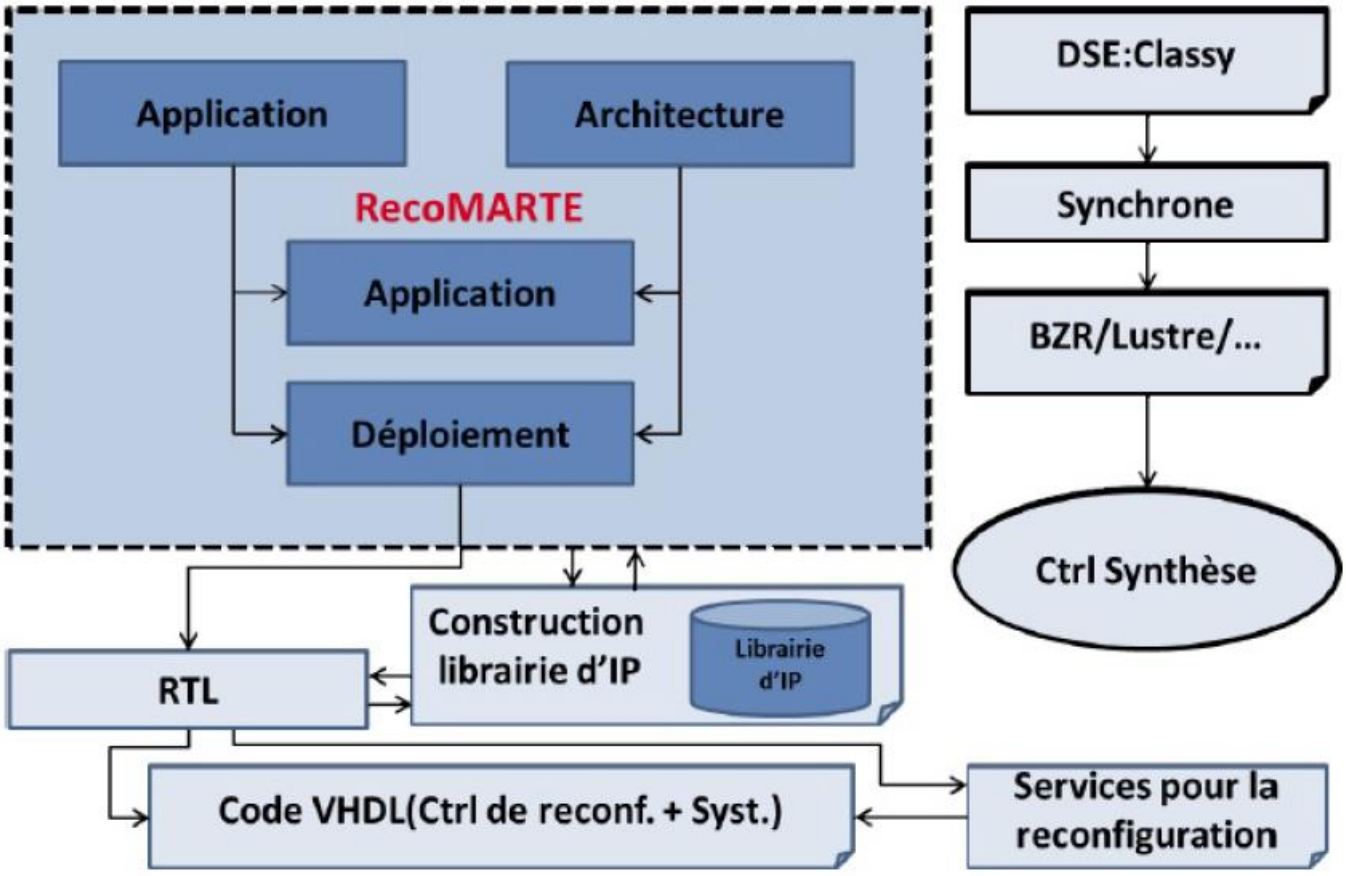



FIGURE 2.14 : Flot de conception FAMOUS [74]

### 2.7.4 MopCom

La méthodologie MopCom [75] peut être considérée comme un raffinement de l'approche MDA (Model Driven Architecture) intégrant l'exploration d'architecture, l'approche *"Platform Based Design"* et la réutilisation d'IP. Elle prend en entrée une architecture fonction-

nelle de type PIM *Plateform Independant Model* exprimée en SysML [20] qui est un langage de modélisation de OMG (Object Modeling Group) [76]. La figure 2.15 donne un aperçu du flot MoPCoM et met en exergue 3 niveaux de modélisation :

— le niveau AML (*Abstract Modeling Level*) adresse l'expression de la concurrence et de la communication sans présumer de la limitation des ressources,

— le niveau EML (*Execution Modeling Level*) définit une topologie physique abstraite permettant de faire des analyses à gros grains,

— le niveau DML (*Detailed Modeling Level*) décrit la plateforme de manière détaillée et permet une analyse fine menant à l'implémentation du système.

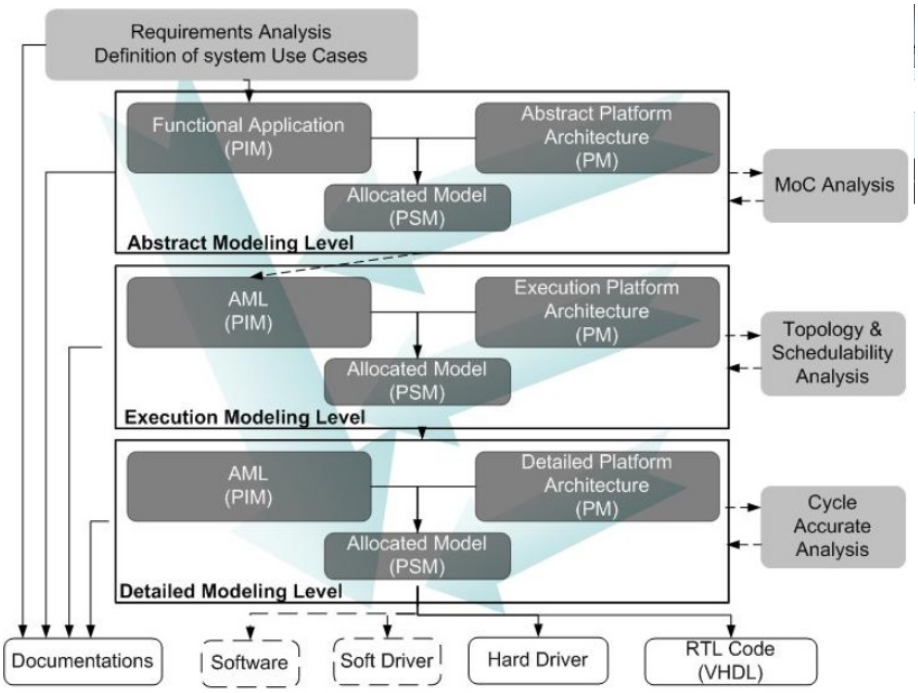



FIGURE 2.15 : Méthodologie de conception avec MopCom [75]

### 2.7.5 Les limites des outils MDE/MDA

Les outils MDE ne permettent pas une synthèse automatique de l'exécutable à partir du modèle, certaines étapes sont laissées à la responsabilité du concepteur. Quelques outils demandent au concepteur de fournir un grand nombre de paramètres et le contraignent à utiliser une plateforme ou un ensemble de composant pré-établis pour synthétiser le système à partir de son modèle abstrait. Le concepteur ne dispose généralement pas de tous les différents modèles abstraits pour les composants d'interconnexion, les processeurs ou les IP spécialisées ce qui empêche de prévoir une synthèse fidèle à la conception. Avec ces outils, le concepteur n'a pas de garantie sur les performances de l'application finale avant sa réalisation effective.

20. System Modeling Language

Le concepteur doit apprendre une méthodologie nouvelle et très souvent complexe pour utiliser ces outils.

## 2.8 Conclusion

Nous avons vu les outils pour reconfigurer dynamiquement un FPGA (GoAhead, PlanAhead, PARBIT,...) et présenté des plateformes MP-RSoC (SOC-MPI, FOSFOR, BORPH ...) en montrant leurs limites par rapport à l'utilisabilité et la flexibilité. Nous n'avons pas trouvé d'outils qui facilitent le déploiement d'application sur un MP-RSoC, en faisant abstraction des couches de communication entre les processeurs du MP-RSoC et supportant le passage à l'échelle. Nous avons noté avec intérêt l'utilisation des primitives de la bibliothèque MPI version 1 dans le projet TMD-MPI.

Ce chapitre nous a permis d'étudier les avancées technologiques et les méthodologies de conception qui fondent le développement d'applications sur MP-RSoC. Nous avons recensé les outils de l'état de l'art suivant quatre niveaux d'abstraction : le niveau RTL, le niveau HLS, le niveau PBD [21] et le niveau MDE [22]. Nous retenons que :

- La reconfiguration statique du MP-RSoC est prise en charge par les outils standards du marché (XST, PlanAhead, Impact), par contre pour les MP-RSoC reconfigurables dynamiquement, un flot plus complexe doit être suivi pour le déploiement de l'application. Nous avons présenté des outils qui facilitent partiellement le flot de reconfiguration des FPGA de Xilinx comme PARBIT ou GoAhead.
- Plutôt que de concevoir un MP-RSoC en partant de zéro, il peut être avantageux d'exploiter un modèle basé sur une plateforme qui nous facilite la tâche en intégrant le composant adéquat pour l'interconnexion et le modèle adéquat pour la programmation parallèle.
- Les modèles d'abstraction de haut niveau peuvent accélérer le temps de conception et de simulation d'une application pour MP-RSoC, mais la transformation d'un modèle abstrait en une application n'est ni automatique, ni triviale ; le concepteur doit utiliser plusieurs outils qui ne sont pas toujours bien intégrés et effectuer manuellement certaines phases de la méthodologie de conception.

Les nombreux outils et plateformes pour la conception d'applications sur MP-RSoC de l'état de l'art nous montrent l'intérêt des chercheurs et des industriels pour ce sujet. Les outils que nous avons analysés permettent d'implanter les tâches matérielles reconfigurables de l'application sur un MP-RSoC au prix de l'apprentissage de méthodes spécifiques liées à des technologies particulières. Nous n'avons recensé aucun outil qui adresse spécifiquement **la conception et le déploiement des applications parallèles construites à partir de tâches matérielles et logicielles hétérogènes** et qui satisfasse les besoins tels que la

21. Plateform Base Design
22. Model Driven Engineering

souplesse, la dynamicité et le passage à l'échelle.

Les EDA[23] de niveau MDE sont encore trop complexes pour réaliser une application parallèle sur MP-RSoC tandis que les outils de niveau HLS ne sont pas indiqués pour la mise en œuvre des tâches parallèles et reconfigurables le niveau PDB apparaît comme le plus pertinent.

Nous constatons aussi que les systèmes à mémoire partagée sont un frein au passage à l'échelle. Dans ce contexte, nous nous proposons d'étudier la conception et le déploiement d'applications sur un MP-RSoC à mémoire distribuée permettant de réduire les temps de développement tout en maintenant les performances du système et en garantissant un passage à l'échelle.

Dans le prochain chapitre, nous présentons notre première contribution qui est la conception de la plateforme MP-RSoC nommée MATIP permettant au concepteur VHDL de déployer une application parallèle dans un FPGA.

23. Electronic Design Automation

# Chapitre 3

# MATIP : Plateforme MP-RSoC utilisant MPI-2

## Sommaire

## 3.1 Introduction

Nous présentons dans le présent chapitre la plateforme MATIP [1] qui permet le déploiement d'applications parallèles sur un MP-RSoC. MATIP est un environnement permettant de déployer des tâches matérielles dans un MP-RSoC avec une configuration à mémoire distribuée. Nous avons choisi de représenter MATIP avec une structure en trois couches en nous inspirant des travaux décrits par Pavel ZayKov dans [77] et par Saldaña et al. dans [62] (voir figure 3.1). L'organisation en couche nous permet de découper fonctionnellement les composants constituants notre plateforme. La couche d'interconnexion réalise les liaisons

1. MPI Application Task Intégration Platform

physiques permettant le transit des données entre les tâches dans le MP-RSoC. La couche de communication offre des services de communication aux tâches. La couche application regroupe l'ensemble des tâches qui constituent l'application.

Dans la suite de ce chapitre, la section 2 décrit le standard MPI, en particulier MPI-2 RMA et son intérêt dans le déploiement d'applications parallèles dans un MP-RSoC ; la section 3 introduit les principaux concepts de la plateforme MATIP ; la section 4 présente une conception détaillée de la couche d'interconnexion ; la section 5 donne le détail du processeur de communication nommé MPI-HCL de MATIP ; La section 6 est consacrée au modèle de tâches reconfigurables et la section 7 est une conclusion.

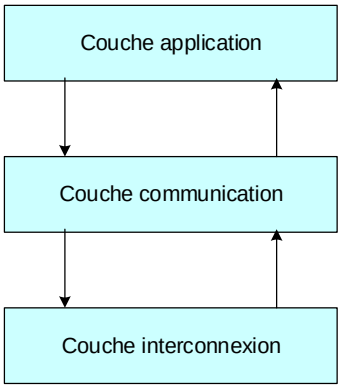



FIGURE 3.1 : Modèle en couches de MATIP

## 3.2 La programmation parallèle et le MPI

Afin d'assurer le passage à l'échelle, nous avons fait le choix d'un système distribué ; dans ce contexte, MPI est un candidat naturel que nous avons retenu.

### 3.2.1 Le standard MPI et ses différentes mises en œuvre.

MPI pour « Message Passing Interface » est une norme conçue en 1994 par une association d'une quarantaine d'organisations [78]. Elle définit une bibliothèque dédiée à la programmation parallèle par passage de messages dans des environnements multiprocesseurs. Grâce à cette bibliothèque, il est possible d'écrire des applications parallèles indifféremment du type de système sur lequel elles sont exploitées. Cette bibliothèque fonctionne aussi bien sur des machines multiprocesseurs que sur des clusters de machines, des machines à mémoire partagée ou distribuée. MPI est disponible pour toutes les plates-formes de programmation parallèle grâce aux implémentations LAM [79] et MPICH [80] qui relèvent du domaine public. La première version de MPI est notée MPI-1 et propose les fonctionnalités suivantes :

— fourniture d'un environnement de développement parallèle ;

— communications point-à-point ;

— communications collectives ;
— types de données dérivés : MPI fournit pour chaque type de donnée existant (int, char, ...) un type correspondant afin de les rendre portables vers d'autres environnements (par exemple, un entier, « int » en C, aura pour type en MPI « MPI_INT »). Cette règle fonctionne pour les types de données classiques mais aussi pour les structures ;
— groupes et communicateurs : il est possible de regrouper les processus actifs pour envoyer des données à traiter à un groupe sans choisir précisément quel processus sera concerné par le traitement. Les communicateurs sont utilisés pour désigner les processus concernés par la communication, par défaut les processus communiquent dans un communicateur appelé MPI_COMM_WORLD.

En 1997, une seconde version de MPI, MPI-2 est introduite. Elle apporte les améliorations suivantes :

— la gestion dynamique des processus : il devient possible de créer et de supprimer des processus durant l'exécution du programme.
— les copies de mémoire à mémoire : avec MPI-1, il existait les communications point-à-point par échange de messages. Les deux processus devaient être disponibles en même temps pour que la communication puisse avoir lieu. Avec MPI-2, il est désormais possible d'accéder directement à la mémoire d'un processus distant. Ce mécanisme est appelé RMA pour « Remote Memory Access ». Cette nouvelle forme de communication permet de libérer le processus distant, ce qui apporte de meilleures performances.

### 3.2.2 MPI-2 et RMA.

Le concept de communication par copies de mémoire à mémoire est un concept puissant mais pas nouveau. MPI ayant simplement unifié les solutions déjà existantes telles que shmen de CRAY [81], lapi d'IBM [82], en proposant ses propres primitives RMA. Via ces fonctions, un processus a directement accès en lecture, écriture ou mise à jour, à la mémoire d'un autre processus distant. Dans cette approche le processus distant n'a pas à intervenir dans la procédure de transfert. Les principaux avantages du RMA sont les suivants :

— des performances améliorées lorsque le matériel le permet : possibilité de réduire les données de synchronisation nécessaires lors du passage de messages.
— une programmation plus simple de certains algorithmes : RMA libère le processeur de certaines procédures de transfert explicite de données.

L'approche RMA de MPI peut être divisée en trois parties distinctes :

1. définition sur chaque processeur d'une zone mémoire (fenêtre mémoire locale) visible et susceptible d'être accédée par des processus distants ;
2. déclenchement du transfert des données directement de la mémoire d'un processus à celle d'un autre processus. Il faut alors spécifier le type, le nombre et la localisation

initiale et finale des données ;

3. Achèvement des transferts en cours par une étape de synchronisation, les données étant alors réellement disponibles pour les calculs.

MPI permet à un processus de lire à l'aide de la primitive MPI_GET(), d'écrire via la primitive MPI_PUT() et de mettre à jour en utilisant la primitive MPI_ACCUMULATE(), des données situées dans la fenêtre mémoire locale d'un processus distant.

On nomme "origine" le processus qui initie le transfert et "cible" le processus qui possède la fenêtre mémoire locale qui va être utilisée dans la procédure de transfert.

Lors de l'initialisation du transfert, le processus cible n'appelle aucune primitive MPI. Toutes les informations nécessaires sont spécifiées sous forme de paramètres lors de l'appel de la primitive MPI par l'origine.

Remarques :

- La syntaxe de MPI_GET est identique à celle de MPI_PUT, seul le sens de transfert des données est inversé.
- Sur le processus cible, les seules données accessibles sont celles contenues dans la fenêtre mémoire locale.
- Les primitives de transfert de données RMA sont des primitives non bloquantes (choix délibéré du standard MPI).

Un transfert non bloquant a comme caractéristique de s'achever même lorsque la cible n'est pas "disponible". Ce type de transfert peut nécessiter de la mémoire tampon pour stocker les messages sortants et entrants.

### 3.2.3 Intérêt de MPI2-RMA pour les MP-RSoC

Les messages peuvent être échangés entre les tâches matérielles même lorsque celles-ci sont "indisponibles". MPI-2 autorise des communications avec cibles actives et avec cibles passives ; cette deuxième méthode ouvre la voie à plus de souplesse dans un environnement dynamique où les tâches matérielles peuvent changer d'un état actif à un état inactif et vis versa sans que leurs communications ne soient interrompues. Ceci n'est pas possible avec MPI-1 qui oblige les tâches communicantes à être actives et synchronisées lors des communications. RMA en permettant l'accès direct à la mémoire de la cible accélère la délivrance des données lors des communications entre tâches matérielles. **Avec MPI2-RMA, les communications par passage de message sont plus souples du point de vue du passage à l'échelle que des communications par mémoire partagée et le mécanisme d'accès direct à la mémoire qui réduit la latence des communications**.

## 3.3 Couche d'interconnexion

Avant de mettre en œuvre matériellement des primitives de communication MPI, il est nécessaire de disposer d'un support physique de communication. Ce support correspond à la couche d'interconnexion de la figure 3.1. La régularité temporelle et spatiale des communications d'une application parallèle définit son profil de communication [83] et ce dernier permet de caractériser les besoins de communication de l'application parallèle. L'interconnexion que nous nous proposons de réaliser doit satisfaire les besoins des principaux profils de communications parmi lesquels nous pouvons citer : la **diffusion** (Un-vers plusieurs), la **réduction** (Plusieurs vers un), la **communication synchrone** (de plusieurs UE vers plusieurs UE), la **communication asynchrone** (de plusieurs UE vers plusieurs UE), la **communication quelconque** entre les UE.

### 3.3.1 Types d'interconnexions

La couche d'interconnexion doit satisfaire les différents profils de communication des applications parallèles, supporter une grande densité de communication et avoir une connectivité compatible avec la reconfiguration dynamique du MP-RSoC. Le rapport connectivité sur contention [84], nous a également guidés lors du choix de la structure. Dans un réseau statique de type bus, la connectivité est totale entre les nœuds mais la contention augmente avec le nombre de nœuds connectés. Dans un réseau point-à-point, la connectivité est totale et il n'y a pas de contention mais lorsque le nombre de noeuds augmente il devient difficile de relier physiquement tous les noeuds entre eux (voir figure 3.2). La méthode de routage

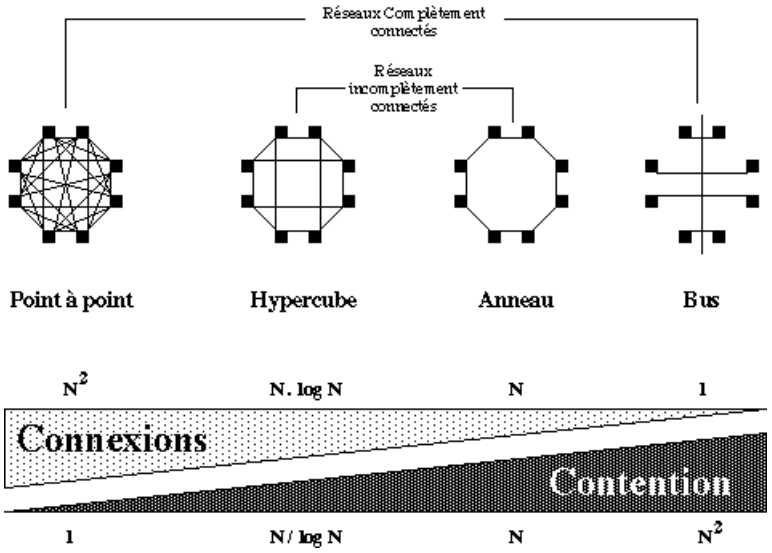



FIGURE 3.2 : Rapport entre connexions et contention dans un réseau (d'après [83])

des paquets est également une caractéristique importante des réseaux. Lorsque le routage

est statique, le chemin à suivre par les données est fixé à l'avance lors de la conception du réseau. Lorsque le routage est dynamique, le chemin des données à travers le réseau est défini par programmation de la matrice d'interconnexion. Plusieurs interconnexions sont possibles pour réaliser un réseau dynamique ;

- le bus,
- l'étoile
- l'anneau à jeton

Il existe plusieurs variantes de bus, d'étoiles et d'anneaux. Il est possible de définir des architectures hybrides. (ie des architectures mêlant bus et étoile dans un même réseau, bus et anneau, etc.). Les différentes architectures peuvent ensuite être cascadées à l'aide de pont réseau ou hiérarchisées (architecture multi-étage). Cette configuration est comparable à un arbre ayant une racine, des branches et au bout des feuilles représentant les hôtes du réseau. Dans les architectures réseau multi-étages, chaque étoile élémentaire ou chaque bus élémentaire est connecté à d'autres bus de niveau supérieur à l'aide de liens spécialisés. Le *crossbar* est à la base de l'architecture en étoile et de l'architecture multi-étages.

Dans les paragraphes qui suivent, nous présentons les deux interconnexions les plus utilisées dans les SoC.

#### 3.3.1.1 L'interconnexion à l'aide d'un bus

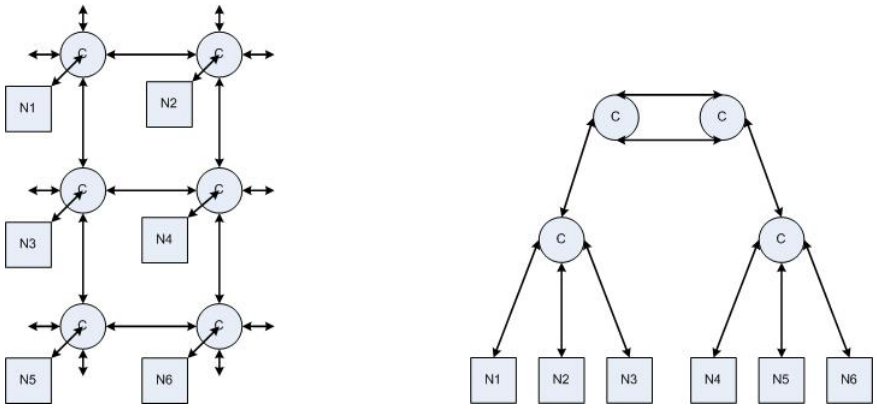
Dans la littérature, il existe des bus hiérarchisés dotés de contrôleurs intelligents et de protocoles élaborés qui sont utilisés dans la réalisation des systèmes sur puce, c'est le cas du bus AMBA (advanced Microcontroler Bus Architecture) développé par ARM Ltd [85] dans les années 1996, qui est constitué de bus AHB [2] pour les processeurs et la mémoire et APB [3] pour les périphériques plus lents et depuis 2003 AXI [4] [86] pour l'interconnexion des IP. Toutefois, l'utilisation d'un bus est valide pour des plateformes limitées à une ou deux dizaines d'unités d'exécution, au-delà, les problèmes de latence lors des communications peuvent déprécier considérablement les performances de toute la plateforme de traitement parallèle.

#### 3.3.1.2 L'interconnexion à l'aide d'un réseau sur puce

Pour des systèmes ayant plus d'une dizaine de nœuds, les réseaux sur puce (NoC [5]) ont été développés. Le NoC désigne un réseau construit à partir d'un ensemble de commutateurs reliés entre eux suivant une architecture particulière (voir figure 3.3) et mettant en œuvre un protocole de routage pour acheminer les données entre les nœuds. L'architecture en grille et l'architecture en arbre sont les plus utilisées dans les NoC comme sur la figure 3.3. Dans [87]

2. Advanced High speed Bus
3. Advanced Peripheral Bus
4. Advanced eXtended Interface
5. Network on Chip

les auteurs du projets ReNoC annoncent une bande passante de 2.4 Gb/s à 100 MHz pour leur NoC reconfigurable 32 bits de type grille. Le NoC est plus efficace qu'un bus lorsque le nombre de nœuds à mettre en relation augmente et que plusieurs échanges simultanés sont nécessaires à travers les UE de la plateforme, mais il occupe plus de ressources logiques et exige pour son interface, sur chaque noeud, une logique de contrôle spécifique appelée NIC [6]. Dès lors tous les périphériques et toutes les unités d'exécution de la plateforme doivent être compatibles avec un NoC donné et posséder le NIC correspondant ; cette exigence a pour conséquence de limiter le choix des nœuds (unités d'exécution et périphériques) de la plateforme en fonction du NoC retenu pour cette plateforme ou d'ajouter un coût pour interfacer chaque nœud avec le NoC.




(a) grille (b) arbre

FIGURE 3.3 : Principe d'un réseau sur puce

#### 3.3.1.3 Choix de l'interconnexion pour un MP-RSoC

Le choix du réseau d'interconnexion détermine les performances globales de la plateforme et conditionne la dynamicité de l'architecture. La largeur des bus de données du réseau impacte directement le débit de la plateforme. Le réseau point-à-point est difficilement compatible avec la reconfiguration dynamique car pour l'ajout d'un nouveau nœud, il faut ajouter une liaison physique entre le nouveau nœud et tous les anciens nœuds du MP-RSoC. Dans un réseau statique comme la grille, l'anneau, ou l'hypercube, les communications transitent entre les nœuds pour atteindre leur destination ; un routage des paquets doit être mis en place pour assurer l'interconnexion entre deux nœuds non contigus, ce qui rend cette structure complexe. Les réseaux statiques sont mal adaptés pour un MP-RSoC car la destination des messages est figée lors de la conception du réseau. Nous avons choisi de mettre en œuvre un réseau dynamique pour sa flexibilité.

Nous avons choisi, parmi les réseaux dynamiques, le plus simple, constitué d'un seul commutateur ou *crossbar* qui ne nécessite pas de routage complexe et qui permet de réaliser un réseau dynamique compatible avec la reconfiguration dynamique [84].

6. network Interface controller

### 3.3.2 Crossbar pour MP-RSoC

Bien que l'objet de nos travaux ne consiste pas à réaliser une interconnexion, nous devons en réaliser une pour un MP-RSoC qui possède des caractéristiques compatibles avec un système distribué intégrant un grand nombre d'UE. Ces caractéristiques sont :

1. une faible latence pour optimiser les performances temporelles ;
2. une faible empreinte pour limiter les ressources occupées par ce composant ;
3. un routage dynamique pour s'adapter aux tâches matérielles qui seront elles-mêmes dynamiques ;
4. une taille modulable en rapport avec le nombre de tâches matérielles devant communiquer.

La figure 3.4 montre le principe du réseau d'interconnexion ; le *crossbar* est au centre puis chaque noeud dispose d'un port d'entrée et d'un port de sortie pour être relié au réseau. Le

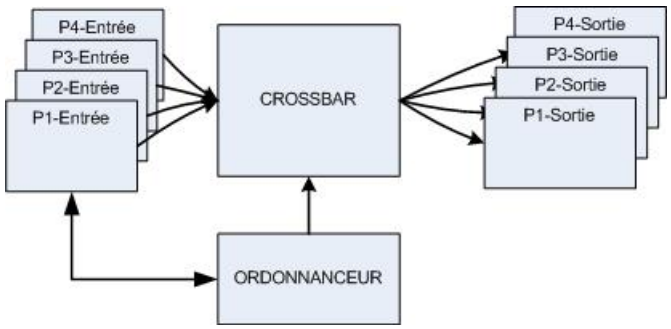



FIGURE 3.4 : Schéma de principe du réseau d'interconnexion

*crossbar* permet l'interconnexion dynamique des différents nœuds. Pour réaliser un réseau d'interconnexion capable de transmettre des messages entre les nœuds, le *crossbar* doit être complété d'un ordonnanceur pour gérer les requêtes et organiser la priorité des nœuds qui veulent communiquer [88]. Il est nécessaire de prévoir un mécanisme de synchronisation des transferts entre les noeuds, un mécanisme qui prévient les blocages et un mécanisme d'adressage de chaque nœud.

#### 3.3.2.1 La matrice interconnectée élémentaire (Crossbar)

Un *crossbar* élémentaire NxN est constitué de N lignes horizontales connectées aux entrées et de N colonnes verticales connectées aux sorties : le *crossbar* est chargé d'interconnecter physiquement les ports d'entrée et de sortie. Il peut être représenté à l'aide d'une grille et une réalisation possible est montrée sur la figure 3.5. L'interconnexion entre deux ports est gérée par un ordonnanceur qui arbitre les requêtes.

La matrice interconnectée met directement en relation chaque port d'entrée avec un port de sortie distinct, ceci permet d'obtenir un rapport connexion-contention acceptable pour notre plateforme MP-RSoC. Nous utilisons le *crossbar* dans un premier temps pour valider

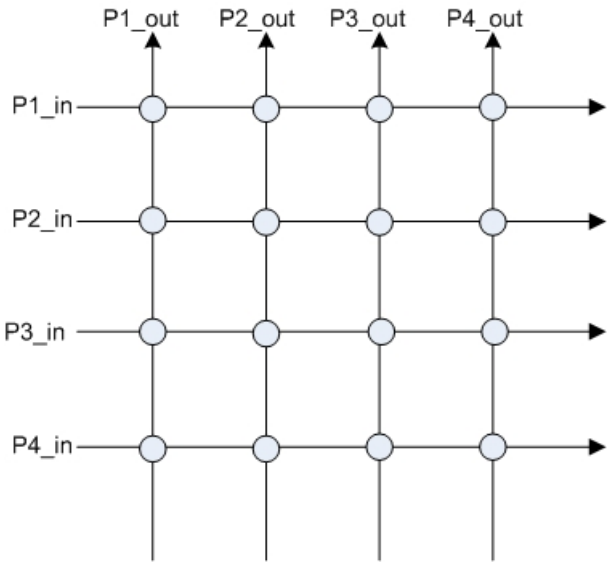



FIGURE 3.5 : Matrice interconnectée (*Crossbar*)

la plateforme MATIP. Ce réseau, malgré son encombrement, possède des propriétés en terme de bande passante et de dynamicité compatibles avec nos besoins.

La figure 3.6 présente l'architecture du réseau intégral dans sa version 4x4 [7]. Il est constitué de quatre modules :

- le module de gestion des ports d'entrée qui gère l'adressage et synchronise les messages entrants dans le *crossbar* ;
- le module de gestion des ports de sortie qui synchronise la relève des messages en sortie du *crossbar* et gère la destruction des messages non relevés après un délai ;
- l'ordonnanceur qui est chargé de gérer les transferts tout en étant équitable au niveau des priorités d'émission ;
- le *crossbar* qui permet de mettre en relation les modules d'entrées et de sorties.

L'objectif recherché lors de la conception du réseau d'interconnexion est d'obtenir le plus grand débit envisageable par cette architecture, la plus petite latence et une consommation de ressources minimale.

#### 3.3.2.2 Gestionnaire de port d'entrée

Chaque port d'entrée comprend un gestionnaire de port. Ces modules constituent l'interface d'entrée du réseau d'interconnexion, ils reçoivent les données en provenance des nœuds de traitement et sont chargés d'effectuer les tâches suivantes :

- stocker les données entrantes dans une mémoire tampon de type FIFO et avertir le nœud lorsque cette mémoire tampon est pleine ;
- déterminer sur la base des champs de l'en-tête du paquet à transmettre, les ports de destination et adresser les requêtes d'émission à l'ordonnanceur ;

7. 4 entrées, 4 sorties

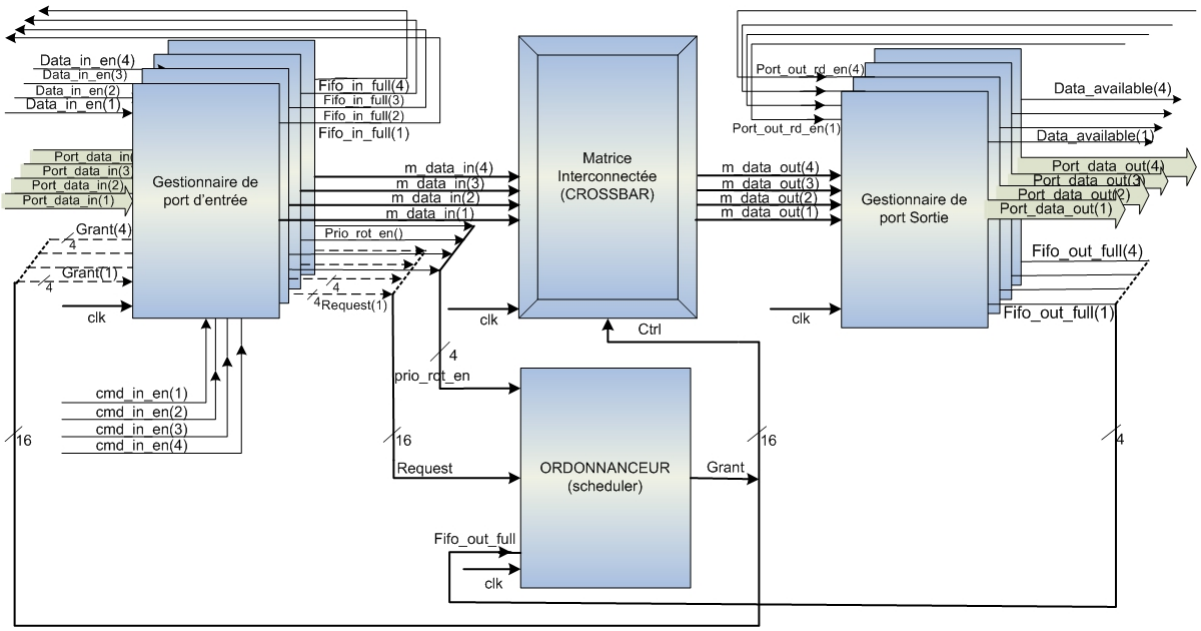



FIGURE 3.6 : Architecture du réseau d'interconnexion à base du crossbar 4x4

- libérer la FIFO lorsque les données ont été transmises sur le réseau. ;

La figure 3.7 donne le schéma synoptique des modules d'entrée du réseau. Le signal *prio_ rot_ en* lorsqu'il est à l'état haut indique que le gestionnaire de port d'entrée occupe de façon prioritaire un port de sortie du *crossbar* afin de transmettre les données. Le bus *Request* véhicule le numéro du port de destination qui est sollicité et un signal de validation de l'adresse *request_ en*. Lorsque le signal *Grant* retourne la même valeur que le signal *Request* alors le gestionnaire de port active le signal *prio_ rot_ en* et peut émettre des données vers le *crossbar*. Lorsqu'il a fini d'émettre la trame de données, il ramène le signal *prio_ rot_ en* au niveau bas pour libérer le port de destination du *crossbar*.

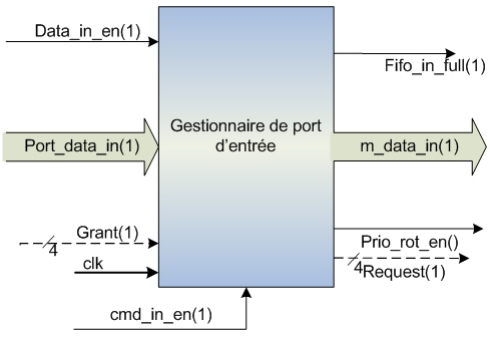



FIGURE 3.7 : Gestionnaire de port d'entrée n° 1 pour le réseau

Nous avons fait le choix d'un ordonnanceur utilisant l'algorithme DPA (Diagonal Propagation Arbiter) [88]. DPA permet de répartir équitablement la priorité à chaque nœud du réseau et de réduire la complexité de l'ordonnanceur.

Afin d'éviter le blocage du réseau, nous avons ajouté à l'entrée et à la sortie du *crossbar* une

mémoire FIFO et un opérateur de destruction de paquets non relevés sur chaque port de sortie. Lorsqu'un port de sortie reçoit un paquet, une temporisation de durée de vie du paquet (TTL [8]) est enclenchée. Le paquet est automatiquement détruit s'il n'est pas relevé au bout du délai fixé. Le composant réseau accepte des paquets qui ont le format de la figure 3.8. Nous avons fait le choix d'une taille de paquet variable ; toutefois le nombre maximum d'octets de la trame de données est fixé à 256 pour limiter l'empreinte mémoire.

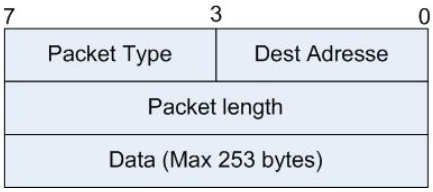



FIGURE 3.8 : Trame des paquets circulant sur le réseau

### 3.3.3 Ordonnanceur

L'ordonnanceur est basé sur une matrice à deux dimensions de cellules arbitres (figure 3.9 et 3.10a) avec propagation de retenue. Il reçoit les demandes d'émission des gestionnaires de port d'entrée et détermine les couples (port d'entrée $i$, port de sortie $j$) qui seront mis en relation pendant un intervalle de temps ; $\text{Grant}_{i,j} = 1$ si relation, 0 sinon. L'ordonnanceur doit assurer les relations de l'équation (3.1) à tout instant :

$$\forall i \sum_{j=1}^{N} G_{i,j} \leq 1 \text{ pour } (1 \leq i \leq N) \text{ et } \forall i \sum_{i=1}^{N} G_{i,j} \leq 1 \text{ pour } (1 \leq j \leq N) \qquad (3.1)$$

L'ordonnanceur à travers le vecteur *Grant()* d'une longueur de 16 bits indique les cellules qui ont été validées. Ce vecteur est connecté aux modules gestionnaires de port d'entrée pour leur notifier que leur demande d'émission a été validée ou non.

Le vecteur *Grant()* assure que chaque port d'entrée adresse au maximum un port de sortie et que chaque port de sortie est en relation avec au maximum un port d'entrée. Une cellule arbitre possède trois entrées et trois sorties (voir figure 3.10). $R_{i,j}$ est l'entrée de requête indiquant à l'ordonnanceur que le port d'entrée $i$ veut transmettre des données vers le port de sortie $j$. $G_{i,j}$ est une sortie qui permet à l'ordonnanceur d'indiquer que la demande de transmission $R_{i,j}$ a été acceptée ; ce signal est utilisé pour configurer la matrice interconnectée.

Toutes les requêtes de communication qui arrivent à l'arbitre sont regroupées dans un vecteur *Request*. Pour un *crossbar* 4x4, chaque port d'entrée dispose de cinq bits pour coder sa requête le bit de poids faible est à 1 lorsque la requête est valide, les bits 1 et 2 codent le numéro de

8. Time To Live

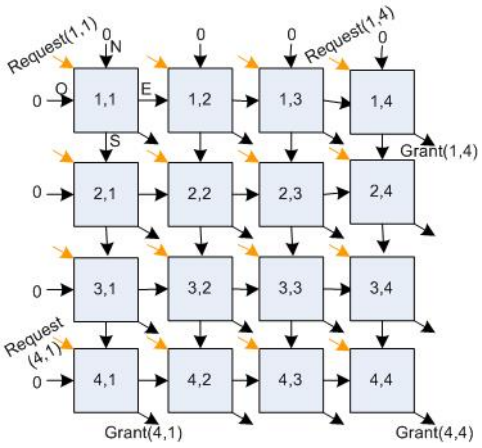
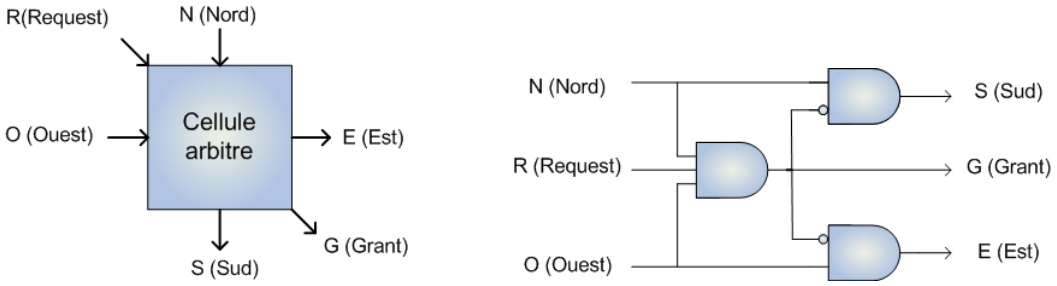



FIGURE 3.9 : Arbitre composé de cellules à propagation de retenue pour *crossbar* 4x4. La première ligne et la première colonne ont respectivement toutes leurs entrées N et O à 0 pour indiquer qu'il n'y a pas de cellule active à gauche et au-dessus. La sortie $Grant(i, j)$ pour chaque cellule dépend des entrées Request, Nord et Ouest de la même cellule.

port d'origine et les bits 3 et 4 codent le numéro de port de destination. Un décodeur dans l'arbitre permet d'appliquer le signal R à la cellule de l'arbitre correspondant.




(a) interface (b) logigramme

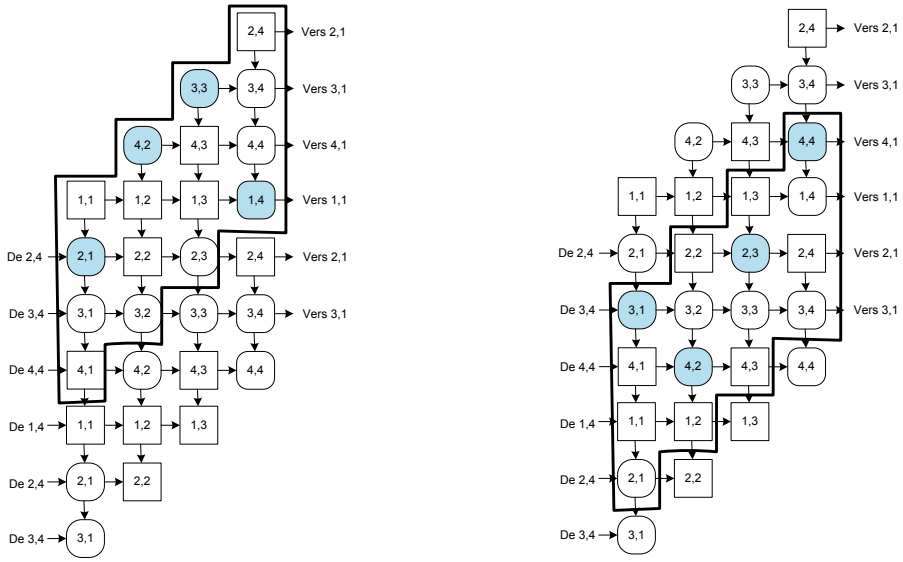
FIGURE 3.10 : Cellule arbitre : (a) interface et en (b) un exemple de mise en œvre en logique combinatoire [88].

Les entrées $N_{i,j}$ et $O_{i,j}$ proviennent des sorties des cellules adjacentes et permettent d'inhiber les autres cellules de telle sorte que les relations vues en équation (3.1) soient toujours vérifiées. Les entrées *Ouest* des cellules de la première colonne ainsi que les entrées *Nord* des cellules de la première ligne sont mises au niveau logique haut. Lorsqu'une cellule $i, j$ n'est pas autorisée à effectuer un transfert ($G_{(i,j)} = 0$), ses sorties $S_{(i,j)}$ et $E_{(i,j)}$ sont au niveau logique haut. Elles sont au niveau logique bas lorsque la cellule est autorisée. Ainsi, une cellule $(i, j)$ est autorisée si et seulement si elle reçoit une requête $R_{(i,j)}$ et si ses entrées $N_{(i,j)}$ et $O_{(i,j)}$ sont au niveau logique haut. Ce qui est le cas lorsqu'aucune des cellules de la colonne $j$ au-dessus de la ligne $i$ et aucune des cellules de la ligne $i$ à gauche de la colonne $j$ n'est autorisée. Le principe du DPA est d'affecter la même priorité d'activation aux cellules situées sur la même diagonale car les cellules $(i, j)$ situées sur une diagonale n'entrent pas en conflit les unes avec les autres. Ces cellules sont dites indépendantes les unes des autres.

Considérons par exemple les cellules (1,1), (4,2), (3,3), (2,4) de la première diagonale ;

ces cellules sont indépendantes les unes des autres, et peuvent être validées ensemble.

Le port d'entrée 1 peut émettre des données vers le port de sortie 1 pendant que le port d'entrée 4 émet des données vers le port 2 et ainsi de suite. Les cellules (2,1), (1,2), (4,3), (3,4) sont également indépendantes. Les cellules indépendantes sont mises en diagonale puis les N-1 premières diagonales sont dupliquées. Pour un arbitre 4x4 cellules chargé de gérer un *crossbar* 4x4, on obtient ainsi 7 diagonales. A chaque diagonale est associé un bit du vecteur de priorité P. Lorsque le bit de priorité $i$ vaut 0 aucune cellule de la diagonale $i$ ne peut être activée. Lorsque le bit de priorité $i$ vaut 1 alors les cellules de la diagonale $i$ peuvent être activées si aucune cellule au-dessus d'elle n'est active. A la fin de chaque transfert de données, le vecteur de priorité P subit une rotation et donne la plus grande priorité à une nouvelle diagonale pour rendre l'ordonnanceur équitable [88].
Sur le schéma de la figure 3.11 plusieurs cellules *Request{(2,1) ; (4,2) ; (3,3) ; (3,4) ; (3,1) ; (4,4) ; (3,2) ; (2,3)}* demandent à émettre simultanément. Sur la figure 3.11a celles qui sont sur la diagonale 1 sont prioritaires puis celles qui sont sur la diagonale 2 et enfin celle qui est sur la diagonale 4 car le vecteur de priorité à ce moment vaut $P = 1111000$. Par contre, sur la figure 3.11b, les cellules de la diagonale 3 sont prioritaires puis celles de la diagonale 4 et enfin celles de la diagonale 5 car le vecteur de priorité $P = 0011110$. Ainsi chaque cellule qui demande à émettre aura la priorité en fonction de la valeur du vecteur de priorité d'où l'équité de l'ordonnanceur DPA.




(a) diag 1 (b) diag 2

FIGURE 3.11 : Architecture DPA. (a) la diagonale (1,1) a la plus haute priorité. (b) la diagonale (3,1) a la plus haute priorité. Les cellules avec bords arrondis ont effectué une requête vers l'arbitre. Les cellules ombragées sont celles qui sont actives.

La rotation de la priorité est réalisée à l'aide du vecteur P. Dans le cas d'un *crossbar* 4x4 le vecteur P est initialisé à 1111000. P subit une rotation vers la droite à chaque cycle

d'arbitrage et il est réinitialisé à 1111000 lorsqu'il vaut 0001111. Ainsi la zone des cellules actives se déplace du haut vers le bas. La première diagonale de la fenêtre est toujours celle de plus haute priorité. La structure de l'arbitre de base est modifiée afin d'inclure le signal de priorité $P_r$ où $P_r$ est $r^{ime}$ élément du vecteur P de longueur 2N – 1. Ceci est illustré à la figure 3.11.

La figure 3.13 présente l'ordonnanceur du réseau d'interconnexion que nous avons réalisé.

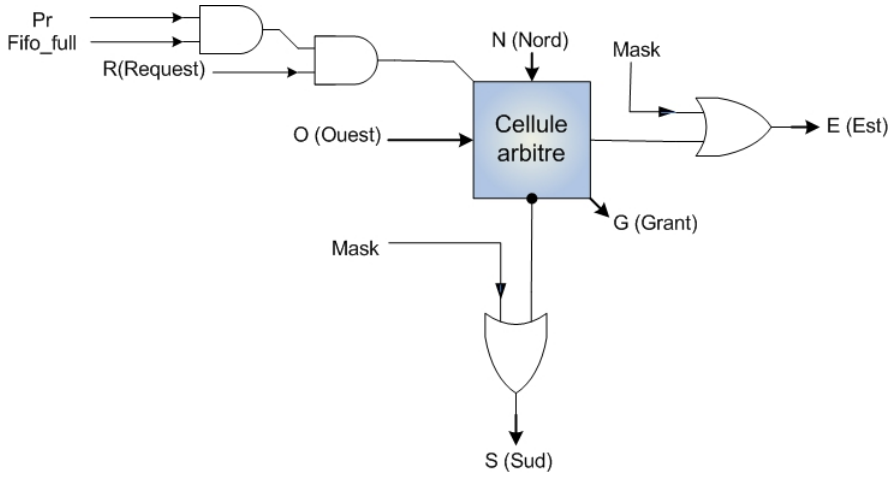



FIGURE 3.12 : Cellule arbitre DPA pour le réseau d'interconnexion

Il possède en entrée trois principaux vecteurs qui sont :

- *Request*, d'une taille de 16 bits : chaque bit de ce vecteur (en provenance des modules gestionnaire des ports d'entrée) représente une demande de transmission d'une donnée d'un port source $i$ vers un port destination $j$.
- *Fifo_full* d'une longueur de 4 bits : ce vecteur représente l'état des Fifo des ports de sorties. Ce signal évite que les données ne soient envoyées vers un port de sortie dont la Fifo est pleine ce qui entraînerait une perte de données.
- *Prio_rot_en* d'une taille de 4 bits qui indique à l'ordonnanceur que des données sont en train d'être émises et qu'il ne doit pas effectuer de rotation de priorité.

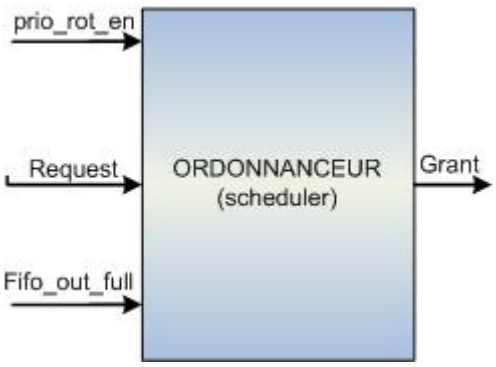



FIGURE 3.13 : L'ordonnanceur

La cellule de l'arbitre de l'ordonnanceur a été modifiée pour inclure le vecteur *Fifo_full* afin de ne valider un transfert que si la Fifo de destination n'est pas pleine. La structure des

cellules arbitres de l'ordonnanceur est donnée à la figure 3.12

#### 3.3.3.1 Gestionnaire du port de sortie

Chaque port de sortie comprend un gestionnaire de sortie construit autour d'une Fifo (voir figure 3.14). Ces modules sont chargés de stocker les données destinées aux ports de sortie, de signaler à l'hôte connecté à la sortie du port qu'une donnée est disponible et de signaler à l'ordonnanceur lorsque la Fifo d'un port de sortie est pleine afin que ce dernier n'y oriente plus de données. Le module de sortie visible sur la figure 3.14 déclenche un timer dès qu'un paquet est présent. Au bout d'un délai, le paquet qui n'est pas relevé est automatiquement détruit.

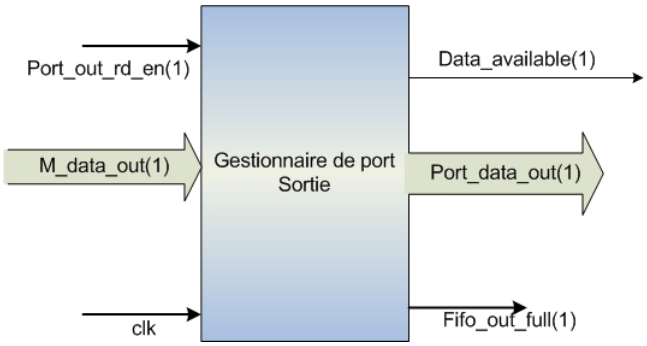



FIGURE 3.14 : Gestionnaire du port de sortie n°1

Les signaux d'interface du module de sortie sont :

- *m_ data_ out* : bus de données (8 bits) en provenance du *crossbar* qui alimente la Fifo de sortie ;
- *port_ out_ data* : données de sortie (8 bits) vers le nœud de destination ;
- *data_ available* : signale la disponibilité d'une donnée dans la Fifo de sortie ;
- *port_ out_ rd_ en* : signal de lecture de la Fifo de sortie par le nœud de destination ;
- *Fifo_ out_ full* : signal qui passe à 1 lorsque la Fifo de sortie est pleine, ce qui force l'ordonnanceur du *crossbar* à retirer la priorité au port émetteur des données.

### 3.3.4 Résultats de mise en œuvre de la couche d'interconnexion

Nous avons mis en œuvre ce réseau dans une ARD de type FPGA de la série ARTIX-7 du fabricant Xilinx.

Les résultats affichés sur le tableau 3.1 donnent les ressources logiques utilisées par le composant d'interconnexion lorsqu'il est synthétisé avec l'outil XST (Xilinx Synthesis Tool) pour différentes valeurs de ports d'E/S du réseau. Les paramètres du synthétiseur ont été réglés pour optimiser la vitesse d'exécution.

Le tableau permet d'estimer le débit maximal que l'on peut obtenir avec la couche d'interconnexion en fonction du nombre de ports qu'elle aura. Si on considère la fréquence donnée

Tableau 3.1: Ressources utilisées par les versions 4 à 16 ports du composant d'interconnexion sur un FPGA Artix-7 xc7a100t

| Ports E/S | Registres | | LUTs | | BRAM | | Freq (MHz) |
|---|---|---|---|---|---|---|---|
| 4 | 1777 | 1 % | 3559 | 6 % | 5 | 4 % | 154 |
| 5 | 1965 | 2 % | 3993 | 6 % | 6 | 4 % | 165 |
| 6 | 2209 | 2 % | 4625 | 7 % | 7 | 5 % | 136 |
| 7 | 2484 | 2 % | 5244 | 8 % | 8 | 6 % | 117 |
| 8 | 2801 | 2 % | 6086 | 10 % | 9 | 7 % | 94 |
| 9 | 3044 | 2 % | 6710 | 11 % | 10 | 7 % | 89 |
| 10 | 3341 | 3 % | 7403 | 12 % | 11 | 8 % | 81 |
| 11 | 3635 | 3 % | 7902 | 12 % | 12 | 9 % | 74 |
| 12 | 3947 | 3 % | 8661 | 14 % | 13 | 10 % | 70 |
| 13 | 4269 | 3 % | 9373 | 15 % | 14 | 10 % | 65 |
| 14 | 4585 | 4 % | 10317 | 16 % | 15 | 11 % | 60 |
| 15 | 4906 | 4 % | 11005 | 17 % | 16 | 12 % | 57 |
| 16 | 5217 | 4 % | 11950 | 19 % | 17 | 13 % | 52 |

pour quatre ports E/S dans le tableau 3.1 et le fait qu'il est possible que quatre transferts indépendants se fassent simultanément, nous calculons une bande passante maximale de 4,93 Gb/s ($4 ports * 8 bits * 154 * 10^6 Hz$) tandis que pour la version 16 ports, la bande passante calculée à la fréquence maximale est de 6,65 Gb/s ($16 * 8 * 52 * 10^6 Hz$) Dans [87] les auteurs du projets ReNoC annoncent une bande passante de 2,4 Gb/s à 100 MHz pour leur réseau reconfigurable 32 bits de type grille. Ces résultats sont corrects lorsqu'on les comparent à ceux obtenus par les concepteurs du projet ReNoC. Nous pouvons toutefois remarquer que la fréquence maximale de fonctionnement est progressivement réduite lorsque le nombre de ports augmente ceci est dû à l'architecture de l'arbitre DPA (voir 3.11) qui utilise une propagation de retenue.

Le graphique 3.15 montre que les ressources croissent en fonction du nombre de ports avec un facteur d'environ 3 tandis que la fréquence chute dans les mêmes proportions entre la synthèse avec 4 ports et la synthèse avec 16 ports. Il est peut-être envisageable de réduire le délai de propagation de la retenue dans l'ordonnanceur DPA en s'inspirant des optimisations réalisées dans le cadre des additionneurs [89].

### 3.3.5 Conclusion

Nous avons, dans cette section, présenté le choix puis la réalisation du réseau d'interconnexion de la plateforme MATIP. En première approche, nous avons opté pour un réseau basé sur un *crossbar* en raison de la densité de communications qu'il autorise, de sa simpli-

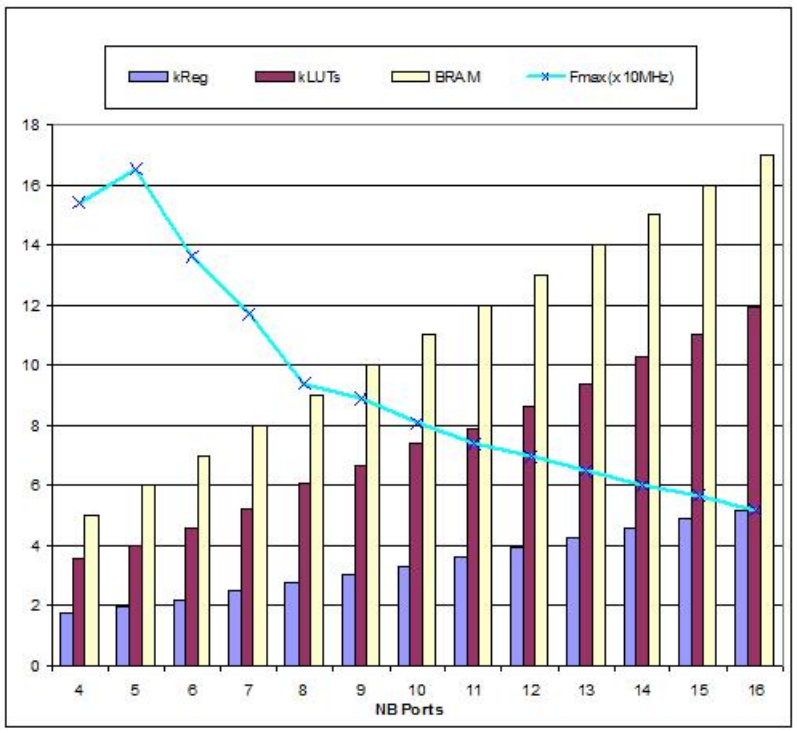



FIGURE 3.15 : Performance de la couche d'interconnexion en fonction du nombre de ports.

cité de mise en œuvre et de sa capacité de reconfiguration. L'inconvénient du *crossbar* est qu'il consomme un nombre de ressources proportionnelles au carré du nombre de ports ; ceci limitera de facto le nombre de nœuds à quelques dizaines lors de la synthèse du réseau d'interconnexion. Il sera nécessaire dans un second temps d'étudier la possibilité d'une architecture qui passe mieux l'échelle tout en conservant les propriétés intéressantes du *crossbar*.

Disposant à présent d'une couche d'interconnexion, support physique des communications, nous présentons dans la prochaine section la conception de la couche de communication de MATIP.

## 3.4 Couche de communication : Le composant MPI-HCL

La couche de communication établie le mode des communications entre les tâches et permet une mise en œuvre matérielle des primitives de transferts habituellement rencontrées dans l'environnement logiciel de programmation parallèle en utilisant MPI. L'objectif de cette couche de communication est d'offrir aux tâches matérielles des primitives de type *put* et *get* pour échanger les messages suivant la norme MPI-2 RMA et de façon transparente vis-à-vis de l'interconnexion physique.

### 3.4.1 Principe de la couche de communication

Nous avons conçu un composant qui recrée une version matérielle de l'environnement de communication MPI-2 et qui exploite le réseau d'interconnexion que nous avons réalisé et décrit à la section 3.3. Nous avons baptisé ce composant : MPI-HCL (**Messsage Passing Interface Hardware Communication Layer**).La figure 3.16 illustre le principe de fonctionnement de la couche de communication.

MPI-HCL intègre un sous-ensemble de primitives du standard MPI-2 RMA (Remote Memory Access) qui sont nécessaires pour exécuter des communications non bloquantes à synchronisation unilatérale. Les communications non-bloquantes sollicitent moins les tâches matérielles par rapport aux communications bloquantes ; une tâche peut recevoir un message alors qu'elle est occupée à d'autres traitements ou envoyer un message à une autre tâche, alors que celle-ci n'est pas immédiatement disponible pour le traiter. Ces communications exigent qu'un espace de stockage soit réservé à l'avance pour recevoir les messages éventuellement attendus.

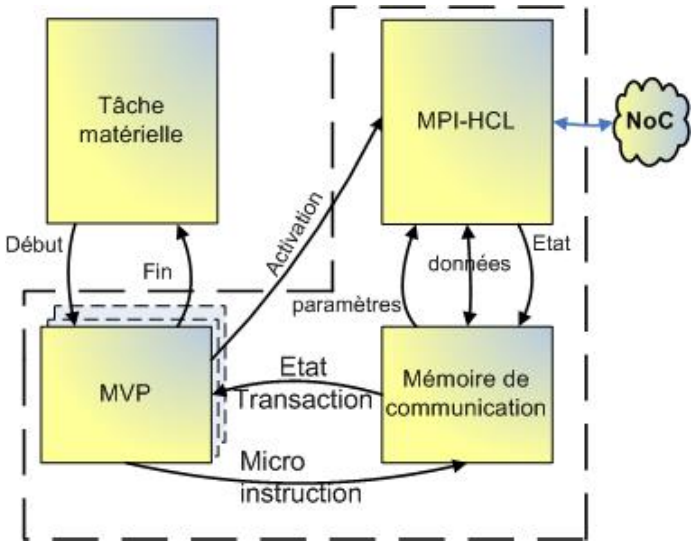



FIGURE 3.16 : Couche de communication

### 3.4.2 Rôle des primitives MPI codées en Vhdl

La tâche matérielle de la couche application utilise des modules décrits en Vhdl et invoqués sous forme de procédures pour réaliser les communications. Nous avons nommé ces modules *MPI Vhdl Primitives* (**MVP**). Lorsqu'une MVP est appelée, elle copie dans la mémoire de communication (voir figure 3.16) tous les paramètres nécessaires à l'exécution de la primitive de communication puis active le composant MPI-HCL qui exécute réellement les différentes opérations nécessaires pour réaliser la primitive demandée.

Les principaux paramètres des primitives de communication sont :

– code instruction de la primitive,

– rang MPI du destinataire,
– adresse source,
– adresse destination.

### 3.4.3 Description des modules de MPI-HCL

#### 3.4.3.1 Modèle logique du composant de communication MPI-HCL

Le composant de communication MPI-HCL est un processeur qui exécute les primitives MPI fournies sous forme de micro-instructions. Le modèle processeur a été retenu car il apporte une interface homogène à la tâche matérielle et permet de faire évoluer le composant par ajout d'instructions lorsqu'il faut mettre en œuvre de nouvelles primitives MPI. Ce processeur est logiquement composé de deux parties : l'une pour exécuter les primitives sollicitées par la tâche matérielle locale, appelée `Core 1` et l'autre pour traiter les primitives initiées par les tâches matérielles distantes à travers le réseau d'interconnexion, cette partie est appelée `Core 2` (voir figure 3.17).

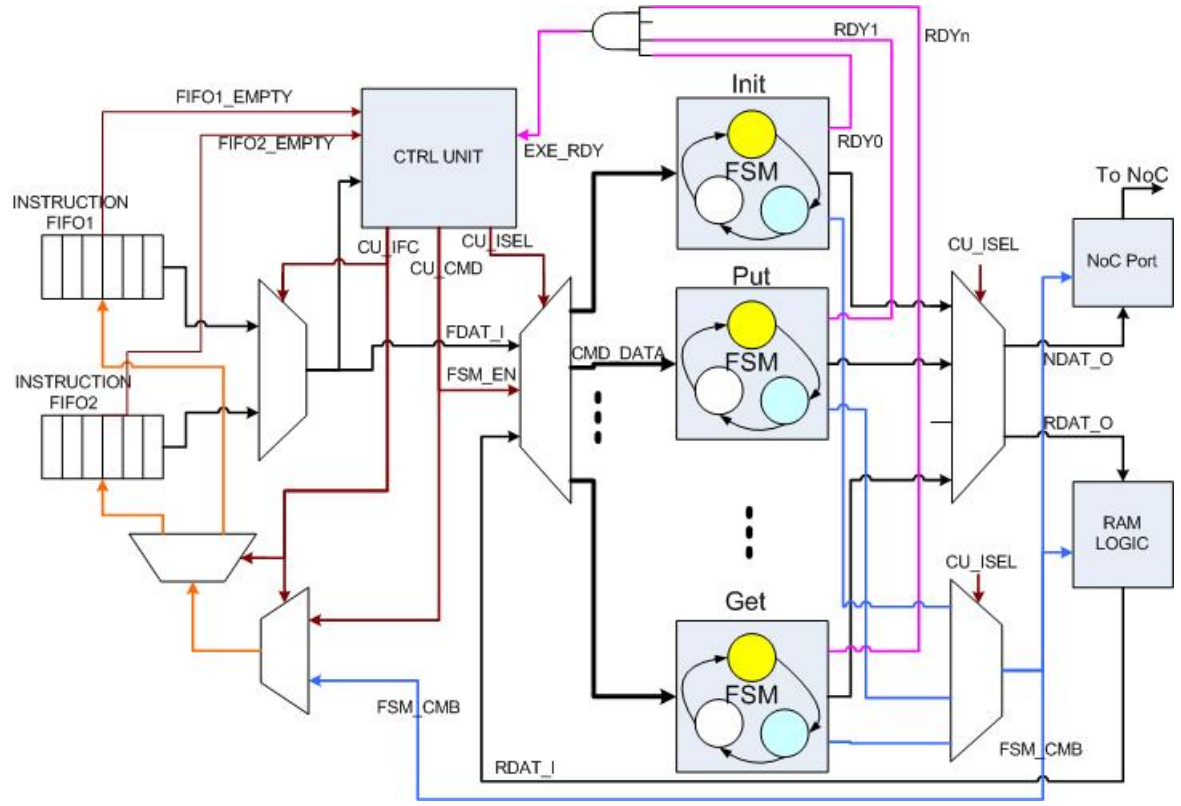
Chaque instruction est composée d'un code instruction et des données additionnelles. La longueur d'une instruction est variable. Le premier octet identifie le code de l'instruction et la cible du transfert s'il y a lieu. Les octets suivants constituent les paramètres de l'instruction (voir tableau 3.2).

Tableau 3.2: Format des instructions de MPI-HCL

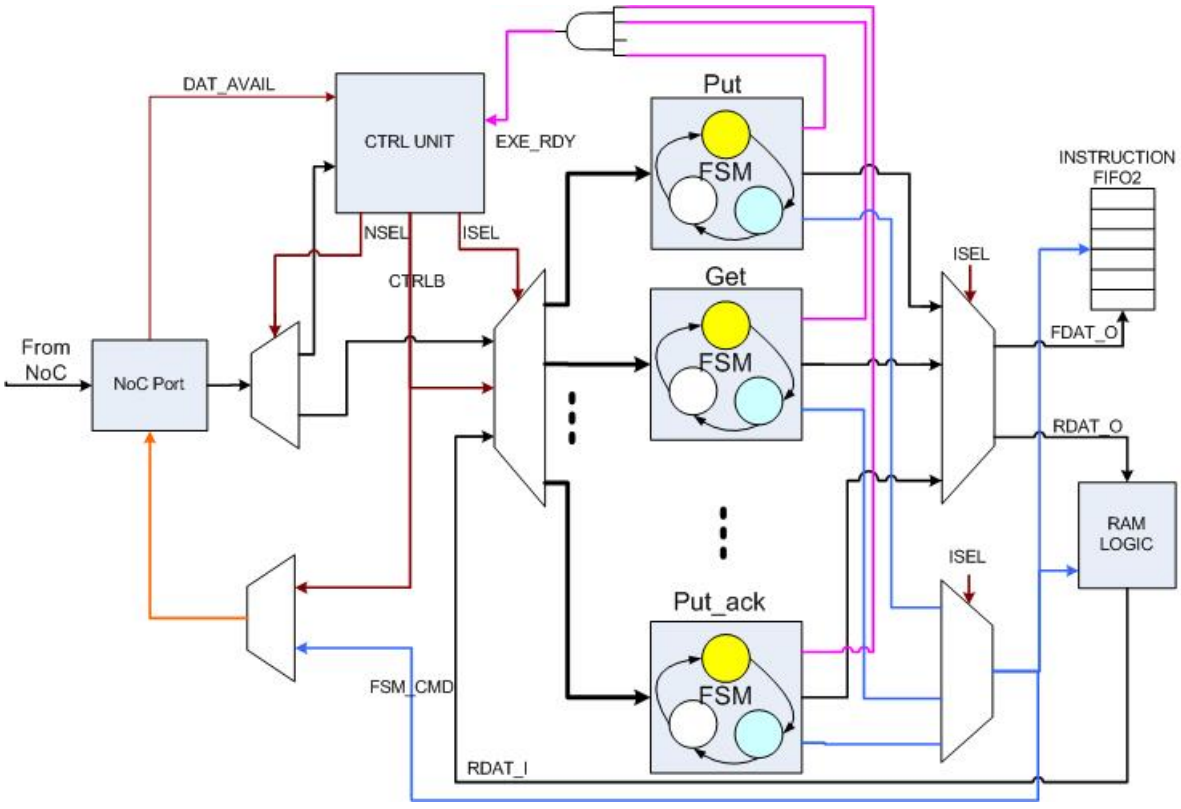
| octet 1 | | octet 2 | octet 3 | | octet n |
|---|---|---|---|---|---|
| Code instruction | destination | taille | P1 | ... | Pn-2 |
| 7 0 | | 7 0 | 7 0 | | 7 0 |

L'unité de contrôle du Core 1 `CTRL UNIT` lit l'instruction à partir de la FIFO `INSTRUCTION FIFOn`, la décode, puis active la machine à états (`FSM`) chargée de traiter l'instruction. Une seule `FSM` peut être active à la fois. Lorsque la `FSM` s'active, elle récupère les autres paramètres de la primitive dans la FIFO `INSTRUCTION FIFOn` puis exécute les actions associées. Chaque `FSM` signale la fin de son traitement à l'unité de contrôle pour permettre le décodage d'une prochaine instruction. Pour le Core 1, les actions d'une `FSM` peuvent consister à lire, écrire en mémoire de communication ou à envoyer des données à travers le réseau.

L'unité de contrôle du Core 2 charge les instructions à partir du port de sortie de la couche d'interconnexion et procède à leur traitement de façon analogue au Core 1. Pour le Core 2, les actions d'une `FSM` peuvent conduire à lire le port d'entrée du réseau, lire ou écrire en mémoire de communication, écrire dans la FIFO `INSTRUCTION FIFO2`.




(a) Core 1




(b) Core 2

FIGURE 3.17 : Modèle logique du processeur MPI-HCL. a) le Core 1 exécute les primitives locales. b) Le Core 2 exécute les primitives distantes.

#### 3.4.3.2 Modèle physique du composant MPI-HCL

Le composant MPI-HCL est physiquement organisé autour de quatre modules qui fonctionnent en parallèle pour traiter les instructions de communications. MPI-HCL utilise une mémoire de communication externe pour stocker ses registres d'état et les données de communication. La mémoire de communication est partagée entre la tâche matérielle et le composant MPI-HCL. Les quatre modules de MPI-HCL accèdent à tour de rôle à la mémoire de communication à travers un gestionnaire DMA (Direct Memory Access) à priorité *round robin* qui assure par principe une égalité d'accès à la mémoire à chaque module.

La figure 3.18 présente l'ensemble des modules du composant MPI-HCL. Chaque tâche matérielle est connectée à travers une interface ayant deux groupes de signaux : les signaux pour le contrôle et l'état de la RAM et les signaux pour le contrôle et l'état des communications. Le composant est directement connecté à la couche d'interconnexion à travers un port. Chaque tâche matérielle est reliée à un composant MPI-HCL. Les modules qui constituent le composant MPI-HCL sont : EX1_FSM qui renferme le Core 1, EX2_FSM qui renferme le Core 2, EX4_FSM, LOAD_INSTR_FSM, DMA_CTRL, INSTRUCTION_FIFO1, INSTRUCTION_FIFO2, NoC I/O Port. Ces modules sont décrits dans la suite.

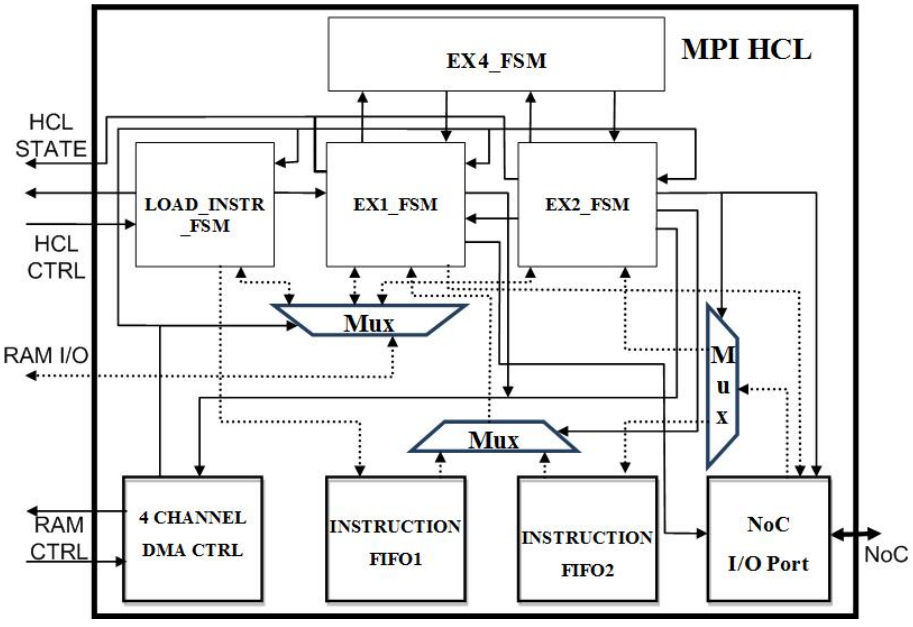



FIGURE 3.18 : Architecture interne du composant MPI-HCL. Les chemins de données principaux sont en traits interrompus et les chemins de contrôle en traits continus. Ce composant s'interface d'une part avec une tâche matérielle et d'autre part avec un port du NoC.

**EX1_FSM** : module d'exécution qui renferme l'essentiel du core 1 du composant MPI-HCL. Il exécute toutes les instructions, réalise les communications sortantes du nœud vers le réseau et met à jour les registres d'état des transactions [9]. En entrée, il possède deux Fifo,

9. *Win_source_status* et *Win_dest_status* qui sont présentés dans la section 3.4.6.3

qui contiennent les instructions à exécuter. Ces instructions proviennent soit de la tâche matérielle locale (INSTRUCTION_FIFO1), soit d'une tâche matérielle distante à travers le réseau (INSTRUCTION_FIFO2). La structure du module Ex1_FSM est donnée à la figure 3.18.

**EX4_FSM** : module d'exécution qui regroupe les process FSM qui réalisent l'initialisation des communications et de l'activation des nouvelles tâches.

**EX2_FSM** : module d'exécution qui renferme l'essentiel du core 2. Il est chargé de gérer les trames en provenance du réseau. Lorsqu'il reçoit une trame, il analyse son en-tête et suivant le type de trame, il peut soit empiler une instruction dans le module INSTRUCTION_FIFO2 (cas des paquets de MPI_GET) soit effectuer une requête DMA pour stocker les données reçues dans la mémoire de communication de la tâche matérielle (cas de la primitive MPI_PUT). Il met à jour les registres d'état des transactions *Win_source_status* et *Win_dest_status*.

**DMA CTRL** : Contrôleur chargé de gérer les accès à la RAM double port de communication pour les modules du composant MPI-HCL. Lorsqu'un module veut lire ou écrire une donnée en RAM, il effectue une demande au contrôleur qui, à son tour, effectue une requête vers la tâche matérielle. Si la tâche matérielle n'occupe pas la mémoire, elle répond favorablement à la requête. Le contrôleur DMA gère les accès à la mémoire entre les modules EX1_FSM, EX2_FSM, EX4_FSM, et LOAD_ INSTR_FSM en utilisant un ordonnanceur de type *round robin* pour satisfaire les requêtes d'accès à la mémoire. L'arbitrage dit *round robin* est utilisé ici car l'ordre d'arrivée de la requête détermine la priorité de l'accès et il permet de satisfaire les requêtes à tour de rôle.

**INSTRUCTION_FIFOn** : Les mémoires tampons d'instructions servent à stocker les instructions qui seront exécutées par le module EX1_FSM. Les instructions produites localement sont stockées dans le module INSTRUCTION_FIFO1. Les instructions générées par le module EX2_FSM, sont stockées dans le module INSTRUCTION_FIFO2.

**LOAD_ INSTR_FSM** : module de chargement des instructions qui copie les micro instructions compilées par les MVP de la mémoire de communication vers la Fifo INSTRUCTION_FIFO1.

**NoC I/O Port** : réalise l'interface de connexion à la couche d'interconnexion. Il est chargé d'envoyer les trames entrantes vers le module Ex2_FSM et d'expédier vers le réseau les trames issues du module EX1_FSM.

### 3.4.4 La mémoire de communication

Afin de mettre en œuvre MPI-2 RMA, il est nécessaire d'intégrer une mémoire de stockage au sein de la couche de communication. Cette mémoire est partagée entre la tâche matérielle et le composant MPI-HCL. Elle est structurée en trois zones (voir figure 3.19) :

- la zone des registres d'état, de configuration et de synchronisation ;

– la zone de stockage des données à envoyer ou à recevoir ;
– la zone de transfert des instructions.

Les registres d'état, de configuration et de synchronisation (voir table 3.3) ont été créés pour servir d'interface entre la tâche matérielle et le composant de communication d'une part, et pour synchroniser les transactions entre les tâches d'autre part.

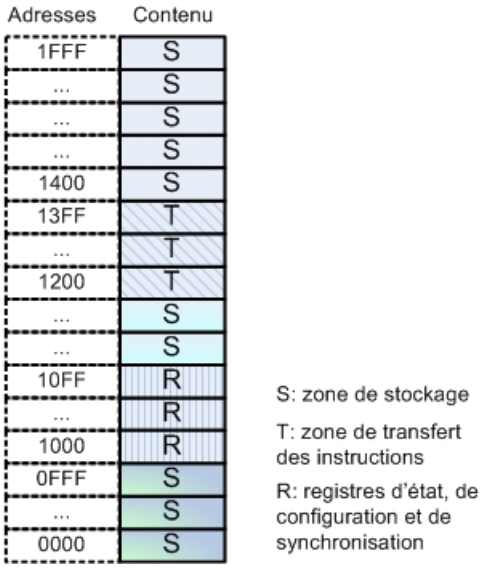



FIGURE 3.19 : Organisation de la mémoire de communication du composant MPI-HCL. Plages d'adresses des zones de stockage des données (S), d'accès aux registres (R), et transfert des instructions et leurs paramètres vers le composant MPI-HCL (T)

La zone R des registres MPI est dédiée au stockage des paramètres MPI pendant l'utilisation du composant MPI-HCL. Elle comprend les registres de la table 3.3, que nous présentons en détail dans la suite.

#### 3.4.4.1 Zone registre R

Le registre *status* :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Reservé | Reservé | `Spawned` | `Initialized` | Reservé | Reservé | Reservé | `InitAck` |

Le registre *status* contient trois indicateurs binaires : `Initialized` pour signaler lorsque la tâche matérielle est initialisée ; `Spawned` pour signaler si la tâche matérielle est dynamique ; `InitAck` pour signaler la fin de la primitive d'intialisation dans MPI-HCL.

Le registre *Config* :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Reservé | Reservé | Reservé | Reservé | Reservé | Reservé | Reservé | `Ipulse` |

Le registre *config* permet de déclencher la prise en compte d'une instruction en mettant à 1 le bit `Ipulse`.

Les registres *Iadr0* et *Iadr1* stockent une valeur sur 16 bits, cette valeur correspond à l'adresse d'un bloc d'instructions dans la zone de codage et de transfert. Ces registres sont utilisés par la MVP pour écrire l'adresse de l'instruction à transférer au composant MPI-HCL. Le module LOAD_INSTR du composant MPI-HCL consulte cette adresse pour copier

Tableau 3.3: registres du composant MPI-HCL

| **Nom** | **position** | **Description** |
|---|---|---|
| status | 00h | Registre d'état |
| config | 01h | Registre de configuration |
| IAdr0 | 02h | Adresse basse de l'instruction |
| Iadr1 | 03h | Adresse haute de l'instruction |
| Win0 | 04h | Adresse de la fenêtre 0 (Mpi_create_win utilise cette adresse pour stocker l'objet) |
| Win1 | 14h | Adresse de la fenêtre 1 |
| Win2 | 24h | Adresse de la fenêtre 2 |
| Win3 | 34h | Adresse de la fenêtre 3 |
| Grp0 | 44h | Adresse du groupe 0 (16 membres par groupe) |
| Grp1 | 46h | Adresse du groupe 1 |
| Grp2 | 48h | Adresse du groupe 2 |
| Grp3 | 50h | Adresse du groupe 3 |
| Comm0 | 52h | Données de COMM_WORLD le communicateur par défaut |
| Comm1 | 54h | Données pour les communicateurs additionnels |
| Comm2 | 56h | Données pour les communicateurs additionnels |
| Comm3 | 58h | Données pour les communicateurs additionnels |
| RankToPort | 60h | Table pour translater les rangs MPI en port réseau |
| MemHeap | 70h | Adresse de la mémoire de la bibliothèque |
| MemSize | 72h | Taille de la mémoire de la bibliothèque |
| HeapPtr | 74h | Pointeur de la mémoire de la bibliothèque |

l'instruction de la zone de codage et de transfert vers le tampon d'instructions INSTRUCTION_FIFO1, lorsque le bit 0 (IPulse) du registre *config* passe à 1.

Le registre *RankToPort* d'une taille de 16 bits pointe le début d'une table de correspondance qui fournit le port du NoC à partir du Rang MPI. Ex. l'adresse *RankToPort+1 (soit 61h)* contient le numéro du port réseau de la deuxième tâche matérielle. La table contient 16 entrées de 8 bits dans cette version de MPI-HCL.

Les registres *MemHeap*, *HeapPtr* sont utilisés par la MVP MPI_Alloc_mem pour gérer les blocs mémoires alloués dans la mémoire de communication.

Les registres *MemSize* et *Memsize+1* forment un mot sur 16 bits pour donner la taille de la mémoire de communication. Cette information est utilisée par la primitive MPI_Alloc_mem() lors de l'allocation de mémoire.

Les registres *Winx*, *Grpx* et *Commx* sont prévus pour gérer les fenêtres, les groupes et les communicateurs. Ils ne sont pas utilisés dans la première version du composant MPI-HCL.

#### 3.4.4.2 La zone de codage et de transfert T

La zone de codage et de transfert (T) des primitives MPI est utilisée pour stocker les paramètres d'appel et les paramètres de résultats lors de l'exécution des MVP. Cette zone est organisée en bloc de 10 octets pour chaque primitive soit environ 6 octets pour les paramètres d'appel et 4 octets pour les valeurs de retour.

Le tableau 3.4 donne la liste des instructions réalisées et leur code instruction correspondant pour le composant MPI-HCL. Pour coder une instruction dans la zone de codage et de transfert, la MVP écrit le code instruction correspondant à chaque primitive et les paramètres propres à cette primitive. L'instruction `HCL_ack` n'est pas codée par une MVP mais elle est utilisée en interne par MPI_HCL pour acquitter les messages reçus lors de la synchronisation des transferts comme présentée dans la section 3.4.6.4.

### 3.4.5 Fonctionnement du composant MPI-HCL

Le composant MPI-HCL est activé par les modules MVP. Après avoir codé la micro instruction dans la mémoire de communication, la MVP active le composant MPI-HCL. La première tâche du composant MPI-HCL est de transférer l'instruction dans la Fifo `instruction FIFO 1` du Core 1 en vue de l'exécution de la primitive de communication associée à la MVP. A la fin de l'exécution, le composant MPI-HCL écrit les résultats d'exécution directement dans la mémoire de communication.

#### 3.4.5.1 L'initialisation des communications avec MPI-HCL

Avant toute communication, il est indispensable que la couche de communication reçoive un rang qui permettra d'identifier la tâche matérielle qui lui est associée dans l'environnement MPI. Lors de l'initialisation, chaque tâche matérielle utilise la MVP `MPI_INIT` qui construit

Tableau 3.4: liste des instructions traitées par MPI-HCL et leur code d'instruction

| **code instruction** | | | | **primitive MPI** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | MPI_rank() |
| 0 | 0 | 0 | 1 | MPI_size() |
| 0 | 0 | 1 | 0 | MPI_Barrier() |
| 0 | 0 | 1 | 1 | MPI_Barrier_reached() |
| 0 | 1 | 0 | 0 | HCL_Ack()* |
| 0 | 1 | 0 | 1 | MPI_Put() |
| 0 | 1 | 1 | 0 | MPI_Get() |
| 0 | 1 | 1 | 1 | MPI_Bcast() |
| 1 | 0 | 0 | 0 | MPI_Init() |
| 1 | 0 | 0 | 1 | MPI_Spawn() |
| 1 | 0 | 1 | 0 | MPI_Finalize() |
| 1 | 0 | 1 | 1 | MPI_Win_Start() |
| 1 | 1 | 0 | 0 | MPI_Win_Completed() |
| 1 | 1 | 0 | 1 | MPI_Win_Post() |
| 1 | 1 | 1 | 0 | MPI_Win_Wait() |

*`Messages d'acquittement propre aux composants MPI-HCL`

et transfère l'instruction `INIT` au composant MPI-HCL. Les composants MPI-HCL vont élire celui d'entre eux qui sera considéré comme la référence pour les autres composants en ce qui concerne les fonctions collectives. L'élection du composant de reférence est basée sur l'algorithme suivant :

1. Identifier le numéro du port du réseau.
2. Si le numéro du port du réseau sur lequel est connecté la tâche matérielle est 0 alors le composant est désigné composant de référence. Il envoie le message `INIT_SETRANK` aux autres composants qui ont émis le message `INIT_SEEKMAIN`
3. Si le numéro du port est différent de 0 alors le composant construit le message `INIT_SEEKMAIN`. Dès lors il attend de recevoir le rang qui lui sera affecté par le composant de référence pour terminer son initialisation.

Après l'élection, tout composant MPI-HCL qui envoie le message `INIT_SEEKMAIN`, reçoit le message `INIT_SETRANK`, en provenance du composant de référence. Ce message contient le rang MPI du composant ayant effectué la requête. Le rang affecté au composant MPI-HCL est utilisé par la tâche matérielle pour s'identifier dans l'environnement de communication MPI.

Chaque composant MPI-HCL initialisé gère un banc de registres qui contient la correspondance entre le rang MPI et le numéro de port du réseau de tous les autres composants

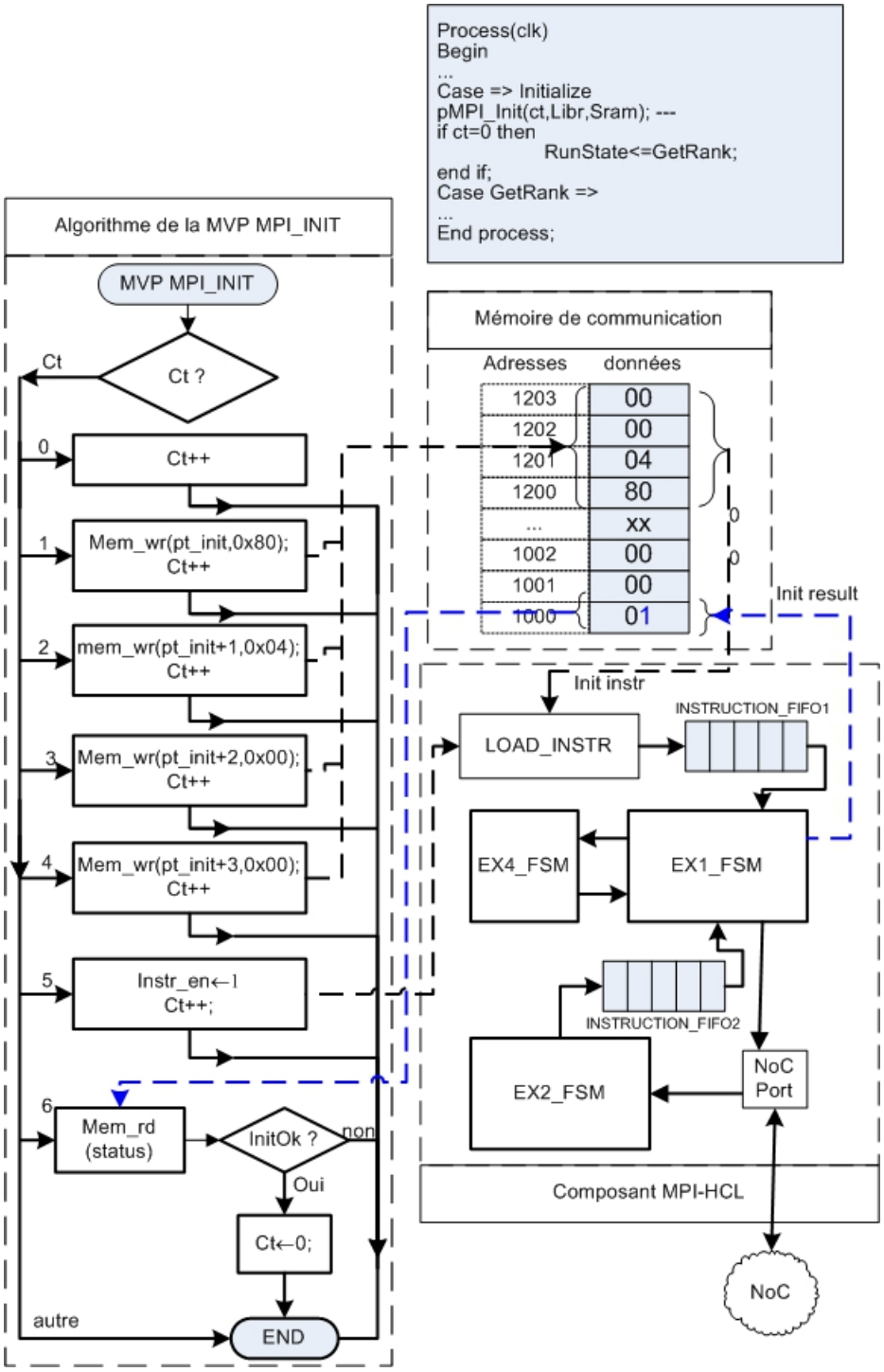



FIGURE 3.20 : Illustration du fonctionnement de la MVP MPI_INIT.

MPI-HCL qui sont initialisés ; l'adresse d'accès au banc de registres est le rang MPI et la donnée extraite du banc de registre est le numéro du port réseau. Cette table de stockage est exploitée localement pour exécuter les primitives `MPI_GET_RANK()` et `MPI_GET_SIZE()`.

Tant que la tâche matérielle hôte n'a pas initialisé le composant MPI-HCL, aucune communication n'est possible à travers le réseau ; les composants non initialisés ignorent tous les messages en provenance du réseau.

#### 3.4.5.2 Utilisation des modules MVP

Au début de son exécution, la MVP écrit les informations de l'instruction dans une zone de la mémoire de communication qui est dédiée à cet effet. Pour la MVP *MPI_INIT*, la plage d'adresse que nous avons réservée est 0x1200-0x1203. Cette plage d'adresse est fixée par une constante (*CORE_INIT_ADR*) du template que nous avons réalisé.

Après le rangement en mémoire des données de l'instruction à exécuter, la MVP met à 1 le drapeau *Instr_en* du registre `config` pour indiquer au module LOAD_INSTR qu'une instruction est prête à être exécutée. Le module LOAD_INSTR copie l'instruction dans la mémoire INSTRUCTION_FIFO1 et produit un acquittement. Dès qu'une instruction est présente dans le tampon d'instruction, le module EX1_FSM lit la Fifo et exécute l'instruction. Si c'est une instruction d'initialisation, il fait appel au module EX4_FSM pour la gérer.

Lorsque l'instruction est complètement exécutée, le module EX1_FSM écrit le résultat de l'exécution s'il y a lieu, dans le bloc mémoire de codage et de transfert de l'instruction, à l'adresse dédiée pour les résultats (0x120A pour la MVP *MPI_PUT* ; voir figure 3.22).

La MVP scrute cette adresse de résultat pour déterminer la fin de l'exécution de l'instruction et son résultat éventuel. S'il y a lieu, la MVP va recopier les résultats d'exécution dans les paramètres de retour de la primitive (cas par exemple de MPI_Comm_Rank).

La figure 3.21 illustre le fonctionnement de la MPV MPI_INIT et la figure 3.22 illustre le fonctionnement de la MVP MPI_PUT. Chaque étape de la MVP nécessite au moins un cycle d'horloge, ainsi les MVP ont besoin de plusieurs cycles d'horloge pour s'exécuter.

#### 3.4.5.3 Initialisation des fenêtres mémoire de stockage des données

Une étape importante pour la mise en œuvre de MPI-RMA, consiste à définir les fenêtres de lecture et d'écriture dans la mémoire de communication de la tâche matérielle. Nous avons réalisé une première version de la bibliothèque MPI qui gère une unique fenêtre appelée *Win0* par tâche matérielle et un groupe. Cette fenêtre donne accès à toute la mémoire de communication. La tâche matérielle doit indiquer le rang des tâches concernées par les transactions à travers l'appel à `MPI_WIN_Start()` côté source et `MPI_WIN_Post()` côté cible. Le paramètre fenêtre `MPI_Win` est inopérant. Les primitives `MPI_PUT()` et `MPI_GET()` utilisent des paramètres `srcAdr(adresse source)` et `destAdr(adresse destination)` pour stocker

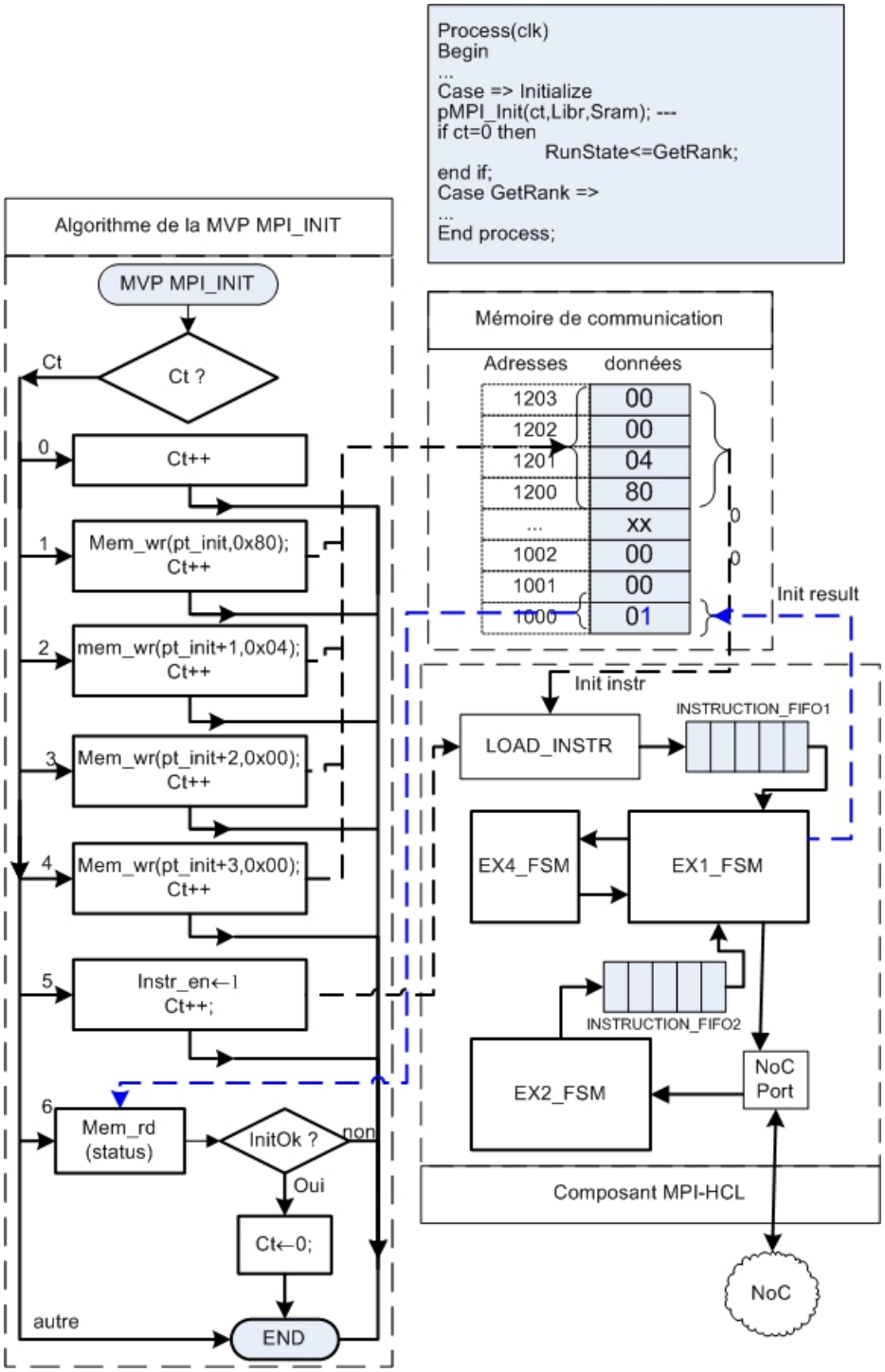



FIGURE 3.21 : Illustration du fonctionnement de la MVP MPI_INIT.

les données dans les zones de stockage (voir figure 3.19) de la mémoire de communication.

#### 3.4.5.4 Émission-réception des données

Pour transmettre ou recevoir des données, la tâche matérielle appelle une MVP, qui place dans la **zone de codage et de transfert** de la mémoire de communication, le code et les paramètres de l'instruction. La fin de l'exécution de l'instruction est signalée à travers des informations dans la zone de transfert de la même mémoire, ou à travers les signaux dédiés du composant MPI-HCL. Chaque tâche matérielle utilise une adresse spécifique dans la zone de codage et de transfert de la mémoire de communication pour loger les paramètres d'appel des primitives MPI et scruter les valeurs de retour après leur exécution.

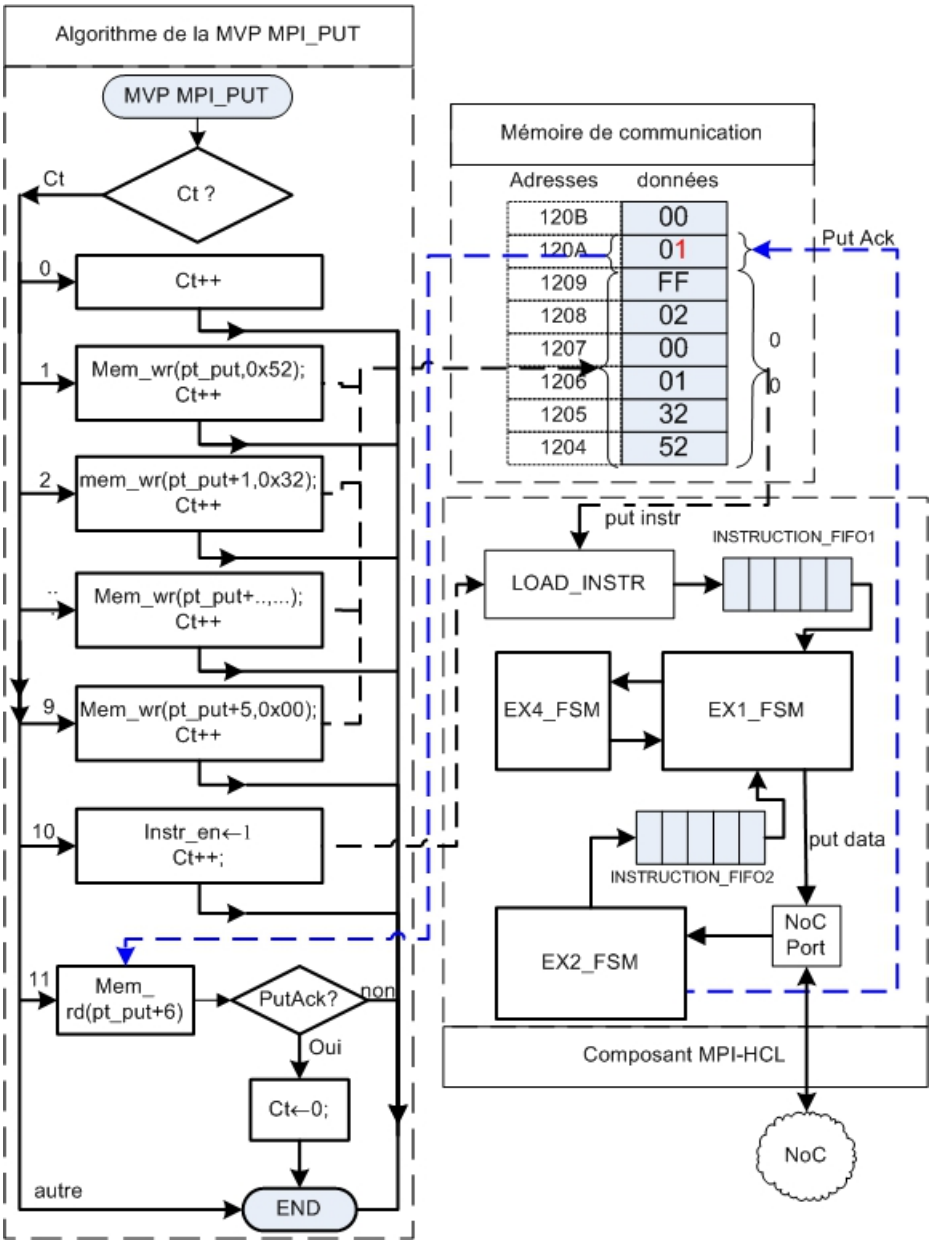



FIGURE 3.22 : Illustration du fonctionnement de la MVP MPI_PUT.

### 3.4.6 La synchronisation des transferts avec MPI-HCL

Lors d'une communication, il est important d'identifier le début et la fin des transferts : c'est le but de la synchronisation. Parmi les trois techniques de synchronisation qui sont envisageables, nous avons réalisé la synchronisation cible active et sans barrière comme dans [90]. Cette synchronisation est la plus flexible lorsque les différentes tâches matérielles sont hétérogènes. Elle génère peu de messages à travers le réseau par rapport à la synchronisation par barrière, et, elle est moins contraignante pour la gestion de la mémoire de communication que la synchronisation cible passive. En effet, d'après la spécification du standard MPI, la synchronisation cible passive nécessiterait la mise en œuvre d'un mécanisme complexe de verrouillage de la mémoire de communication lors des transactions MPI.

#### 3.4.6.1 Mise en œuvre adoptée pour les primitives de synchronisation

Le standard MPI stipule qu'un appel de la primitive `Win_start` sur la source donne lieu à un appel `Win_post` sur la cible. Et qu'un appel de `Win_complete` sur la source doit donner lieu à un appel `Win_Wait` sur la cible. Une première interprétation de cette directive conduit aux choix de mises en œuvre suivantes :

- dès l'appel de `Win_post` sur la cible, envoyer un message à toutes les sources concernées ;
- lors de l'appel de `Win_start` sur la source, vérifier la réception ou attendre le message `Win_post` équivalent avant de se terminer pour permettre aux fonctions de transfert de s'exécuter ;
- lors de l'appel `Win_Complete` sur la source, envoyer un message à chaque cible concernée ;
- lors de l'appel `Win_wait` sur la cible, attendre de recevoir le message `Win_complete` équivalent pour se terminer.

Nous avons adapté ces choix afin de réduire le nombre de messages échangés entre les sources et les cibles :

- lorsque `Win_post` est appelé sur la cible, nous mettons à jour le registre *GPost* de la cible qui indique que des messages sont attendus des membres désignés par leur rang dans le paramètre `groupe` ;
- lorsque `Win_start` est appelé sur la source, nous enregistrons dans le registre *GStart*, la liste des rangs des membres qui feront l'objet de transactions RMA ;
- la réception effective des messages de transfert synchronise le registre *GPost* ;
- lors de l'appel à `Win_Complete`, nous envoyons un message à toutes les cibles qui avaient été impliquées par `Win_start` ;
- l'appel à `Win_wait` sur la cible, vérifie la fin des transferts RMA et attend le message `Win_complete` correspondant pour se terminer.

#### 3.4.6.2 Les différents types de messages entre une source et une cible

Le message désigne une information qui est transmise entre les composants MPI_HCL à travers le réseau. Seul le message `Put` contient des données issues de la tâche matérielle. Tous les autres messages contiennent des instructions de commande (`Init, Get`), de contrôle (`Ack_Put, Ack_Get`) ou de synchronisation (`Win_Compl`).

#### 3.4.6.3 Les registres de gestion de la synchronisation

Nous utilisons quatre registres pour gérer les synchronisations des transactions. Deux registres, *GPOST* côté source et *GSTART* côté cible, contiennent les rangs des tâches impliquées dans les transactions et deux registres *Win_ source_ status* et *Win_ dest_ status* regroupent les indicateurs d'état de la transaction.

Les registres d'état de *Win0* : *Win_ Source_ status* ou *Win_ Dest_ status* ont la même structure et leurs bits ont la même signification.

| registre *Win_ source_ status* (resp. *Win_ Dest_ status*) | | | | | | | |
|---|---|---|---|---|---|---|---|
| WCOMP | WBUSY | DSENT | DRED | WPOST | DSING | DRING | WSTART |

Les bits `DRING` [10] et `DSING` [11] indiquent l'attente des transferts. Les bits `DRED` [12] et `DSENT` [13] indiquent l'exécution de transferts.

Le registre *Win_ Dest_ status* à l'adresse `Core_base_adr+4` est utilisé pour gérer la synchronisation des transactions en cours dans la fenêtre côté cible.

le registre *Win_ Source_ status* à l'adresse `Core_base_adr+5` est utilisé pour gérer la synchronisation des transactions en cours dans la fenêtre côté source.

Lorsque le bit `WPOST` est mis à 1, il signale que la fenêtre mémoire est cible des opérations RMA. La primitive `Win_Wait` devra se terminer si le bit DRED est à 1 ou si `DSING=1 et DSENT=1` et si un message `Win_complete` a été reçu de chaque source potentielle.

Si `WSTART` est à 1, la fenêtre est source des opérations RMA, `MPI_Win_Complete` devra se terminer si `DSENT=1 et DRING=0`.

#### 3.4.6.4 Processus de synchronisation de la primitive `MPI_Put`

L'envoi de données avec synchronisation cible active se déroule en trois phases notées *T1,T2* et *T3* sur la figure 3.23. L'action des MVP et des modules qui participent à ce processus sur chaque indicateur des registres de synchronisation est expliquée ci-après.

10. Data receiving
11. Data Sending
12. Data received
13. Data Sent

**Phase T1**

Côté source, la primitive `MPI_Win_Start` met à 1 le bit WSTART du registre *Win_source_status* et met à 1 les champs du registre *GStart*[14].

Côté cible, la primitive `MPI_Win_Post` met à 1 le champ WPOST du registre *Win_Dest_status*, met tous les autres bits à 0 et met à 1 les champs du registre GPost[15] qui identifient les membres à l'origine des primitives de transferts.

**Phase T2**

Du côté source, le module EX1_FSM transmet le message tandis que la MVP attend un message d'acquittement. Le module EX2_FSM met à 1 le bit DSENT lorsqu'il reçoit l'acquittement.

Du côté cible, le module EX2_FSM reçoit le message et met à 1 le bit DRED. Puis le module EX1_FSM renvoie un message d'accusé de réception.

**Phase T3**

Côté source, la primitive `MPI_Win_complete` provoque l'envoie du message `Win_Complete` à chaque cible qui attend les données. Elle utilise le registre *GStart* pour identifier les cibles concernées. Tous les bits du registre *Win_source_status* et du registre *GStart* sont remis à 0 à la fin de la primitive `MPI_Win_complete`.

Côté cible, lorsque le module EX2_FSM reçoit l'information `Win_Complete`, il met à 1 le bit WCOMP du registre *Win_dest_status*. La primitive `MPI_Win_Wait` se termine si le bit WCOMP est à 1, si WBUSY est à 0.

Lors de l'appel de la MVP `MPI_Put`, une instruction `Put` est construite et rangée dans la Fifo d'instructions du module EX1_FSM. La MVP `MPI_put` se termine dès que le message d'acquittement (Put_Ack) est reçu. Une nouvelle primitive `MPI_Put` peut être exécutée. Lors de l'envoie d'un message `Put` par Ex1_FSM, le bit `DSENT` du registre *Win_Source_status* passe à 0 et dès la réception du message `Put_Ack` `DSENT` passe à 1. Toutefois, les données proprement dites peuvent encore être en cours d'acheminement car le message `Put_Ack` est envoyé par le module EX2_FSM de la cible dès qu'il reçoit l'en-tête du message `Put`. `Put_Ack` peut être reçu par EX2_FSM à la source alors que EX1_FSM est encore en train d'envoyer les données du messages `Put` car les ports du réseau acheminent les données du message au fur et à mesure de leur réception. (voir fig 3.23)

#### 3.4.6.5 Processus de synchronisation de la primitive `MPI_Get`

La figure 3.24 montre la synchronisation pour une requête de communication entraînant la réception d'un message par la source. L'exécution de la primitive `MPI_Get` se déroule en

14. registre côté source qui identifie les membres ciblés par les primitives de transfert RMA
15. registre côté cible qui contient tous les membres sources des primitives de transfert RMA

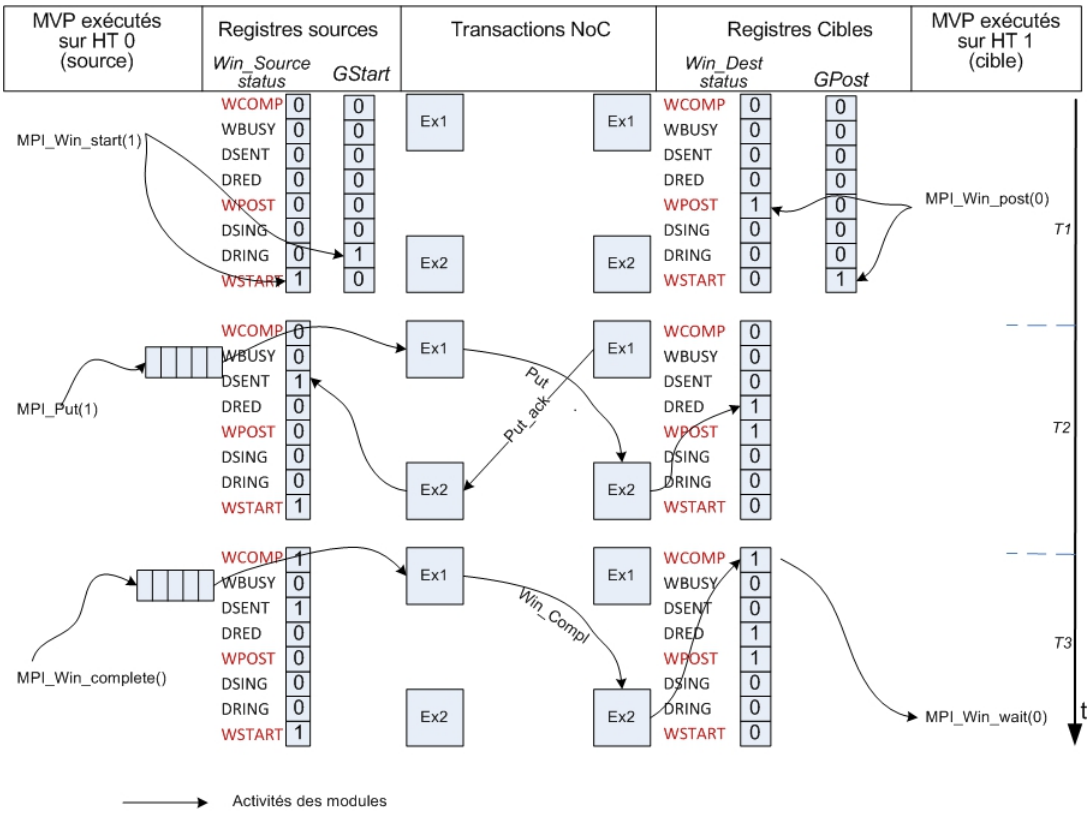



FIGURE 3.23 : Synchronisation lors du processus d'envoi d'un message (put_prg).

trois phases : initialisation (*T1*), transfert(*T2*), achèvement(*T3*) sur la source et sur la cible.

### Phase T1

Côté source, la MVP `MPI_Win_Start` permet d'initialiser le registre *Win_source_status* dont le bit WSTART est mis à 1 et de sélectionner les membres qui seront cibles des transactions (*GStart*=0x2 pour cibler le rang n° 1) ; elle n'envoie aucun message à travers le réseau.

Côté cible, la MVP `MPI_Win_Post` enclenche le processus d'attente d'une transaction RMA en initialisant les registres *Win_dest_status*=0x8 (pour mettre le bit WPOST à 1) et *GPost*=0x1 (pour attendre du rang n° 0).

### Phase T2

Lors de l'appel de la MVP `MPI_Get`, une instruction `Get` est construite et rangée dans la Fifo INSTRUCTION_FIFO1, puis le module EX1_FSM envoie un message `Get` et met à 1 le bit DRING du registre *Win_Source_status* indiquant qu'une réception est en cours (voir fig 3.24). Dès que le module EX2_FSM reçoit le message d'acquittement `Ack_Get`, la MVP `MPI_GET` se termine. Le module EX2_FSM construit un identifiant (ID) formé par concaténation de quelques valeurs (Rang&AdrDest&taille) de l'en-tête du message `Put` attendu et le conserve dans un banc registre interne nommé *IDR*. Un compteur nommé *Cget*

est associé à *IDR* pour indiquer à tout instant le nombre de messages `Get` attendus par la source. Le compteur *Cget* est incrémenté chaque fois qu'un acquittement `Ack_Get` est reçu et le bit `WBUSY` est mis à 1.

Dès qu'un message `Put` est reçu par Ex2_FSM et que son ID correspond à l'un des ID du banc de registres internes *IDR*, le compteur *Cget* est décrémenté et le bit `DRED` est mis à 1 pour indiquer que la réception est terminée. Le bit `WBUSY` est mis à 0 lorsque le compteur *Cget* revient à 0.

La MVP `MPI_GET` ne se termine pas tant que Ex2_FSM n'a pas reçu le message d'acquittement `Ack_Get`. Après un délai sans réponse, elle reconstruit l'instruction `Get` pour une prochaine tentative.

La primitive `MPI_Get` génère une dizaine d'activités entre les modules EX1_FSM et Ex2_FSM de la source et ceux de la cible. Sur la figure 3.24 les numéros à côté des arcs orientés de la phase T2 correspondent aux actions suivantes :

1 la MVP `MPI_Get` génère une instruction et la copie dans le tampon `INSTRUCTION_FIFO1` ;

2 l'instruction est chargée par EX1_FSM pour être exécutée ;

3 un message `Get` est envoyé sur le réseau ;

4 le Bit DSING est activé ;

5 un message d'acquittement `Ack_Get` est renvoyé vers la source ;

6 le Module Ex1_FSM de la source active le bit DRING ;

7 le message `Put` contenant les données est envoyé vers la source ;

8 le bit DRED est activé ;

9 un message d'acquittement `Ack_Put` est envoyé à la cible pour son dernier message `Put` ;

10 le bit DSENT est activé sur la cible marquant la fin de la transaction.

### Phase T3

Côté source, lorsque la MVP `MPI_Win_complete` est exécutée, le module EX1_FSM met à 1 le bit `WCOMP` et envoie le message `Win_complete` à la cible.

Côté cible, lorsque le module EX2_FSM reçoit le message `Win_complete`, il met à 1 le bit `WCOMP` et à 0 le bit `WBUSY` et la MVP `MPI_Win_Wait` détecte alors cette condition pour terminer son exécution. Toutefois, la MVP vérifie qu'au moins une transaction MPI-2 RMA (`MPI_Put` si `DRED=1` ou `MPI_Get si DSING=1 et DSENT=1`) a été exécutée.

Dans la mémoire de communication, des registres réservés au composant MPI-HCL, permettent d'indiquer le statut des transferts en cours dans la fenêtre, afin de gérer les synchronisations `Win_Complete` et `Win_Wait`.

Le module EX2_FSM est chargé de mettre à jour les bits DRED et DSENT des registres *Win_dest_stauts* ou de *Win_source_status* de la fenêtre active.

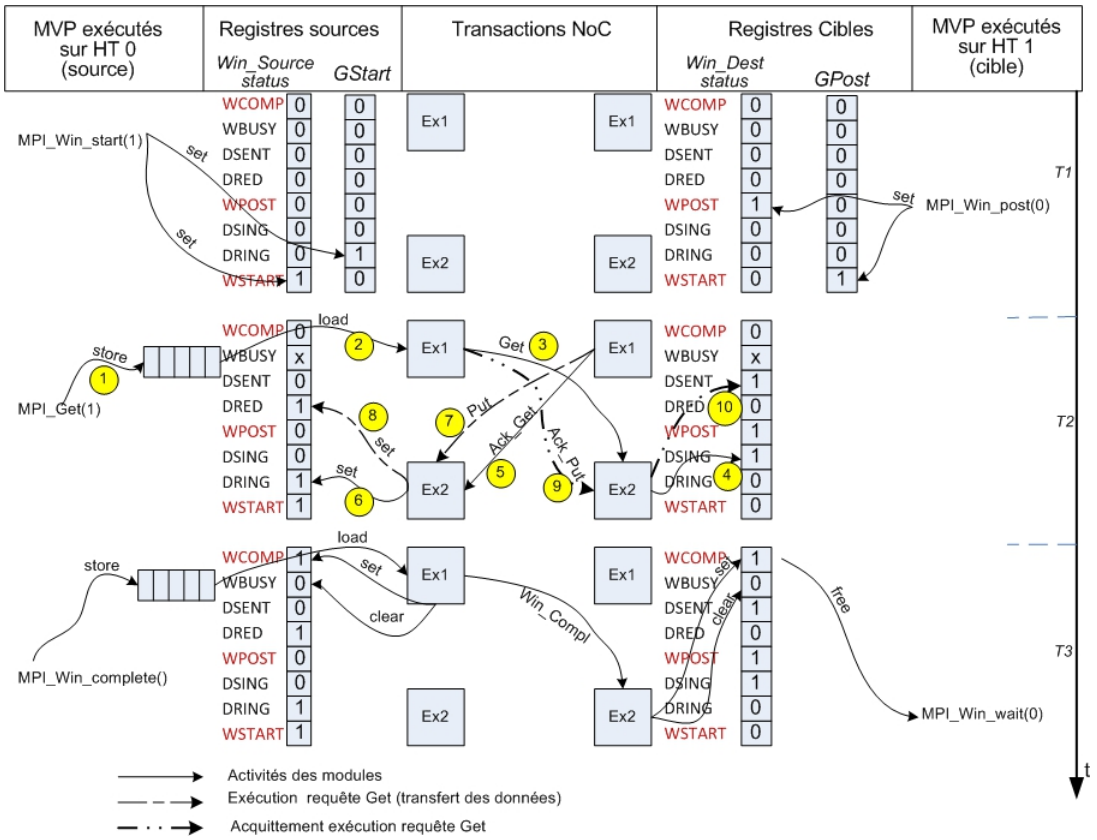



FIGURE 3.24 : Synchronisation lors du processus de réception d'un message (Get_prg).

Le module EX2_FSM indique au module EX1_FSM d'envoyer un message d'acquittement chaque fois qu'il reçoit une instruction de transfert (`MPI_PUT, MPI_GET,`...).

### 3.4.7 Finalisation des transferts

A la fin de tous les transferts RMA, la seule primitive de synchronisation qui envoie un message à travers le réseau est `MPI_Win_Complete`, elle envoie un message de synchronisation à toutes les cibles des opérations RMA qui ont eu pour origine la source courante. La fin des opérations `Put` et `Get` ne peut être signalée que par l'envoie explicite de ce message de synchronisation car la cible ne peut pas savoir le nombre d'opérations qui seront effectuées dans la fenêtre à l'avance. Ainsi `Win_Post` sur la cible va attendre ce message pour terminer son exécution.

### 3.4.8 Mise en œuvre sur un FPGA

Nous avons réalisé la première version de la plateforme MATIP à l'aide de Xilinx ISE 12.3 sur l'ARD Xilinx FPGA Spartan 6. Nous avons synthétisé et simulé la version huit bits de la plateforme. Les tables 3.5 et 3.6 présentent les ressources consommées par la plateforme. Pour les besoins du test, le réseau a été configuré pour offrir quatre ports d'entrée/sortie et deux tâches matérielles (HT) y ont été connectées.

La consommation de ressources sur un FPGA d'entrée de gamme est moyenne et laisse assez des ressources libres pour déployer des tâches matérielles plus complexes sur cette gamme de FPGA. Chaque HT comporte une mémoire RAM interne de 8 Ko pour stocker les messages.

Les ressources logiques consommées par une instance de la tâche matérielle ont été comparées aux ressources consommées par la plateforme complète. Chaque HT occupe 8 % de la puce. Comme la plateforme entière occupe 34 % de la puce, nous en déduisons que l'infrastructure de base de notre plateforme, à savoir le composant MPI-HCL et le réseau, occupent (34-2*8) soit 18 % de la puce.

La fréquence maximale de fonctionnement obtenue est de 76 MHz sur un FPGA Xillinx Spartan-6 xc6lx45.

Tableau 3.5: Ressources consommées par une HT dans le FPGA xc6lx45

| **Slice Logic Utilization** | **Used** | **Available** | **Utilization** |
|---|---|---|---|
| Number of Slice Registers | 92 | 54576 | 1 % |
| Number of Slice LUTs | 2187 | 27288 | 8 % |
| Number of bonded IOBs | 58 | 218 | 26 % |
| Number of BUFG/BUFGMUXs | 1 | 16 | 6 % |
| Average Fanout of Non-Clock Nets | 6.88 | | |

Tableau 3.6: Ressources consommées par MATIP 1.0 avec deux HT dans le FPGA xc6lx45

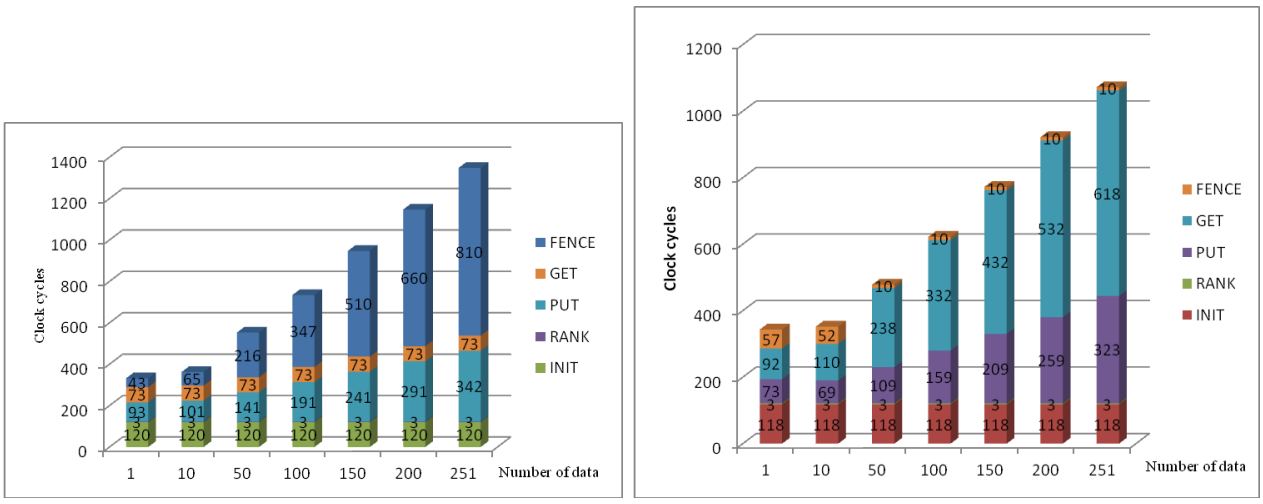
| **Slice Logic Utilization** | **Used** | **Available** | **Utilization** |
|---|---|---|---|
| Number of Slice Registers | 2426 | 54576 | 4 % |
| Number of Slice LUTs | 9447 | 27288 | 34 % |
| Number of bonded IOBs | 10 | 218 | 4 % |
| Number of BUFG/BUFGMUXs | 3 | 16 | 18 % |
| Average Fanout of Non-Clock Nets | 5.55 | | |

Tableau 3.7: Latence (en cycle d'horloge) des primitives MPI-HCL

| **Data size(byte)** | **INIT** | **RANK** | **PUT** | **GET** | **FENCE** |
|---|---|---|---|---|---|
| **1** | 120 | 3 | 93 | 73 | 43 |
| **10** | 120 | 3 | 101 | 73 | 65 |
| **50** | 120 | 3 | 141 | 73 | 216 |
| **100** | 120 | 3 | 191 | 73 | 347 |
| **150** | 120 | 3 | 241 | 73 | 510 |
| **200** | 120 | 3 | 291 | 73 | 660 |
| **251** | 120 | 3 | 342 | 73 | 810 |

Le tableau 3.7 présente les latences obtenues sur la plateforme de test lors de l'exécution de chaque primitive du composant MPI-HCL. Ces latences ont été mesurées en insérant des *sondes* pour relever le temps de simulation. Avec deux HT connectées au NoC, il n'y a ni congestion ni perturbation du trafic dues à des collisions. Par conséquent, les valeurs de latence obtenues dans le tableau 3.7 sont les valeurs optimales que nous pouvons attendre de la plateforme MATIP.

La figure 3.25 montre les performances des deux instances de HT. L'histogramme donne le nombre de cycles d'horloge nécessaires pour exécuter chaque MVP. La MPV RANK dure 3 cycles car elle consiste à lire une donnée en mémoire qui a été stockée par la MVP INIT lors de l'initialisation. Chaque accès en mémoire consomme trois cycles d'horloge, ainsi la progression des barres d'histogramme est linéaire et l'incrément vaut trois fois le volume des données à transférer. La MVP GET poste une requête et c'est à la cible de transférer les données ; c'est pourquoi sa durée n'est pas fonction de la taille des données pour la HT 1. Toutefois si le module Ex2 de réception des données de MPI-HCL est occupé, il est possible que cette MVP ne se termine pas rapidement comme le montre les données pour la HT 2. La MVP FENCE se termine lorsque tous les transferts sont achevés.




(a) HT 1 (b) HT 2

FIGURE 3.25 : résultats des performances en cycles d'horloge de MATIP

Nous avons calculé une latence de 3,43 $\mu s$ pour le transfert d'un octet et une latence de 10,72 $\mu s$ pour le transfert de 251 octets. La mise en œuvre tout en matériel de MATIP justifie le gain en performance par rapport à SOC-MPI [63] et MPI HAL [16] [91], qui sont réalisés en C puis compilés pour les Softcores NIOSII ou MicroBlaze. Saldaña et al. dans [62] pour la plateforme TMD-MPI ont obtenu une latence de 8,5 µs lors des transferts de message de longueur nulle entre unités d'exécutions instanciées dans le FPGA (voir table 3.8). MATIP offre les meilleurs résultats et montrent qu'il est avantageux d'utiliser MPI-2 RMA pour un MP-RSoC.

16. Hardware Abstraction Layer

Tableau 3.8: Comparaison des latences de plateformes MPI

| Auteur (s) | Ref | Plateforme | Latence | | Type de transfert |
|---|---|---|---|---|---|
| | | | µs | cycles | |
| Saldaña et al. | [62] | TMD-MPI | 8,5 | 340 | "on-chip TMD-MPI zero-length message" |
| Minhass et al. | [91] | MPI HAL | 23 | 1150 | "MPI minimum size packet" to near neighbor in same FPGA |
| Mahr et al. | [63] | SOC-MPI | 90 | 9000 | 128 Bytes MPI message |
| Gamom et al. | [92] | **MATIP** | **3,43** | **93** | "One byte MPI_Put latency" |

A l'aide de MPI-HCL qui nous permet d'avoir une couche de communication, nous pouvons réaliser la couche application.

## 3.5 Couche application

### 3.5.1 Introduction

Après avoir réalisé le composant MPI-HCL, nous abordons dans cette section la conception de la couche application du modèle en trois couches que nous avons proposé (voir figure 3.26). La couche application a pour but de faciliter l'intégration d'une tâche matérielle à la plateforme MATIP. La figure 3.29 montre un exemple de MP-RSoC ayant quatre tâches matérielles qu'il est possible de réaliser avec MATIP. C'est sur cette couche que le concepteur interagit avec la plateforme MATIP pour déployer son application. Nous détaillons dans la suite l'environnement de la tâche matérielle et montrons comment réaliser une application à partir de cette plateforme. Pour cela le concepteur dispose d'un *template* que nous avons appelé TIC (**Task Integration Component**).

### 3.5.2 Tâche matérielle

Dans la plateforme MATIP une **tâche matérielle** désigne la tâche utilisateur, c'est-à-dire le module qui réalise une fonctionnalité, ou un traitement recherché, et l'environnement qui permet à cette tâche matérielle de communiquer avec l'extérieur est le TIC (voir figure 3.27). La plateforme MATIP définit un modèle pour la réalisation de tâches matérielles communicantes, en associant à une tâche utilisateur, une mémoire de communication et des primitives, pour envoyer et recevoir des messages basés sur MPI.

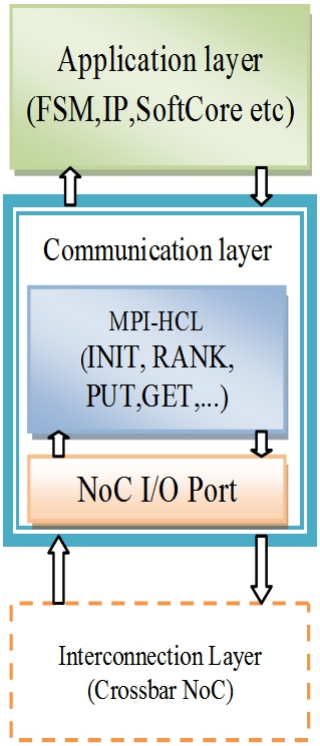



FIGURE 3.26 : Modèle en couches de la plateforme MPI-HCL

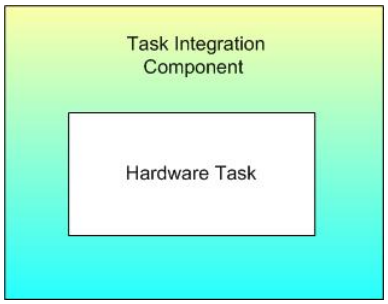



FIGURE 3.27 : Tâche matérielle intégrée dans le TIC

La figure 3.28 illustre un cas de communications entre deux tâches matérielles. La tâche matérielle 1 copie en mémoire la valeur à la sortie du compteur puis elle envoie cette valeur à la tâche matérielle 2. Cette dernière attend une valeur et lorsque celle-ci est disponible, elle l'utilise pour initialiser le compteur. Les communications s'effectuent à travers des écritures en mémoire se conformant aux mécanismes RMA.

Les échanges de messages sont réalisés à l'aide du composant MPI-HCL qui est détaillé dans la section 3.4.3. MPI-HCL prend en charge toutes les étapes de synchronisation des communications et renseigne la MVP lorsqu'une primitive est terminée. (voir figure 3.16)

### Interface entre la couche de communication et la couche d'application

Le composant MPI-HCL de la couche de communication est physiquement connecté au composant TIC de la couche application. La tâche matérielle (HT) est physiquement connectée à la couche de communication à travers la TIC (Task Integration Component)

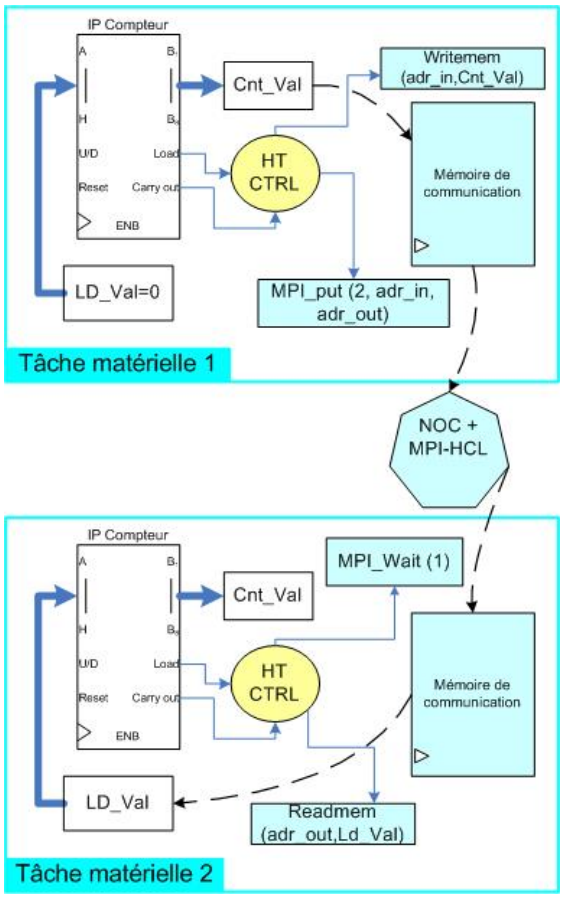



FIGURE 3.28 : Principe d'utilisation des tâches matérielles

qui comprend un ensemble de signaux physiques pour la liaison des modules et un package contenant des procédures Vhdl pour appeler les primitives de communication dans le code de la tâche matérielle.

Les primitives de communication MPI sont invoquées à travers les MVP. C'est uniquement à travers ces modules que la tâche matérielle peut exploiter les primitives de communication du composant MPI-HCL. Les MVP sont une interface entre la tâche matérielle et le composant MPI-HCL qui donne au concepteur de tâche matérielle la même vue du modèle MPI que celle d'un programmeur de tâches logicielles.

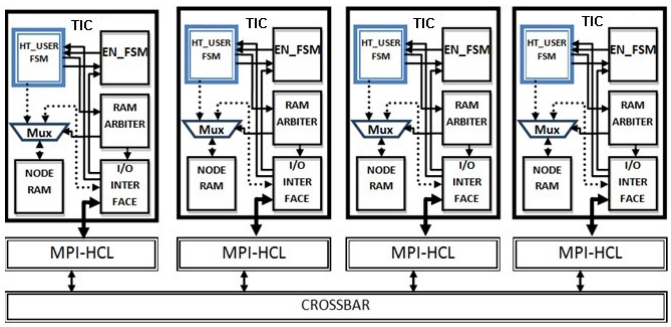



FIGURE 3.29 : Architecture de la plateforme MATIP

### 3.5.3 Composant d'intégration de la tâche matérielle : TIC

Outre la tâche matérielle à intégrer (HT_USER), le TIC comprend les modules : EN_FSM, RAM ARBITER, I/O INTERFACE, NODE RAM. Pour utiliser MATIP, le concepteur a la responsabilité de synthétiser le TIC après avoir personnalisé son HT_USER.

La table 3.9 présente les fichiers du template de l'environnement MATIP qui permettent de réaliser des applications parallèles à l'aide des tâches matérielles en utilisant MPI. Ces fichiers sont automatiquement fournis et il n'est pas besoin de les modifier pour la majorité des applications. Le fichier `hcl_arch_conf.vhd` contient des constantes qu'il faut mettre à jour pour prendre en compte le nombre de tâches et la taille du réseau de l'application parallèle. Le fichier `pe.vhd` montre comment un groupe de tâches statiques et dynamiques peuvent être connectées dans l'environnement MATIP. Pour utiliser plus de deux groupes de tâches matérielles, il conviendra de modifier le contenu du fichier `pe.vhd`.

Tous les fichiers de la table 3.9 doivent être ajoutés au projet pour réaliser une application avec MATIP. Les fichiers `ht_stat.vhd` et `ht_dyn.vhd` sont à personnaliser pour décrire la tâche de l'application. Ceci peut être fait en s'inspirant du code que nous avons décrit à la page 164.

La figure 3.30 montre la structure du TIC (pe.vhd). Il est décrit à l'aide de plusieurs modules dont le rôle est donné ci-après :

**HT_USER_FSM** : ce module contient la description de la tâche matérielle que l'utilisateur souhaite mettre en parallèle. Le code de ce module est issu des fichiers `ht_stat.vhd` ou `ht_dyn.vhd`. Il est organisé autour d'une machine à états qui permettra d'appeler les primitives MPI décrites sous forme de MVP.

**EN_FSM** : Ce module permet d'activer le module HT_USER_FSM sur requête du composant MPI-HCL.

**Mux** : Ce multiplexeur permet, soit au composant MPI-HCL, soit au module HT_USER-_FSM d'accéder à la mémoire suivant une stratégie qui est contrôlée par le composant **Arbitre RAM**.

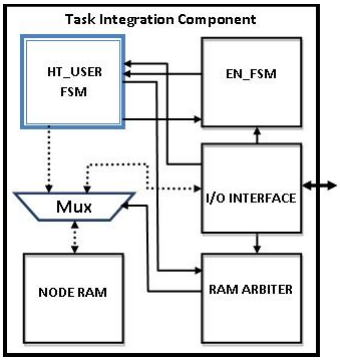



FIGURE 3.30 : Structure du TIC qui encapsule la tâche utilisateur

Tableau 3.9: Liste des fichiers du template de la plateforme MATIP (MPI Application Task Integration Platform)

| **Module** | **type** | **fichier Vhdl** | **Description** |
|---|---|---|---|
| com_icap | entité | com_icap.vhd | Contient le module de chargement de bitstream partiel |
| hcl_arch_conf | package | hcl_arch_conf.vhd | contient les constantes de l'application (Nbre de tâches statiques, nombres de tâches dynamiques,...) |
| ht_dyn | entité | ht_dyn.vhd | module pour les HT dynamiques |
| ht_stat | entité | ht_stat.vhd | module pour les HT statiques |
| mem8k8 | entité | mem8k8.vhd | module pour la mémoire BRAM de communication de 8 ko |
| pe | entité | pe.vhd | composant TIC permettant d'assembler les éléments de l'environnement d'une tâche matérielle (voir figure 3.32) |
| mpi_template | entité | mpi_template.vhd | composant permettant d'assembler les TIC, le composant MPI-HCL et le NoC |

**Arbitre RAM** : ce module gère la priorité d'accès à la RAM. il s'agit d'une Machine à états finis qui contrôle l'état de Mux suivant la valeur de deux variables binaires `Hold_Req` (demande venant de MPI-HCL) et `Rambusy` (contrôlé par USER_HT). Lorsque `Ramsel=0` alors le module USER_HT_FSM a accès à la RAM, lorsque `Ramsel=1` alors MPI-HCL a accès à la RAM). Le composant qui a accès à la RAM doit le libérer pour que l'autre puisse y accéder sauf dans le cas où `rambusy=0` (accès non verrouillé). Concrètement, pour écrire dans la RAM, le module HT_USER_FSM teste l'état de `ramsel`, si `ramsel=0`, il verrouille l'accès au bus RAM (`membusy` passe à 1), l'écriture dans la mémoire peut commencer, autrement, il faut attendre que la RAM soit libérée par MPI_HCL (`Ramsel=0`). La table de vérité 3.10 résume les principaux états de notre mutex, il respecte les contraintes de vivacité (pas de famine, pas d'étreinte fatale) et de sûreté [93].

**NODE RAM** : ce module comprend la mémoire partagée entre le module HT_USER_FSM et le composant MPI-HCL. Dans le cadre de la mise en œuvre sur un FPGA Spartan-6, cette

mémoire est réalisée à partir des BRAM qui sont contenues dans le FPGA. Cette mémoire est utilisée pour le stockage des messages, des paramètres des primitives MPI, pour la configuration des communications et pour le suivi du statut du composant MPI-HCL.

**Interface I/O** : cette interface décrit l'ensemble des signaux qui sont requis pour communiquer avec le composant MPI HCL. Ces signaux sont regroupés par fonction ; fonction d'accès à la mémoire, fonction entrées/sorties. Pour simplifier la manipulation de ces signaux dans le module HT_USER_FSM, des types Vhdl ont été créés. Le concepteur de la tâche matérielle doit déclarer des variables à l'aide de ces types dans le module HT_USER_FSM. Certains signaux permettent de partager entre toutes les MVP et le module HT_USER_FSM, les paramètres importants de l'application parallèle : rang, nombre de tâches actives, fenêtre active, état interne des procédures.

Tableau 3.10: Signaux de gestion de l'arbitre RAM

| Rambusy | Hold_Req | Ramsel | Etat de l'arbitre RAM |
|---|---|---|---|
| 0 | 0 | 0 | ➢ HT USer FSM a accès à la RAM<br>➢ MPI HCL peut à tout moment en prendre le contrôle. |
| 1 | X | 0 | ➢ HT USer FSM a accès à la RAM<br>➢ MPI HCL ne peut pas en prendre le contrôle. |
| X | 1 | 1 | ➢ HT USER FSM n'a pas accès à la RAM et ne peut pas en prendre le contrôle.<br>➢ MPI HCL a accès à la RAM. |

### 3.5.4 L'interface entre la tâche matérielle et le TIC

Le concepteur, en prenant connaissance de ces signaux, peut intégrer une IP à la tâche matérielle.

La figure 3.31 décrit l'interface du TIC. les principaux signaux de cette interface sont les suivants :

- reset : pour réinitialiser la HT ;
- clk : fournit une horloge pour la HT ;
- ce : active ou non la HT ;
- PE_in : entrée 8 bits pour la HT ;
- PE_out : sortie 8 bits pour la HT ;

Les entrées peuvent être fournies par `PE_in` et les sorties peuvent être délivrées par `PE_out`. Les autres signaux du TIC sont connectés au composant MPI-HCL, leur rôle et leur signification sont internes à la plateforme MATIP.

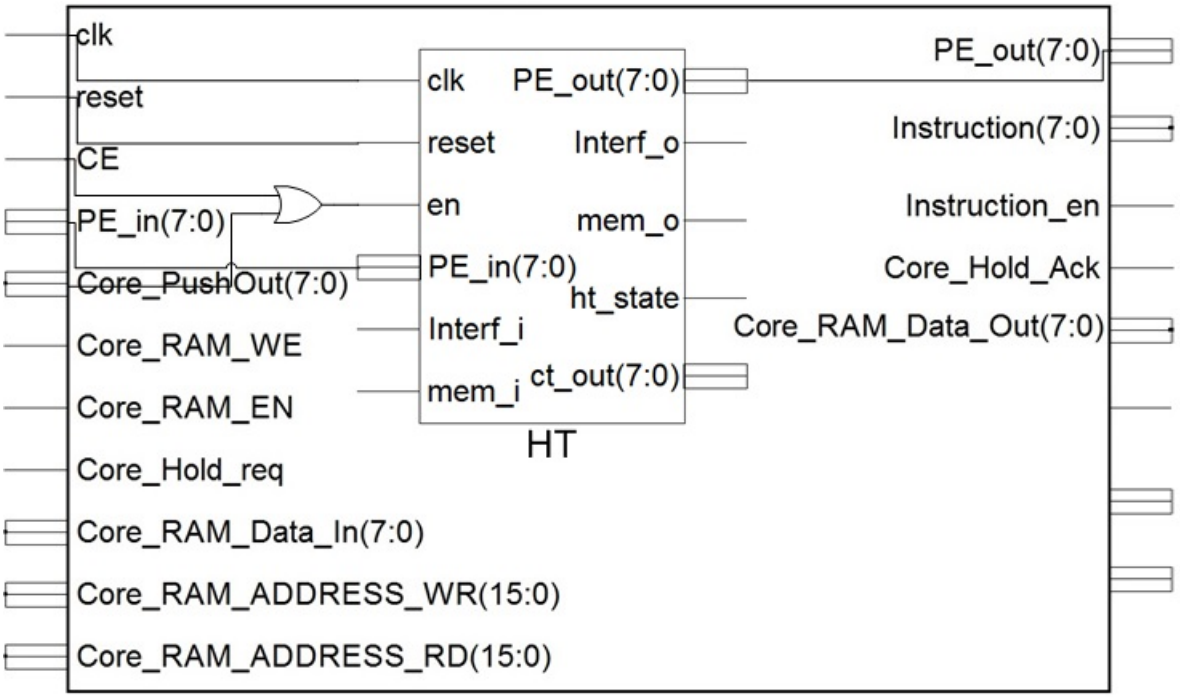



FIGURE 3.31 : Interface TIC encapsulant la tâche matérielle (HT)

### 3.5.5 L'interface de la couche application

La figure 3.32 présente l'interfaçage entre le composant de communication MPI-HCL , le composant d'interconnexion et le composant d'intégration de la tâche matérielle. Pour gérer plusieurs tâches matérielles, le concepteur doit ajouter autant de composants TIC que de composants MPI-HCL à cette architecture. La couche d'interconnexion peut être configurée pour accueillir un nombre entre 2 et 16 tâches matérielles. Elle fixe pour le moment la limite du nombre de tâches que peut gérer cette version de la plateforme MATIP.

### 3.5.6 L'interface logique entre la tâche matérielle et le composant MPI-HCL

Les MVP sont utilisées pour interagir logiquement avec la couche de communication qui comprend le composant MPI-HCL et la mémoire de communication. Dans le code de la HT, le concepteur utilise les procédures d'un package Vhdl (MPI_RMA.pkg) pour réaliser toutes les actions de communication MPI.

Lorsque la tâche matérielle fait appel à une MVP, elle passe obligatoirement trois paramètres qui sont :

- `ct` : le signal qui contrôle l'état interne de la MVP,
- SRAM : La structure permettant de lire ou d'écrire dans la RAM de communication,
- `Libr` : la structure permettant d'accéder aux entrées sorties du composant MPI-HCL.

Si la MVP utilise d'autres paramètres pour son exécution, ils sont passés à la suite des paramètres obligatoires, en respectant l'ordre donné par le prototype de cette primitive, se-

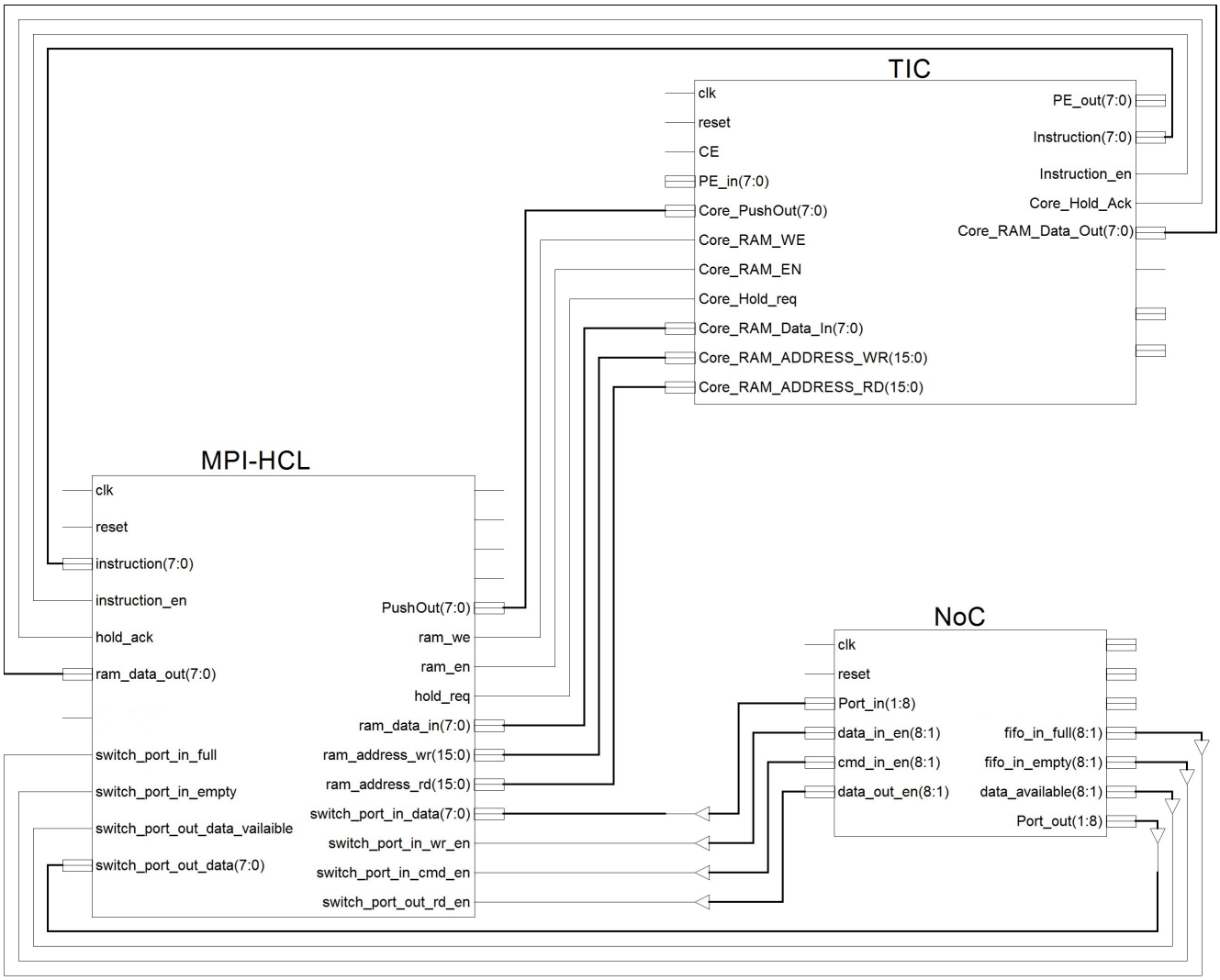



FIGURE 3.32 : Interface entre les trois composants constituant les trois couches de MATIP

lon le standard MPI.

Le signal `ct` doit être initialisé à 0 au début de l'exécution de la MVP et il repasse à 0 pour marquer la fin d'exécution de la MVP. En interne, chaque MVP est codée sous forme d'une machine à états finis. Le paramètre `ct` indique l'état courant de cette machine à états finis. Quelle que soit la MVP, nous avons retenu pour principe :

- l'état initial a le code numérique 0
- une MVP est complètement exécutée lorsque `ct` est revenu à l'état initial.

L'utilisation de signaux `SRAM` et `LIBR` qui sont des structures regroupant un ensemble de signaux complexes permet de fournir aux MVP deux objets simples pour gérer l'ensemble des tâches de communication MPI dont le concepteur a besoin.

l'extrait de code ci-dessous montre comment sont déclarés les principaux signaux requis par les MVP dans la HT.

```
--...
signal ct_state:natural:=0; --Compteur d'état interne pour les MVP
signal Libr : Core_io; --regroupe tous les signaux IO de la de la tâche matérielle
signal sram : typ_dpram; --regroupe les signaux d'accès àla mémoire de
    communication
--...
```

### 3.5.7 Description d'une tâche matérielle utilisant des modules MVP

Une MVP doit être appelée à l'intérieur d'un `process` Vhdl. Tant que *ct* n'est pas égal à 0, le *process* Vhdl de la tâche matérielle doit continuer à exécuter la MVP et ne pas passer à l'étape suivante de l'exécution de la tâche matérielle.

Le listing 3.1 illustre l'utilisation de la MVP MPI_put dans une description Vhdl pour envoyer un message. Tant que la tâche matérielle est dans l'étape `putdata`, la MVP s'exécute à chaque cycle d'horloge. A la fin de l'exécution de la MVP, le paramètre *ct* est mis à 0. C'est par ce moyen que la tâche matérielle détecte la fin de la MVP. Quelques signaux provenant du composant MPI-HCL sont directement accessibles à travers le paramètre *Libr* sans qu'il soit utile de passer par la mémoire pour consulter leur valeur ; il s'agit du résultat d'initialisation, et du résultat d'un chargement dynamique d'une tâche matérielle.

Listing 3.1: Extrait de code d'une tâche matérielle utilisant la MVP MPI_PUT

```
--...
when putdata => -- étape envoie de données
src_adr:=x''0100''; --adresse dans la mémoire côté source
dest_adr:=x''02FF''; --adresse dans la mémoire côté cible
```

```
dlen :=50 ; --nombre d'octets á transférer
destrank:=2;--le rang de la tâche cible de la transaction
pMPI_put(ct,Libr,Clk,Sram,src_adr,Dlen,MPI_char,Destrank,
    dest_adr,Dlen,Mpi_char,Default_win);
   if ct = 0 then --test de la fin de l'exécution
      RunState<=WinComplete; --prochaine étape du traitement
   end if;
When WinComplete =>
      --...
```

Nous avons testé les MVP sur un algorithme démonstratif d'envoi-réception dont le code Vhdl est en annexe B. Cet algorithme est présenté à la figure 3.33.

Cette description montre comment une tâche source envoie des données à une tâche cible en utilisant le mode de synchronisation cible active. Les MVP offrent une interface qui facilite la transcription de l'algorithme de l'application parallèle en tâches matérielles qui communiquent en utilisant l'approche MPI-2 RMA. Le concepteur peut réaliser une application parallèle en utilisant des tâches matérielles sur la plateforme MATIP tout en conservant le modèle algorithmique MPI initial.

En observant le code Vhdl du listing A.3, on peut faire les observations suivantes :

- les instructions du programme sont réalisées dans un process Vhdl synchrone,
- une construction `case` permet de délimiter chaque étape de l'algorithme,
- chaque étape peut s'exécuter en un ou plusieurs cycles d'horloge,
- la fin d'une MVP est détectée lorsque le paramètre d'appel `ct` vaut 0.

Des particularités existent pour les MVP :

- dans cette première version le type MPI_char est le seul pris en charge dans les fonctions de transfert et d'allocation de mémoire ;
- une fenêtre MPI est implicitement créée, il n'est pas besoin de créer explicitement des fenêtres dans nos transactions ;
- un seul communicateur, MPI_COMM_WORLD, est défini avec le composant MPI-HCL ;
- les types de paramètres des MVP sont simplifiés et ramenés lorsque c'est possible aux types Vhdl scalaire (natural, std_logic_vector) pour faciliter leur utilisation.

## 3.6 Discussion

Nous venons d'exposer ce qui est notre première réponse à la question : Comment déployer une application comprenant des tâches matérielles hétérogènes fonctionnant en parallèles sur un MP-RSoC ?

Notre réponse a la particularité de s'intéresser en premier à faciliter l'utilisation des MP-RSoC. Le modèle en trois couches que nous avons proposé masque des aspects matériels

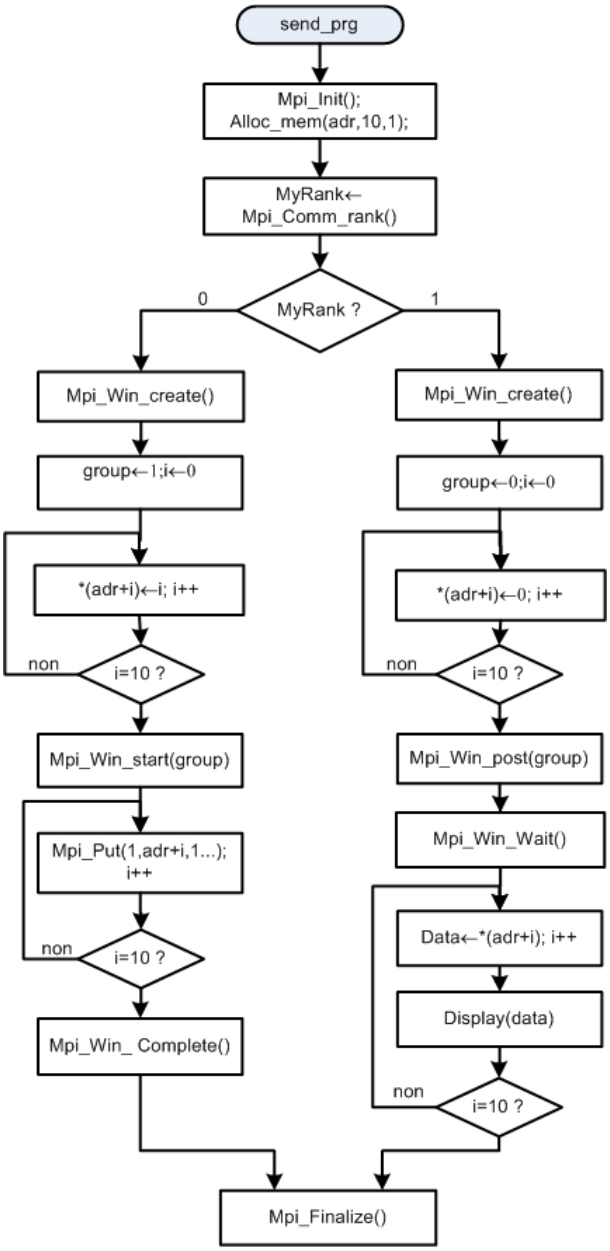



FIGURE 3.33 : Algorithme permettant l'envoi de données entre deux tâches parallèles décrites à l'aide d'un même programme.
Cet algorithme montre l'envoie d'une liste de 10 données en utilisant les fonctions MPI-2 RMA. Le rang détermine le comportement de chaque instance du programme. L'instance qui reçoit le rang 0 est la source et l'instance qui reçoit le rang 1 est la cible. la synchronisation est de type cible active. (Voir code Vhdl en annexe page 164)

de l'environnement MP-RSoC et permet au concepteur de se concentrer sur la description des fonctionnalités de son application. La facilité d'utilisation est garantie par la mise en œuvre du standard MPI qui est très populaire pour réaliser des applications parallèles et par l'exploitation de sous-routines qui est une pratique répandue dans les langages informatiques pour décrire des primitives. En effet, accomplir une fonctionnalité de communication en invoquant une routine est une pratique bien établie dans la communauté des développeurs d'applications.

En proposant un composant MPI-HCL qui intègre des routines de communication, nous avons facilité la description d'application ayant des tâches matérielles communicantes. La plateforme MATIP favorise la conception de niveau PBD en fournissant une plateforme ayant la couche d'interconnexion, la couche de communication et en proposant au concepteur MP-RSoC un template pour décrire la couche application de son MP-RSoC. MATIP est entièrement décrite en Vhdl et n'utilise pas de softcore ou d'IP propriétaires, par conséquent elle peut être mise en œuvre sur toute cible ARD.

Les performances de cette plateforme dépendent des performances de chacune des couches. Le choix du crossbar pour la couche d'interconnexion se justifie par notre volonté de ne pas pénaliser les performances de la plateforme à ce niveau. La couche de communication a été réalisée sous forme de processeur optimisé pour communication MPI. Le `template` fournit des packages Vhdl qui permettent d'invoquer des primitives MPI sous forme de procédures Vhdl. La taille du MP-RSoC que MATIP permet de décrire est limitée par les ressources physiques de la cible servant à le réaliser. La montée en charge par ajout de nouvelles tâches est possible grâce à MPI-HCL sous réserve que la couche d'interconnexion puisse gérer le nombre de tâches sollicitées et que l'ARD cible fournisse des ressources logiques suffisantes.

## 3.7 conclusion

Dans ce chapitre, nous avons exposé la méthodologie pour construire la plateforme MATIP. Nous avons rappelé le fonctionnement des primitives utilisées pour la programmation parallèle du standard MPI-2, puis nous avons réalisé un réseau d'interconnexion basé sur un crossbar, ensuite nous avons conçu le composant MPI-HCL qui met en œuvre une version matérielle des primitives MPI-2 et enfin nous avons présenté le template que le concepteur utilise pour exploiter la plateforme.

Les détails de notre mise en œuvre ont été fournis tout au long du chapitre. Le concepteur utilise un ensemble de procédures Vhdl pour invoquer les primitives de communication MPI. Nos primitives sont semblables à celles existantes dans les bibliothèques logicielles populaires de MPI comme MPICH2. Nous avons utilisé la même convention d'appel des fonctions en Vhdl que celle qui est utilisée par le modèle en C de la bibliothèque MPI. Ceci permet de faciliter la traduction des applications parallèles utilisant MPI en leur équivalent matériel pour MP-RSoC.

Le template de la plateforme MATIP masque tous les détails de l'interconnexion des tâches et facilite la génération automatique d'un environnement de communication entre des tâches matérielles dont le nombre peut varier de deux à seize. Cette limite à seize tâches est liée au réseau que nous avons réalisé et aux ressources des FPGA que nous avons utilisées lors de la synthèse. Les résultats d'implantation donnent des valeurs de latence une fois et demi plus petite que celles de la plateforme TMD-MPI pour le transfert de messages entre UE situées dans le FPGA, ceci est la conséquence de l'approche RMA qui autorise l'accès direct en mémoire lors des transferts et limite les messages de synchronisations au strict minimum entre tâches matérielles cibles et sources.

La plateforme MATIP répond déjà partiellement à notre problématique car elle permet de concevoir un MP-RSoC qui passe l'échelle, est indépendant de la technologie de FPGA, et utilise un standard de référence pour la communication entre les tâches matérielles. Nous abordons dans le prochain chapitre de cette thèse l'aspect reconfiguration dynamique du MP-RSoC afin de donner à notre plateforme la possibilité de s'auto-configurer en fonction des besoins de l'application tout en restant aussi simple d'utilisation pour le concepteur. Nous y rappelerons le fonctionnement des outils de configuration, puis nous y présenterons les développements apportés à la plateforme MATIP pour en faire un outil de déploiement de MP-RSoC.

# Chapitre 4

# Reconfiguration dynamique avec la plateforme MP-RSoC MATIP

## Sommaire

## 4.1 Introduction

Dans le précédent chapitre, nous avons présenté la plateforme MATIP en détaillant le fonctionnement de chaque couche qui la compose. Cette plateforme permet de déployer des applications parallèles sur un MP-RSoC à l'aide du langage VHDL et de primitives de communication de type MPI. Afin de profiter du polymorphisme du MP-RSoC, il est nécessaire de permettre le déploiement d'applications fortement dynamiques. L'étude du standard MPI-2 RMA nous révèle l'existence d'une primitive de chargement dynamique des tâches (`MPI_Comm_spawn`). Nous nous proposons de réaliser une version matérielle de cette primitive et de l'ajouter à MATIP afin de fournir à l'application la capacité de reconfigurer le MP-RSoC et permettre ainsi de décrire des tâches dynamiques à partir du VHDL qui est un langage de description statique par essence. Nous présentons un démonstrateur basé sur la technologie Xilinx de notre plateforme MATIP incluant la reconfiguration dynamique.

La prochaine section de ce chapitre, rappelle le fonctionnement de la primitive `MPI_Comm_spawn` dans un environnement MPI et propose différentes approches pour sa mise en œuvre matérielle ; la section trois, présente la mise en œuvre de MATIP qui est illustrée par la réalisation d'un démonstrateur sur FPGA Xilinx ; la section quatre montre les performances de MATIP et donne des détails sur son l'utilisation ; la section cinq est une conclusion.

## 4.2 Exécution dynamique des tâches matérielles

Nous proposons de prendre en compte la reconfiguration dynamique du MP-RSoC directement dans le flot de définition de l'application. Nous présentons dans la suite la primitive `MPI_Comm_Spawn` de la bibliothèque MPI qui permet de créer des tâches dynamiques.

### 4.2.1 La fonction Spawn de MPI

La fonction `MPI_Comm_spawn` permet, dans un environnement MPI, de créer à la volée des processus capables de communiquer avec les processus déjà existants.

Nous indiquons ici le prototype de la fonction tel qu'il est défini pour le langage C :

```
int MPI_Comm_spawn(
  char *command,
  char *argv[],
  int maxprocs,
  MPI_Info info,
  int root,
  MPI_Comm comm,
  MPI_Comm *intercomm,
  int array_of_errcodes[]
);
```

Cette fonction reçoit en paramètres des informations suivantes :

- Le nom de l'exécutable : `*command`
- les paramètres d'appel : `*argv[]`
- le nombre d'instances qui seront exécutées : `maxprocs`
- les spécifications pour la création des instances : `info`
- le rang de l'hôte qui crée les nouvelles instances : `root`
- le nom du communicateur parent : `comm`
- le nom de l'inter communicateur qui sera mis en place : `*intercomm`
- le code d'erreur pour chaque instance : `array_of_errcodes[]`

L'appel de cette fonction est collectif. Chaque instance de l'application parallèle incluse dans le communicateur parent doit effectuer un appel à cette fonction pour qu'il puisse être effectif.

Pour rappel, le communicateur MPI définit l'espace de diffusion partagé entre plusieurs instances d'une application. Toute instance de l'application parallèle appartient à un communicateur MPI pour la réception ou l'envoi des messages collectifs.

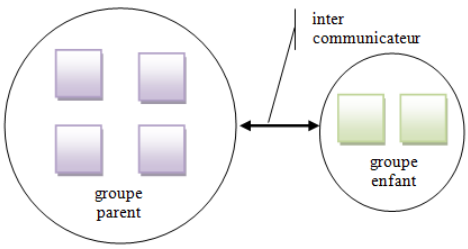



FIGURE 4.1 : Création dynamique de processus à l'aide de la fonction `MPI_Comm_spawn()`

`MPI_Comm_spawn` essaie de démarrer `maxprocs` copies identiques d'un programme MPI spécifié, d'établir une communication avec eux et de retourner un inter-communicateur. Les

processus créés sont désignés comme des fils. Les fils ont leur propre MPI_COMM_WORLD, qui est distinct de celui des parents.

#### 4.2.1.1 Fonctionnement de `MPI_Comm_spawn()`

`MPI_Comm_spawn` est une primitive collective sur le groupe `comm`, qui ne peut se terminer avant que `MPI_Init` ait été appelé par les processus fils. De même, `MPI_Init` chez les fils ne peut se terminer tant que tous les parents n'ont pas appelé `MPI_Comm_spawn`. Par conséquent, `MPI_Comm_spawn` chez les parents et `MPI_Init` chez les fils forment une opération collective sur l'union des processus parents et enfants. L'inter-communicateur retourné par `MPI_Comm_spawn` (nommée aussi **Spawn** [1] dans la suite) contient les processus parents dans le groupe local et les processus fils dans le groupe distant.

Les parents peuvent aussi démarrer plusieurs groupes distincts de processus Fils (multi spawning) et servir de relais de communication entre les groupes distincts de fils. C'est ici l'intérêt de cette fonction qui fournit à l'application un service habituellement fournit par le système d'exploitation. La fonction `MPI_Comm_Spawn` permet qu'un MP-RSoC qui est initialement en train d'exécuter `m` tâches matérielles à un instant $t_1$, puisse par la suite exécuter $n$ tâches l'instant $t_2$ avec $t_2 > t_1$. Il existe une version de `MPI_Comm_spawn` qui utilise un seul groupe et est moins complexe à mettre en œuvre. Cette variante se nomme `MPI_Comm_modify` et nous la décrivons dans le paragraphe suivant.

#### 4.2.1.2 Variante de MPI_Comm_Spawn : `MPI_Comm_Modify()`

Cette version proposée par [94] permet de modifier le nombre d'instances ou tâches d'une application parallèle. Les nouvelles tâches sont ajoutées dans le même groupe que les précédentes et il n'est pas nécessaire de créer un inter communicateur. Cette fonction est également collective. La nouvelle tâche devient participative dans l'application parallèle. Cette variante ne fait pas partie du standard officiel mais elle est présente dans certaines mises en œuvre [79] et est intéressante pour les applications lorsque les tâches participantes n'ont pas besoin d'être regroupées hiérarchiquement ; elle élimine la gestion des communicateurs entre les groupes.

La version actuelle de MPI-HCL que nous avons réalisée exploite un seul communicateur c'est pourquoi nous avons mis en œuvre cette variante afin que les nouvelles HT intègrent le groupe de HT qui effectuent l'appel.

#### 4.2.1.3 Achèvement d'une tâche dynamique

A la fin de son exécution, chaque programme MPI appelle la fonction `MPI_Finalize()` pour indiquer qu'elle met fin à toutes les communications MPI, cet appel est également collectif.

1. Littéralement traduit de l'anglais par pondre ou engendrer

Pour terminer une tâche dynamique, nous utilisons la primitive `MPI_finalize()` sans exiger que l'appel soit collectif puisque nous ne gérons qu'un seul groupe. Pour les tâches statiques, l'appel à `MPI_finalize()` par l'une d'elle exige que toutes les tâches de l'application fassent le même appel pour mettre fin à tout l'environnement MPI. `MPI_finalize()` est collectif pour les tâches statiques mais ne l'est pas pour les tâches dynamiques.

Le fonctionnement de la primitive `MPI_Comm_modify()`, ouvre la voie à plusieurs options de réalisation en matériel.

### 4.2.2 Options de mise en œuvre matérielle de `MPI_Comm_modify()`

#### 4.2.2.1 Architecture fixe

Une première option consiste à disposer d'une architecture où les tâches matérielles (HT [2]) à créer sont préexistantes dans le MP-RSoC, mais ne sont pas encore connectées. Dans ce cas, le rôle de la fonction est d'étendre l'environnement MPI existant aux HT qui étaient non connectées. Cette option :

- suppose qu'à l'avance toutes les tâches qui seront ajoutées dynamiquement sont déjà pré-configurées dans le MP-RSoC mais ne sont pas activées ;
- est rapide en exécution car seule la configuration de l'environnement MPI est nécessaire ;
- est exploitable sur toutes les plateformes configurables ;
- n'a pas besoin d'outils de configuration spécifiques pour gérer l'aspect dynamique du MP-RSoC ;
- ne permet pas la réutilisation des ressources matérielles par différentes HT ;
- oblige le pré-routage de l'ensemble des HT constituant le MP-RSoC.

#### 4.2.2.2 Architecture modifiable

Une autre option consiste à charger une nouvelle architecture en reconfigurant tout le MP-RSoC afin qu'il intègre les nouvelles HT en plus des anciennes. Dans ce cas, il est nécessaire de considérer le délai induit par la reconfiguration et la nécessité de sauvegarder le contexte ou l'état courant des précédentes tâches avant la survenue de la reconfiguration du FPGA. Le rôle de la fonction `MPI_Comm_modify()` est de réaliser la reconfiguration totale du MP-RSoC. Cette option :

- impose de disposer d'une cible reconfigurable dynamiquement ;
- nécessite un outil de configuration externe au MP-RSoC ;
- permet une occupation optimale du MP-RSoC ;
- permet de réutiliser des ressources matérielles du MP-RSoC par différentes HT ;
- introduit un délai dû à la reconfiguration globale ;

2. Hardware Task

- nécessite une synchronisation après reconfiguration pour la restauration du contexte des HT ;
- impose de tenir compte de la latence de reconfiguration lors de l'exécution de l'application.

Tableau 4.1: Synthèse des options de mise en œuvre des tâches dynamiques dans le MP-RSoC

| | **architecture fixe** | **architecture modifiable** | **architecture incrémentale** |
|---|---|---|---|
| Surface occupée/ressources utilisées | - | ++ | +++ |
| Taille du (des) fichier(s) de configuration | + | ++ | +++ |
| Optimisation de l'énergie consommée | non | oui | oui |
| Choix de la cible | toute cible | tout ARD | seul. ARD partiel |
| Latence lors de l'exécution | +++ | + | ++ |
| Réutilisation des ressources pour différentes tâches | – | + | +++ |
| Mise en œuvre de l'auto-configuration | – | + | +++ |
| ressources externes de configuration | +++ | – | + |
| ressources externes de synchronisation | +++ | – | +++ |

#### 4.2.2.3 Architecture incrémentale

Une troisième option consiste à modifier l'architecture en reconfigurant partiellement le MP-RSoC pour ajouter les nouvelles HT sans modifier la partie du MP-RSoC qui exécute les anciennes HT et sans aucune incidence sur les anciennes HT. Dans ce cas, le rôle de la fonction `MPI_Comm_modify()` est de réaliser la reconfiguration partielle du MP-RSoC. Cette option :

- nécessite la prise en charge de la reconfiguration dynamique partielle par le MP-RSoC ;
- permet une mise en œuvre interne au MP-RSoC ;
- permet une occupation optimale du MP-RSoC car les tâches placées sont les tâches actives ;
- réduit la surface à reconfigurer à la seule surface occupée par les nouvelles tâches ;
- nécessite une synchronisation après la reconfiguration partielle ;
- impose de tenir compte de la latence de reconfiguration lors de l'exécution de l'application.

Le tableau 4.1 résume les options d'architectures pour déployer les tâches dynamiques. L'architecture incrémentale est celle qui est la plus intéressante du point de vue réutilisation des ressources matérielles du MP-RSoC pour déployer les tâches dynamiques, mais c'est aussi l'architecture la plus complexe car elle demande des outils autorisant la mise en œuvre de la reconfiguration de l'ARD.

L'architecture modifiable possède une latence de reconfiguration pénalisante lors de l'exécution et nécessite des ressources supplémentaires pour synchroniser l'application. Par contre, cette architecture accepte un grand choix de cibles.

L'architecture fixe a une latence d'activation acceptable et ne nécessite pas de ressources supplémentaires. Elle est adaptée à tout type de cible ayant suffisamment des ressources logiques configurables, mais elle est limitée dans la prise en compte de la dynamicité de l'application.

### 4.2.3 Différents scénarii pour l'implémentation de la primitive spawn

En s'inspirant des trois architectures que nous avons présentées, plusieurs possibilités de mise en œuvre de `MPI_Comm_modify()` sont envisageables.

a) **Le premier scénario** (S1) consiste à disposer au préalable d'une architecture d'interconnexion pouvant accueillir le nombre de HT souhaitées et par la suite, activer les nouvelles HT lorsque `MPI_Comm_modify()` est invoquée (voir figure 4.2). Pour réaliser cette implantation, les actions suivantes doivent être effectuées :

1 activation des HT dans le MP-RSoC ;

2 connexions des nouvelles HT à la MPI-HCL ;

3 initialisation du nouveau groupe de communication ;

4 activation d'un inter communicateur utilisant le NoC.

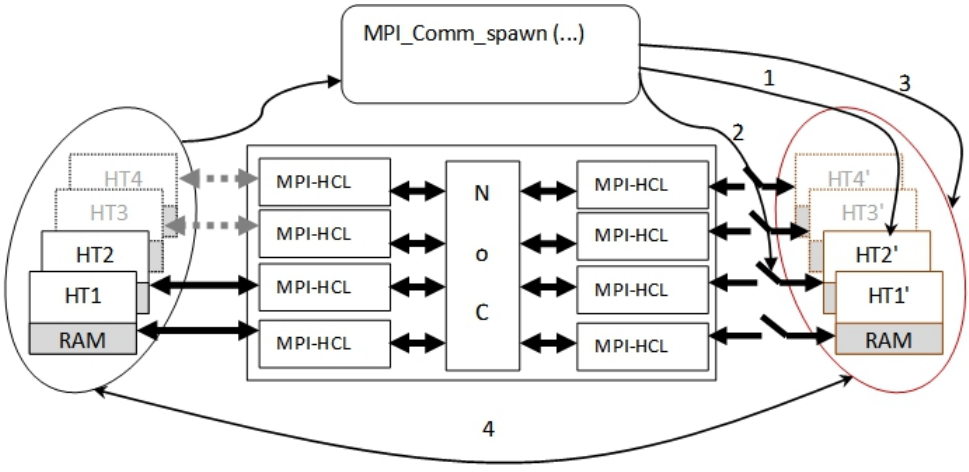



FIGURE 4.2 : *Spawn* par activation de tâches existantes

b) **Le second scénario** (S2) consiste à ajouter dynamiquement à l'ARD du MP-RSoC une instance de MATIP ayant `n` HT lorsque `MPI_Comm_modify()` est appelée. Six actions sont à effectuer dans ce cas (voir figure 4.3) :

1 configuration/placement dans l'ARD d'une instance de MATIP ;

2 configuration/placement dans l'ARD des nouvelles instances de tâches (HT') ;

3 connexion des nouvelles HT à leur MPI-HCL ;

4 initialisation du groupe de communication ;

5 mise en place d'un inter réseau entre les deux groupes ;

6 mise en place d'un inter communicateur entre deux NoC.

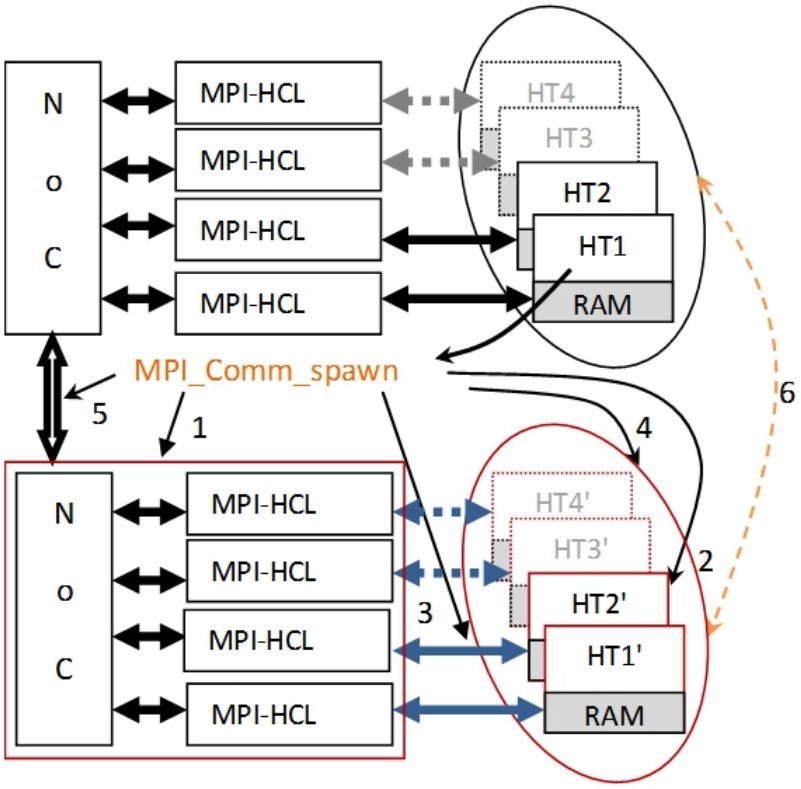



FIGURE 4.3 : *Spawn* par replication dynamique de toute l'architecture

Cette option cible les ARD dont les différentes partitions sont assez grandes pour contenir la plateforme MATIP et un ensemble de HT. Un système de développement construit à partir de plusieurs cartes à FPGA est un exemple de cibles architecturales utilisables.

c) **Le troisième scénario** (S3) consiste à disposer au préalable d'une architecture d'interconnexion pouvant accueillir le nombre de HT souhaitées puis à instancier les nouvelles HT lorsque `MPI_Comm_modify()` est invoquée (voir figure 4.4). Dans ce cas, les actions suivantes doivent être réalisées :

1 placement des HT dans la DRA ;

2 connexions des nouvelles HT à leur MPI-HCL ;

3 initialisation du nouveau groupe de communication ;

4 mise en place d'un inter communicateur.

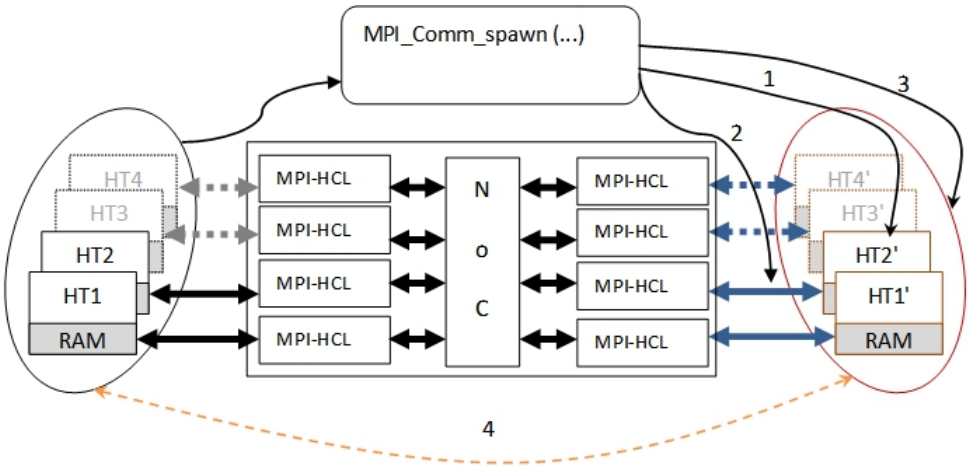



FIGURE 4.4 : *Spawn* par ajout dynamique de nouvelles tâches à une architecture existante

La table 4.2 résume les actions à mettre en œuvre selon chaque scénario de réalisation pour la primitive `MPI_Comm_modify()`. Le scénario S1 demande le moins d'actions il est le plus rapide à mettre en œuvre mais il consomme beaucoup de ressources et suppose que les tâches matérielles sont déjà configurées dans le MP-RSoC mais non activées dans l'application.

Le scénario S2 demande le plus d'actions, chaque fois que la primitive `MPI_Comm_modify()` est exécutée une nouvelle architecture matérielle de n HT est configurée dans le MP-RSoC et connectée à celle qui existait déjà, il est avantageux par rapport à S1 pour l'utilisation des ressources du MP-RSoC mais il peut introduire plus de latence d'exécution que le S1 lors de la reconfiguration. Le scénario S3 offre un compromis entre le S1 et le S2 car MATIP est déjà configurée avec la bonne taille de réseau et seules les tâches dynamiques doivent être configurées dynamiquement lors de l'exécution de la primitive `MPI_Comm_modify()`.

### 4.2.4 Choix d'un scénario pour la mise en œuvre de `MPI_Comm_modify()`

Nous avons choisi de mettre en œuvre dans un premier temps le scénario S3 pour réaliser la fonction *spawn* dans sa variante `MPI_Comm_modify()` car ce scénario est moins complexe à réaliser et permet de réutiliser des ressources pour déployer différentes tâches matérielles, ce que ne permet pas le scénario S1 même s'il est plus simple. Dans un deuxième temps, le scénario S2 sera envisagé pour mettre en œuvre la primitive `MPI_Comm_modify()` car il permet d'ajouter des instances de MATIP dynamiquement et de déployer une architecture matérielle qui correspond à l'architecture logique des applications parallèles dont les tâches sont regroupées en pools et favorisent une communication locale. S2 permet aussi de déployer MATIP sur une cible multi-FPGA.

Tableau 4.2: Synthèse des actions pour chaque scénario

| Actions à mettre en œuvre | S1 | S2 | S3 |
|---|---|---|---|
| Ajout instance MATIP | non | oui | non |
| Ajout des HT dans le MP-RSoC | non | oui | oui |
| Connexion des HT à MPI-HCL | oui | oui | oui |
| Initialisation du groupe de communication | oui | oui | oui |
| Mise ne place d'un inter réseau | non | oui | non |
| Mise en place d'un inter-communicateur | oui | oui | oui |

S1 : Activation de tâches ; S2 : duplication d'architecture ;
S3 : Ajout sélectif de tâches matérielles

### 4.2.5 Étapes d'exécution de la primitive MPI_Comm_modify() sur MATIP

La MVP `MPI_Comm_modify` permet, à l'aide de ses paramètres, d'indiquer le fichier de configuration qui contient le code de la HT à exécuter. Les bistreams sont stockés sur une ressource externe à la plateforme MP-RSoC. La primitive `MPI_Comm_modify()` déclenche la reconfiguration du MP-RSoC pour ajouter de nouvelles HT, les initialiser et initialiser l'environnement MPI avant de terminer son exécution.

Les figures 4.5 et 4.6 montrent les étapes qui permettent d'instancier une HT à partir d'une MVP :

étape 1 chaque membre du groupe exécute la MVP `MPI_Comm_modify` ;

étape 2 HCL0 [3] (voir figure 4.5) est responsable du chargement des tâches à activer ;

étape 3 HCL0 active ensuite les tâches qui ont été chargées ;

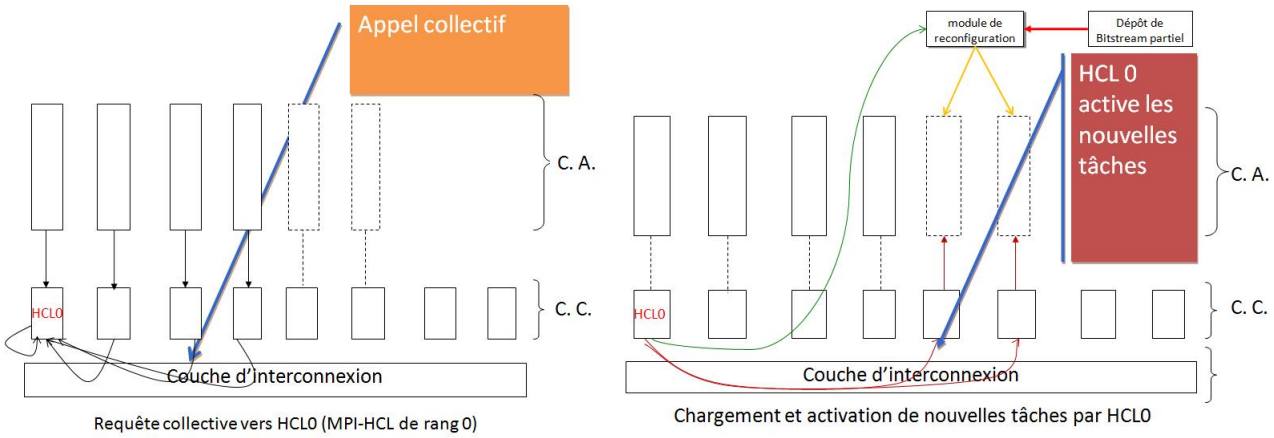
étape 4 chaque tâche nouvellement chargée appelle la Main Lib après son initialisation pour terminer le *spawn* ;

étape 5 HCL0 envoie un message à toutes les autres tâches pour leur signaler la fin du *spawn*.

#### 4.2.5.1 étape 1 : appel collectif du spawn

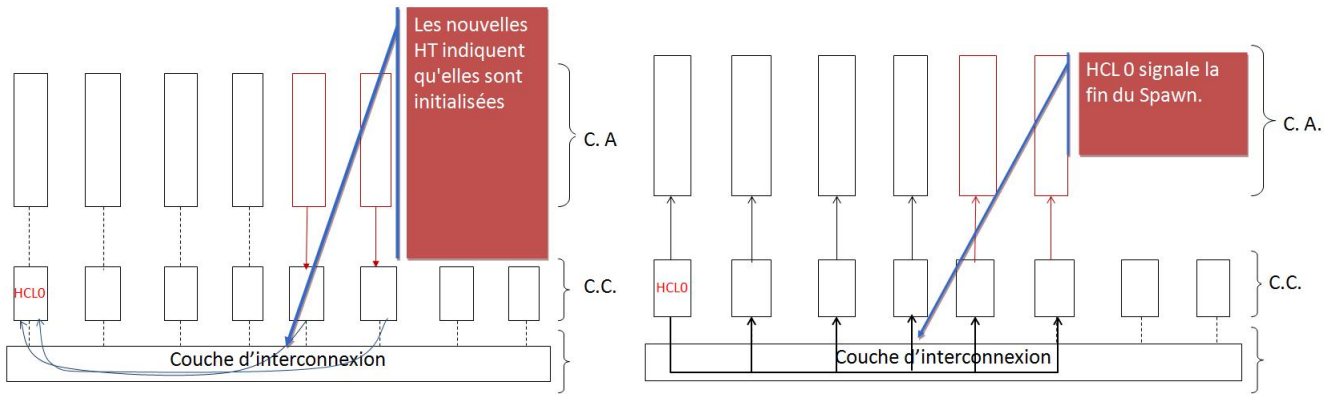
La décision de reconfigurer dynamiquement l'application parallèle ayant des tâches matérielles implique automatiquement de reconfigurer l'ARD sous-jacent. D'après les spécifications du standard MPI toutes les tâches parallèles doivent appeler la primitive *spawn*. L'appel

3. le composant MPI-HCL associé à la tâche de rang 0




(a) étape 1 (b) étape 2

FIGURE 4.5 : Exécution du *spawn* : requête, chargement et activation de nouvelles tâches.




(a) étape 3 (b) étape 4

FIGURE 4.6 : Exécution du *spawn* : Initialisation, acquittement et fin de la primitive MPI_comm_-modify.

collectif du *spawn* force toutes les tâches parallèles dans un état de non communication avant toute modification de l'ARD en utilisant une barrière de synchronisation.

#### 4.2.5.2 étape 2 : Chargement et activation des nouvelles tâches

Une fois que la barrière de synchronisation est atteinte par toutes les tâches, le module MPI-HCL de rang 0 (HLC0) invoque une primitive de reconfiguration de l'ARD. Le rôle de la primitive de reconfiguration est de configurer les modules hébergeant les nouvelles tâches matérielles et de connecter physiquement ces nouveaux modules à l'architecture existante du MP-RSoC. Lorsque la primitive de reconfiguration a terminé son exécution, elle le signale au module HCL0 qui génère un ordre d'activation pour les tâches matérielles nouvellemment chargées et leur transmet les paramètres optionnels de leur exécution. L'activation des tâches est effectuée par le composant MPI-HCL à travers le TIC encapsulant la tâche dynamique. Cette activation consiste à générer un signal de reset et à mettre à l'état haut le signal EN (enable) du module contenant la tâche matérielle.

#### 4.2.5.3 étape 3 : Initialisation des nouvelles tâches

Au début de leur exécution, les nouvelles tâches appellent la primitive `MPI_init()` pour s'initialiser et recevoir un rang MPI. Dans la variante `MPI_Comm_modify()` du *spawn*, les nouvelles tâches reçoivent un rang à la suite des rangs des tâches matérielles déjà présentes dans le groupe de tâches. HCL0 distribue les rangs aux nouvelles tâches.

#### 4.2.5.4 étape 4 : Fin du *Spawn*

Durant les précédentes étapes, aucune communication ne peut avoir lieu car les tâches matérielles sont retenues par la barrière de synchronisation. Une fois les nouvelles tâches initialisées, HLC0 met fin à la barrière en signalant à toutes les tâches la fin du *spawn* et leur communique une copie de la table `Rank2Port` qui permet à une tâche matérielle de déterminer le port réseau d'une autre tâche à partir de son rang. La table `Rank2Port` est localisée dans la zone de registres de la mémoire de communication (voir registres MPI-HCL à la page 89). Les tâches matérielles peuvent à nouveau communiquer après la fin du *spawn*.

## 4.3 Mise en œuvre de MATIP dans un FPGA Xilinx Artix 7

Le déploiement d'une application ayant des tâches matérielles parallèles dynamiques se fait en deux phases : une phase préparation qui se déroule *hors ligne* et une phase d'exécution qui se déroule *en ligne*. La phase de préparation utilise nécessairement des outils compatibles avec la technologie du fabricant de FPGA.

Dans le cas de notre démonstrateur, la phase de préparation (voir figure 4.7) a pour objet de définir à l'aide des outils Xilinx ISE, Xilinx PlanAhead et Xilinx Impact les données de configuration du FPGA.

La phase d'exécution utilise la MVP `MPI_Comm_modify()` une fois que le FPGA est configuré pour charger dynamiquement puis exécuter des tâches dans le FPGA.

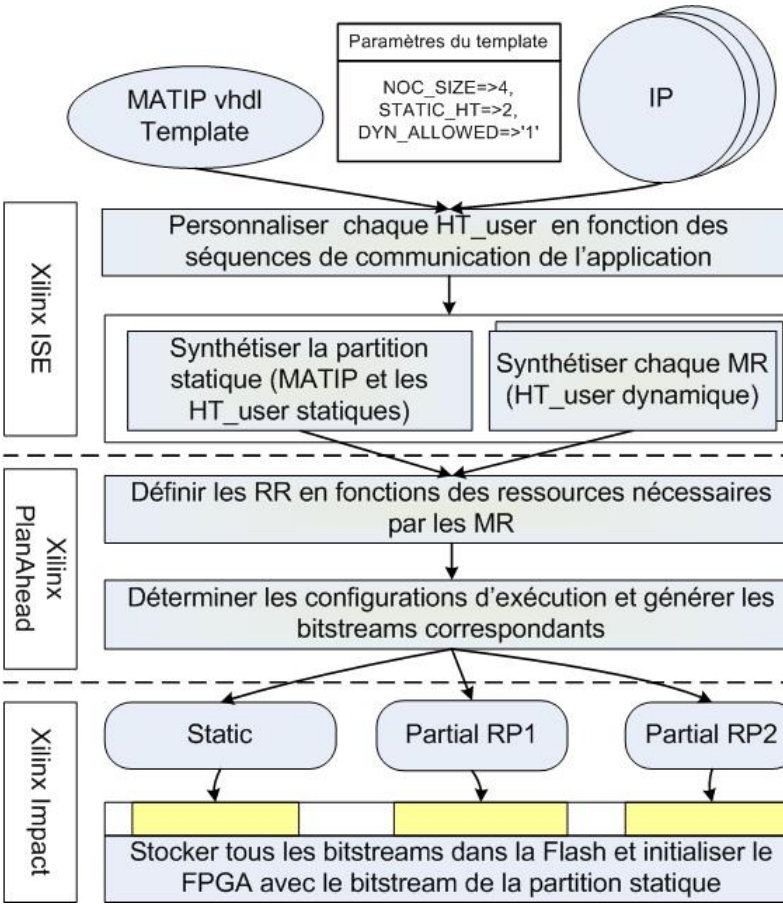



FIGURE 4.7 : Préparation du déploiement de l'application parallèle avec MATIP sur FPGA Xilinx Artix-7.

### 4.3.1 Préparation du déploiement avec MATIP

La phase de préparation comprend trois étapes principales qui sont : la synthèse des partitions de l'application, la création des données de configuration du FPGA et le stockage des données de configuration.

#### 4.3.1.1 Synthèse des partions de l'application

Xilinx ISE est l'outil qui permet de réaliser la partition statique de l'application parallèle et chaque module reconfigurable (MR) le cas échéant. Le *template* de MATIP fournit l'essentiel des composants. Dans les modules dédiés aux tâches utilisateurs (HT_USER_FSM),

le concepteur peut ajouter toutes les IP dont il a besoin. Le concepteur peut personnaliser les paramètres du *template* (voir figure 4.7) en fonction du nombre maximal de tâches de son application (paramètre `NOC_SIZE=>4`), du nombres de tâches statiques (paramètre `STATIC_HT=>2`) et du choix de l'activation de la dynamicité dans MATIP (paramètre `DYN_ALLOWED=>'1'`). Le concepteur personnalise ensuite chaque module `HT_USER_FSM` suivant les séquences de communication de l'application. En cas d'utilisation de la dynamicité des tâches, chaque module reconfigurable est synthétisé séparément.

#### 4.3.1.2 Définition des configurations

Pour mettre en œuvre la reconfiguration dynmaique dans MATIP sur un FPGA Xilinx, nous utilisons PlanAhead : le logiciel de configuration qui effectue une mise en œuvre complète d'une conception contenant une partition statique et des partitions reconfigurables [4].

Une **partition reconfigurable** (PR) est une instance d'un **module reconfigurable** (MR) qui est mappé sur une **région reconfigurable** (RR) du FPGA. Il est important de connaître le budget de ressources nécessaires pour chaque MR avant de définir les RR(voir figure 4.7).

Pour mettre en œuvre différentes configurations d'une conception, le concepteur utilise un sous-ensemble des combinaisons possibles des modules reconfigurables avec le module statique. Chaque mise en œuvre de la conception est appelée configuration. La figure 4.8 présente les différentes configurations pour une application ayant deux régions reconfigurables (RR-1 et RR-2), deux modules reconfigurables (MR-1, MR-2) et un module statique (MS), Il existe neuf configurations correspondant aux possibilités de placement et routage lors de l'exécution, ce qui demande de stocker dix huit bitstreams partiels et un bitstream complet pour la partie statique.

#### 4.3.1.3 Définition des limites de la partition reconfigurable

Nous avons créé deux régions reconfigurables (RR) dans le FPGA pour accueillir les instances HT2 et HT3. Nous avons également partitionné notre design et avons obtenu deux modules reconfigurables (MR) qui correspondent aux instances HT2 et HT3. Dans notre application une RR peut accueillir un MR quelconque ce qui donne quatre possibilités (voir figure 4.8), mais sur le plan fonctionnel de l'application, nous n'avons pas besoin de les mettre toutes en œuvre car c'est le même module qui produit les deux instances ; il est juste nécessaire que les deux MR soient présents sur les deux RR pour que la fonctionnalité dynamique de l'application soit réalisée. Nous avons ainsi défini trois configurations :

- celle avec seulement la partie statique du design mis en œuvre ;
- celle avec la partie statique et un module reconfigurable déployé sur l'une des RR ;
- celle avec la partie statique et deux modules reconfigurables déployés sur chaque RR.

4. partie du design qui est déployée sur une région reconfigurable

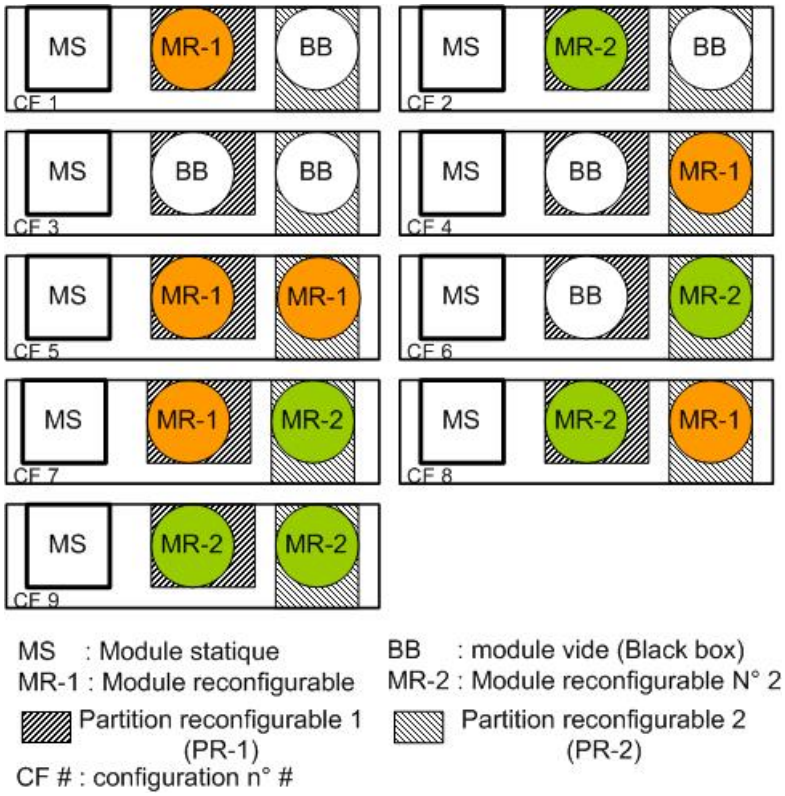



FIGURE 4.8 : Différentes configurations à partir d'un module statique et deux modules reconfigurables.

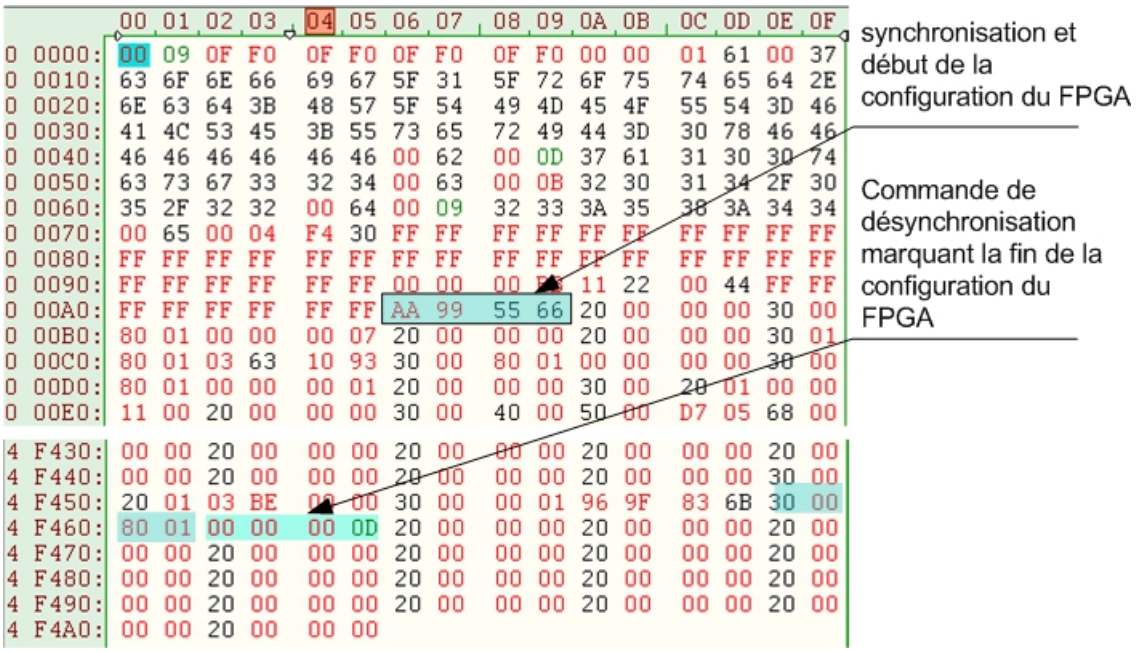
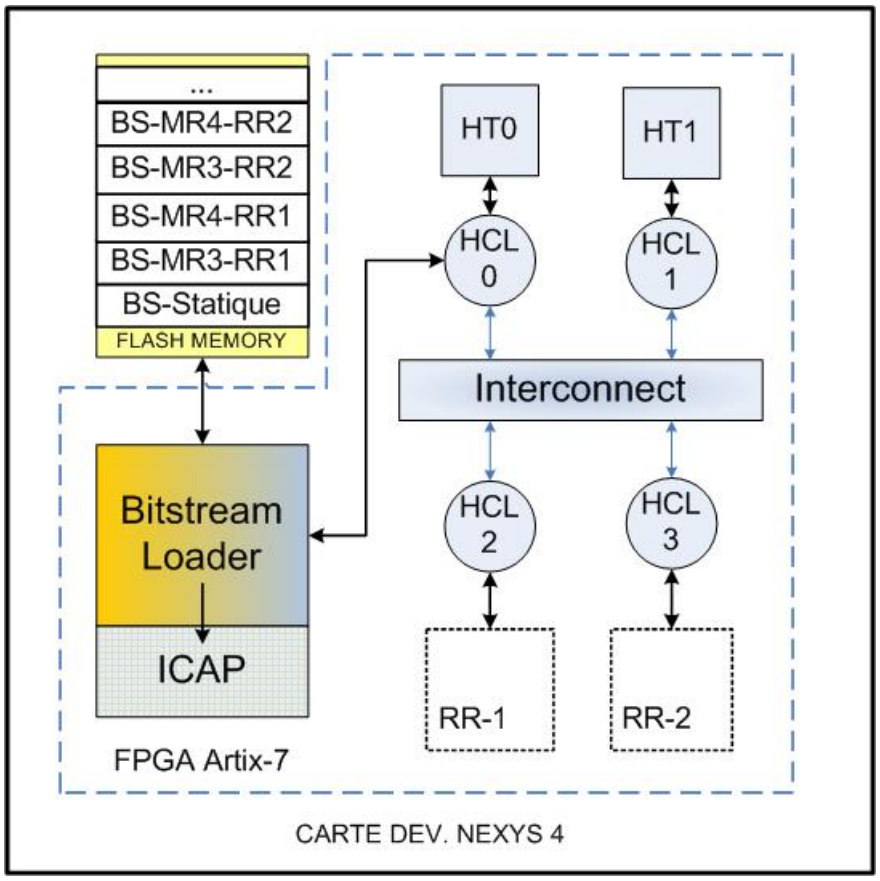



FIGURE 4.9 : Exemple de fichier Bitstream partiel généré par PlanAhead pour le FPGA ARTIX-7. le mot de synchronisation indique le début des données de reconfiguration.




**RR-x** : région reconfigurable-x **HCLx** : instance x du composant MPI-HCL
**HTx** : module de la tâche matérielle x **NoC** : Instance du module d'interconnexion
**BS-MRi-RRj** : BS de l'instance de tâche matérielle dynamique i pour la RR-j

FIGURE 4.10 : Mise en œuvre du démonstrateur MATIP sur carte Digilent Nexys 4.

Pour chaque configuration, PlanAhead nous a produit trois bitstreams ; un bitstream pour le design complet et un bitstream partiel pour chaque RR. Lorsque la RR est vide, PlanAhead propose d'y implanter un module *Black Box* duquel il produit un fichier de configuration vierge [5] [95] qui peut servir à réinitialiser les ressources de la RR dans l'état non configuré.

Nous avons eu besoin de cinq fichiers de configuration pour le démonstrateur (voir figure 4.10) :

1. BS-Statique : bitstream pour la partie statique (1 composant NOC, 4 composants MPI-HCL, HT0, HT1) ;
2. BS-BB-RR1 : bitstream d'effacement pour la RR-1 ;
3. BS-BB-RR2 : bitstream d'effacement pour la RR-2 ;
4. BS-MR2-RR1 : bitstream de l'instance HT2 pour la RR-1 ;
5. BS-MR3-RR1 : bitstream de l'instance HT3 pour la RR-1 ;
6. BS-MR2-RR2 : bitstream de l'instance HT2 pour la RR-2 ;

5. blanking bitsream

7. BS-MR3-RR2 : bitstream de l'instance HT3 pour la RR-2.

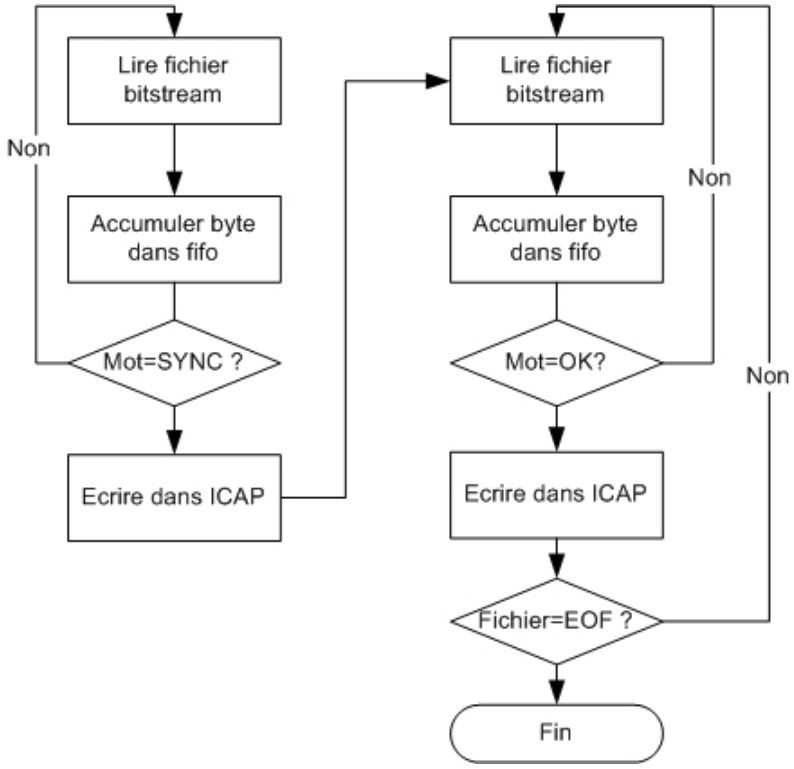



FIGURE 4.11 : Algorithme de la primitive de reconfiguration du FPGA à travers ICAP.

#### 4.3.1.4 Stockage des données de configuration

Le fichier de configuration complet est téléchargé dans le FPGA à l'aide du logiciel Xilinx Impact, et enregistré dans la mémoire Flash de la carte de développement DIGILENT Nexyx 4. L'outil Xilinx IMPACT permet également d'enregistrer les fichiers de configuration partielles dans la mémoire Flash (voir figure 4.7)

### 4.3.2 Exécution du déploiement avec MATIP

L'exécution du déploiement des tâches dynamiques est réalisée à travers la MVP `MPI_Comm_modify()` qui utilise lorsqu'elle est appelée l'IP `BStream Loader` pour reconfigurer le FPGA à la volée. Cette IP est la seule qui soit spécifique à la technologie de l'ARD utilisée ; pour utiliser MATIP sur une autre ARD il faudrait adapter cette IP en conséquence.

#### 4.3.2.1 Découplage des modules pendant la reconfiguration partielle

Du fait de la modification de la logique reconfigurable pendant que le FPGA fonctionne, la logique statique connectée aux sorties des modules reconfigurables doit ignorer les données des modules reconfigurables pendant la reconfiguration partielle. Xilinx suggère qu'un signal

d'activation soit utilisé pour isoler la partition reconfigurable jusqu'à sa complète reconfiguration. MATIP déclenche une synchronisation de type barrière au moment de l'exécution de la primitive `MPI_Comm_modify()`, par conséquent, les tâches matérielles dynamiques ne peuvent pas communiquer avant la fin de la reconfiguration et leur activation explicite, ainsi cette exigence est satisfaite par le composant MPI_HCL.

#### 4.3.2.2 Utilisation du port ICAP pour le chargement d'un bitstream partiel

Nous avons choisi d'utiliser une primitive de reconfiguration qui est réalisée sous forme d'une machine à états qui lit le contenu de la mémoire Flash puis transfère les données de configuration au module ICAP en exploitant les informations fournies par Xilinx [34]. La figure 4.9 montre la structure interne du bitstream ; il comprend une en-tête d'une longueur variable qui contient des informations destinées aux outils de Xilinx concernant le bitstream et le FPGA pour lequel il a été généré. Ces informations sont ignorées par l'interface de configuration ICAP. A la suite de cette en-tête on trouve une séquence d'octets particulière appelée mot de synchronisation (SYNC) qui permet de détecter le début des données de configuration proprement dites [96]. Cette en-tête est AA995566. Tous les mots à partir de ce mot de synchronisation sont à envoyer à l'interface ICAP. La fin de la configuration est marquée par l'envoie de la commande DESYNC (30 00 80 01 00 00 00 0D) suivie par l'envoie d'au moins deux commandes NOP (20 00 00 00) qui permettent de désynchroniser le FPGA et de repasser en mode d'exécution. (voir figure 4.9).

L'algorithme utilisé par la primitive de reconfiguration partielle du FPGA par MATIP est présenté à la figure 4.11. L'interface ICAP peut recevoir des mots de 8 bits, de 16 bits ou de 32 bits. L'algorithme suppose que le fichier est lu octet par octet. Comme le port ICAP est configuré en 32 bits dans le FPGA Artix-7 que nous avons utilisé, la primitive compose un mot correct de 4 octets avant de l'écrire sur le port de l'ICAP.

### 4.3.3 Méthodologie de déploiement du démonstrateur MATIP

L'application de test que nous avons réalisée pour illustrer le fonctionnement de la reconfiguration dynamique à l'aide de la plateforme MATIP comprend quatre tâches matérielles dont deux sont statiques et deux sont dynamiques. Lors de l'exécution de cette application, les deux tâches statiques (HT0 et HT1) s'envoient des données puis configurent à l'aide de la primitive `MPI_Comm_modify()` deux autres tâches dynamiques (HT2 et HT3) qui envoient toutes deux des données à la tâche de rang 0.

Le chargement des fichiers de configuration partielles se fait à partir d'une machine à état qui lit les données dans la mémoire Flash de la carte Nexyx4 et les communique à l'interface de programmation interne ICAP pour la reconfiguration du FPGA chaque fois qu'une primitive `MPI_Comm_modify()` est exécutée.

La figure 4.12 montre le plan d'implantation de la partition statique et les emplacements pour les deux modules reconfigurables du démonstrateur de la plateforme MATIP.

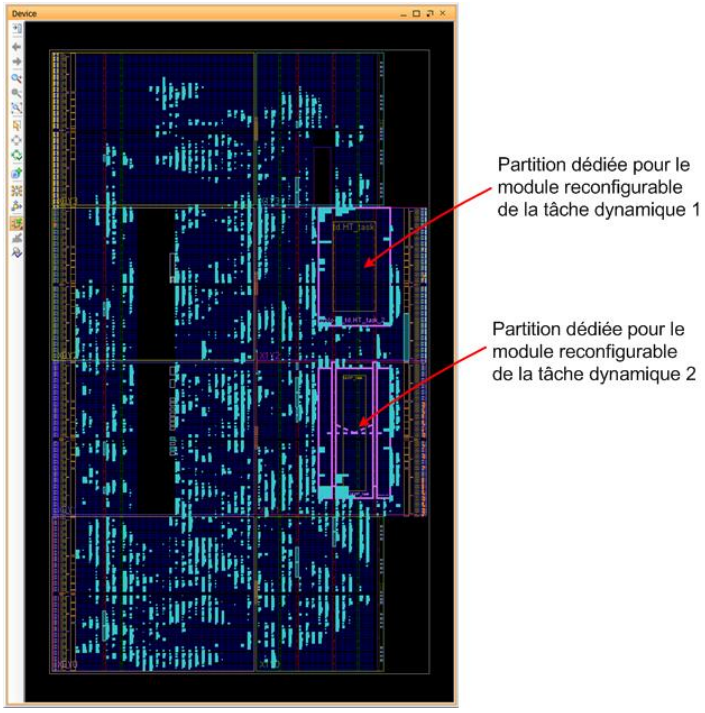



FIGURE 4.12 : Implantation de la partition statique de MATIP dans un FPGA Artix-7.

## 4.4 Performances de MATIP et détails sur son utilisation

Les développements de la plateforme MATIP ont premièrement ciblé le FPGA Spartan 3. Du fait de la limitation des ressources et à cause du manque de support pour la reconfiguration partielle, nous avons opté pour un Spartan 6. La table 4.3 donne les ressources consommées dans ce FPGA par la plateforme MATIP. Nous pouvons y voir le nombre et la variation de ressources utilisées pour deux versions de la plateforme ; une première qui n'intègre pas les primitives Spawn et qui n'utilise pas de bloc de mémoire embarquée (BRAM) et une seconde version qui intègre la primitive Spawn et utilise les BRAM. L'ajout de cette primitive nous a conduits à beaucoup de développements et à l'utilisation de plus de ressources mémoire du FPGA. Nous avons également changé la méthode de synchronisation par barrière (utilisation de `Win_fence`) que nous avions préalablement utilisée par une synchronisation avec cible active (`Win_post, Win_wait`) car elle génère moins de messages à travers le réseau. Toutefois, nous avons constaté que cette synchronisation demande plus de registres pour sa gestion.

Le support de la reconfiguration dynamique partielle sur Spartan 6 n'est pas disponible dans les outils de Xilinx. Il est envisageable d'utiliser des outils tiers et nous avons essayé sans succès, l'outil GoAhead [46] pour activer la reconfiguration des modules dynamiques de

notre plateforme sur Spartan 6. Nous avons finalement ciblé le FPGA Artix-7 pour mettre en œuvre la plateforme MATIP parce que les outils PlanAhead et XST de Xilinx supportent nativement la reconfiguration dynamique partielle du FPGA Xilinx Artix-7.

Les résultats de synthèse du tableau 4.3 montrent que le démonstrateur ayant quatre tâches matérielles occupe 13105 LUT soit 48 % des ressources sur un FPGA xc6lx45. La primitive Spawn ajoute un surcoût de 14 % en consommation des ressources à la plateforme MATIP mais la possibilité de réutiliser des ressources pour déployer des tâches dynamiquement est un avantage qui justifie ce développement.

Tableau 4.3: Consommation des ressources après synthèse de la plateforme sur un Xilinx Spartan 6 xc6lx45

| Utilisation des tranches (slices) | **Disponible** | **MATIP sans la primitive spawn** | **MATIP avec la primitive Spawn** | **variation** |
|---|---|---|---|---|
| Number of Slice Registers | 54576 | 2426(4 %) | 5858 (10 %) | 6 % |
| Number of Slice LUTs | 27288 | 9447(34 %) | 13105 (48 %) | 14 % |
| Number of bonded IOBs | 218 | 10(4 %) | 10 (4 %) | 0 % |
| Number of BUFG/ BUFGMUXs | 16 | 3(18 %) | 2 (12 %) | -1 % |
| Number of RAMB16BWERs | 116 | - | 16 (13 %) | 13 % |
| Number of RAMB8BWERs | 232 | - | 8 (3 %) | 3 % |

La mise en œuvre du démonstrateur sur un FPGA Artix-7 xc7a100t donne un taux de consommation des ressources de 38 % (voir table 4.4). L'overhead comprenant les composants MPI-HCL et TIC pour l'ajout d'une tâche matérielle est évalué à moins de 4 % des ressources du xc7a100t. Ainsi, MATIP ajoute un surcoût faible pour le déploiement dynamique des tâches matérielles à l'aide des primitives de la bibliothèque MPI.

Le tableau 4.5 montre que les ressources supplémentaires nécessaires sur MATIP pour déployer une tâche matérielle sont limitées par rapport à celles nécessaires sur FOSFOR [61]. Pour chaque plateforme, nous avons comparé les ressources utilisées par l'intergiciel et par les modules environnant la tâche matérielle. L'intergiciel [6] correspond à la couche de communication pour MATIP et il correspond aux couches HwOS (système d'exploitation matériel) et CS (service de communication) pour FOSFOR [61]. Lorsque nous comparons les ressources utilisées par le middleware et le wrapper autour des tâches matérielles pour assurer les communications, la plateforme MATIP occupe 23 % moins de registres et 10

6. Middleware

% moins de LUT que FOSFOR et surtout elle n'a pas besoin d'un système d'exploitation matérielle qui a lui seul occupe plus de de ressources que toute la plateforme MATIP (voir tableau 4.5). le tableau 4.6 compare MATIP et FOSFOR sous le rapport du déploiement des tâches matérielles.

Tableau 4.4: Utilisation des ressources par module du démonstrateur

| Module Name | Slices | Slice Reg | LUTs | LUTRAM | BRAM |
|---|---|---|---|---|---|
| Total xc7a100t* | 15850 | 126800 | 63400 | 19000 | 405 |
| **DEMO_MATIP** | **6066 (38%)** | **6850** | **15122** | **233** | **20** |
| BStream loader | 10 | 27 | 36 | 0 | 0 |
| TIC1 | 327 | 274 | 805 | 0 | 2 |
| TC1.HT_task | 280 | 242 | 717 | 0 | 0 |
| TIC2 | 331 | 274 | 811 | 0 | 2 |
| TIC2.HT_task | 282 | 242 | 720 | 0 | 0 |
| TIC3 | 268 | 222 | 687 | 0 | 2 |
| TIC3.HT_task | 225 | 190 | 604 | 0 | 0 |
| TIC4 | 253 | 194 | 640 | 0 | 2 |
| TIC4.HT_task | 213 | 162 | 556 | 0 | 0 |
| **MATIP L1& L2** | **4464** | **5070** | **11351** | **96** | **8** |
| HCL0 | 972 | 1088 | 2428 | 24 | 0 |
| HCL1 | 949 | 1088 | 2425 | 24 | 0 |
| HCL2 | 963 | 1086 | 2424 | 24 | 0 |
| HCL3 | 914 | 1086 | 2419 | 24 | 0 |
| Interconnect 4x4 | 666 | 722 | 1655 | 0 | 8 |

* Un slice contient 4 LUT et 8 registres sur un Xilinx Artix-7 xc7a100t

La fréquence maximale de fonctionnement de la plateforme MATIP sur FPGA Artix-7 est de 100 MHz ce qui est recommandé pour un fonctionnement sur carte Nexys-4. Il est possible d'améliorer ces performances en optimisant les descriptions Vhdl des modules du composant MPI-HCL mais ceci n'est pas l'objet de cette première version de MATIP. Une prochaine version de MATIP utilisera un NoC et composant MPI-HCL optimisés en terme de ressources logiques utilisées, et de performances temporelles.

### 4.4.1 Performances et utilisation des MVP

Nous évaluons les performances des primitives MPI de MATIP, en utilisant le démonstrateur que nous avons précédemment réalisé. Ce démonstrateur comprend quatre tâches matérielles dont deux sont statiques et deux sont dynamiques (Voir figure 4.13). Les tâches statiques HT0 utilisent la primitive `MPI_put()` pour envoyer des données vers HT1 et `MPI_get`

Tableau 4.5: Ressources utilisées par les démonstrateurs FOSFOR et MATIP

| FOSFOR DEMO [61] | REGISTERS | LUTS | BRAMS |
|---|---|---|---|
| Leon based MPSoC | 4463 | 7951 | 9 |
| 4 reconfigurable actors | 8172 | 27256 | 8 |
| **Actors Overhead** | 3368 | 2088 | 4 |
| **Hardware Middleware** | **2412** | **9118** | 0 |
| **Hardware OS** | **11286** | **26292** | 0 |
| **DRAFT NoC 8 ports** | **920** | **2902** | 0 |
| | | | |
| **MATIP DEMO** | **Slice Reg** | **LUTs** | **BRAM** |
| **Total** | **6 850** | **15 355** | **20** |
| 4 HT | 836 | 2597 | 0 |
| **Surcoût HT = 4 *(TIC-HT) + Bs-tream loader** | **128** | **346** | 8 |
| **MPI-HCLx4** | **4 348** | **9 792** | |
| **HARDWARE OS** | **0** | **0** | **0** |
| Interconnect 4x4 | 722 | 1655 | 8 |
| | | | |
| **(Actors Overhead + Hardware MiddleWare)-(Surcoût HT + MPI_HCLx4)** | **1 304** | **1 068** | **-4** |

pour recevoir des données de HT1. Dans le même temps HT1 utilise la primitive `MPI_put()` pour envoyer les données vers HT0 et sollicite des données de HT1 à l'aide de la primitive `MPI_get()`. Ensuite HT0 et HT1 chargent deux tâches dynamiques (HT2 et HT3) à l'aide de la primitive `MPI_Comm_modify()` et celles-ci, une fois chargées envoient des données à la tâche statique HT0 à l'aide de la primitive `MPI_put`. Plusieurs exécutions de l'application ont été effectuées en faisant varier la taille des données de 1 à 248 octets.

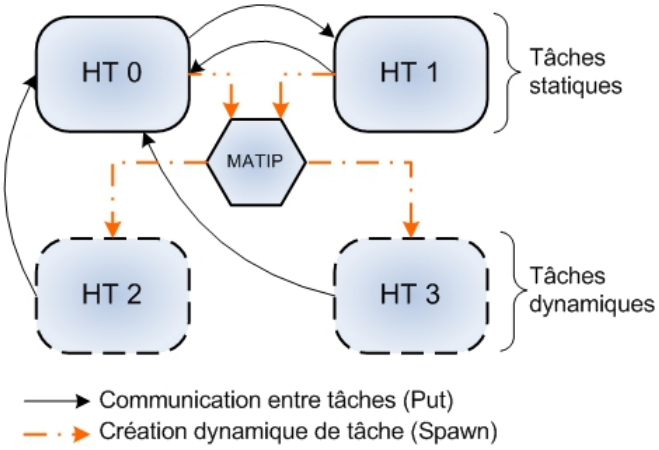



FIGURE 4.13 : Démonstrateur MATIP avec 4 tâches dont deux sont chargées dynamiquement.

### 4.4.2 La latence des primitives de communication

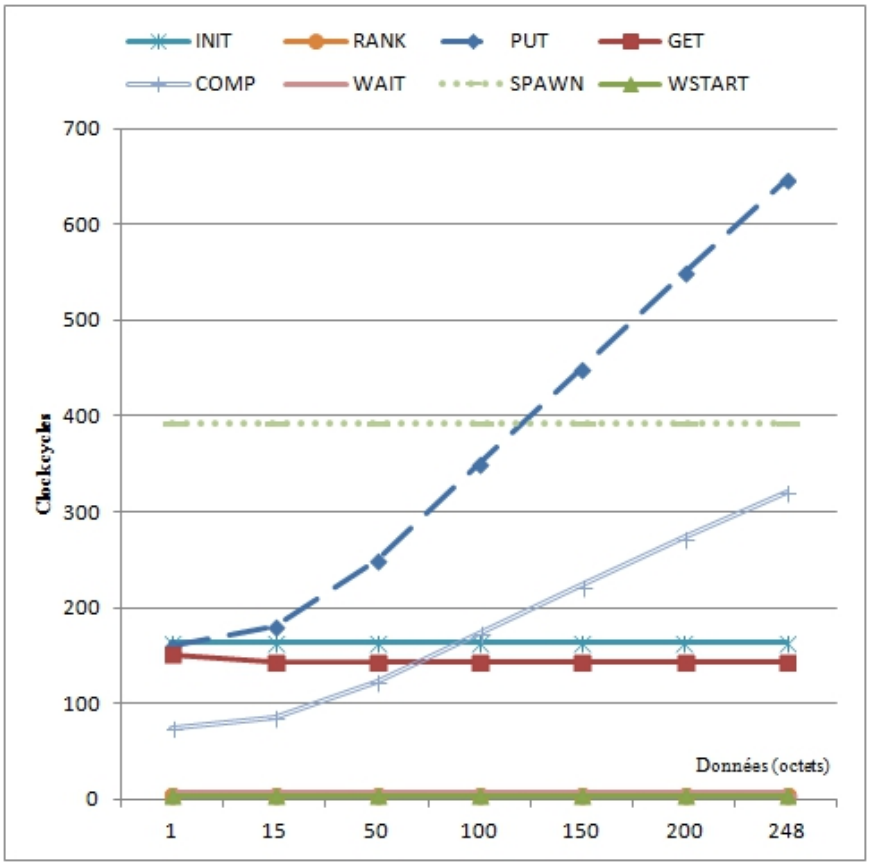
Les figures 4.14 et 4.15 montrent les temps d'exécution nécessaires pour chaque primitive sur chaque tâche matérielle. La primitive `MPI_Init()` a une latence qui est constante autour 200 cycles environ ; chaque HT n'est initialisée que si elle a reçu un rang du composant principal (noté HCL0). HT0 étant connecté à HCL0, il s'initialise le plus vite (163 cycles) puis initialise les autres composants. Les primitives `MPI_Win_Start()` et `MPI_Win_Post()` ont une latence constante (4 cycles et 5 cycles) car elles n'envoient pas de messages à travers le NoC. Elles initialisent des registres et se terminent aussitôt.
Les primitives `MPI_Win_Complete()` et `MPI_Win_Wait()` ont une latence qui varie en fonction du nombre d'octets de données *Dlen* impliquées dans la transaction et qui vérifie les relations :

$$L_{Wcomp} = 73 + Dlen \text{ et } L_{Wwait} = 6 + Dlen.\text{(en cycles)}$$

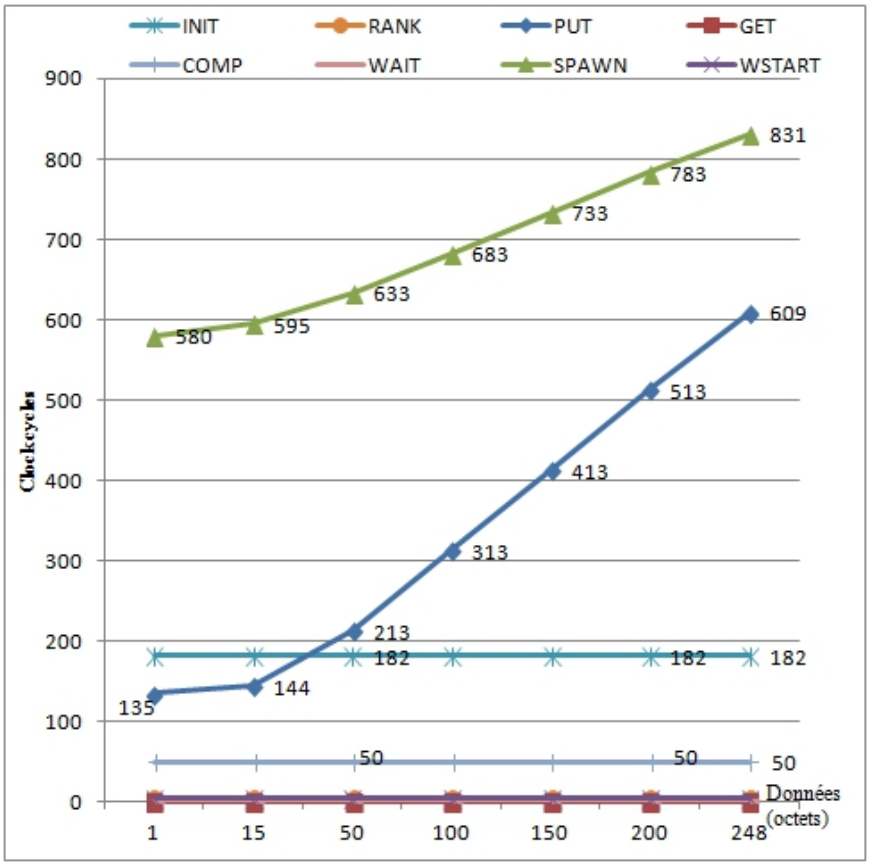
Dans le cas de cet exemple, `MPI_Win_Wait()` n'attend pas car les données sont déjà arrivées lorsqu'elle est appelée.
La primitive `MPI_Put()` a une latence qui varie toujours en fonction de la taille des données *Dlen* à transmettre. Elle vaut environ :

$$L = 150 + 2 * Dlen \text{ pour Dlen}>15$$
$$L = 150 + Dlen \text{ pour Dlen}<15$$

Sur la HT1 la primitive *Spawn* a une latence qui augmente avec la taille des données. Comme *Spawn* est une primitive collective, une barrière de synchronisation est atteinte lorsque cette

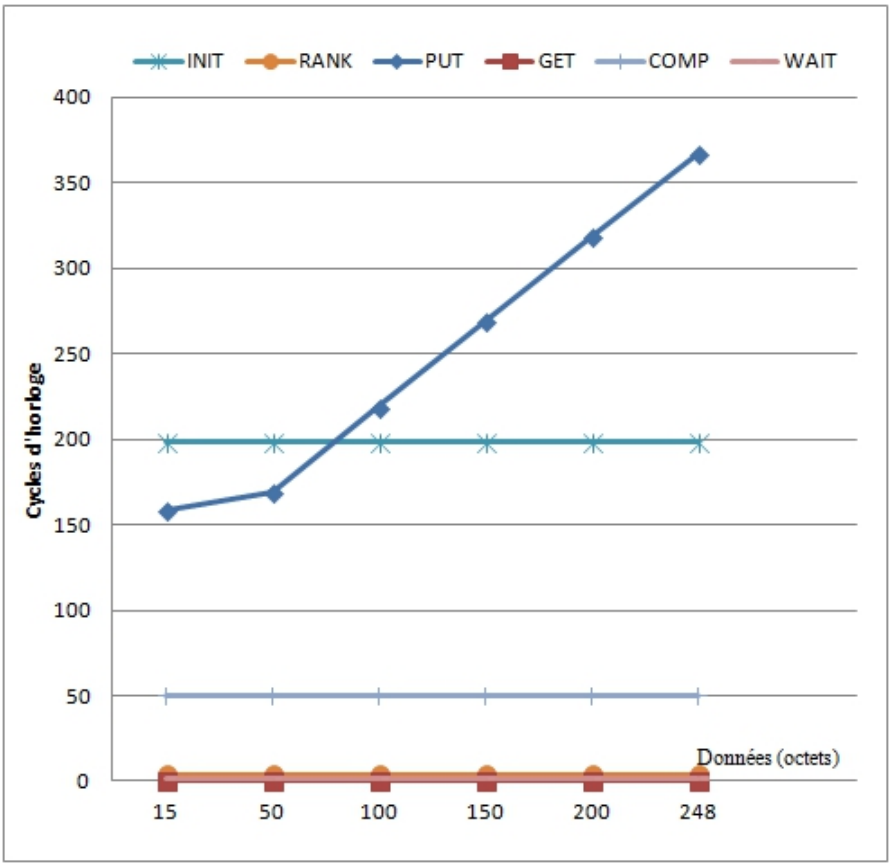



(a) HT0

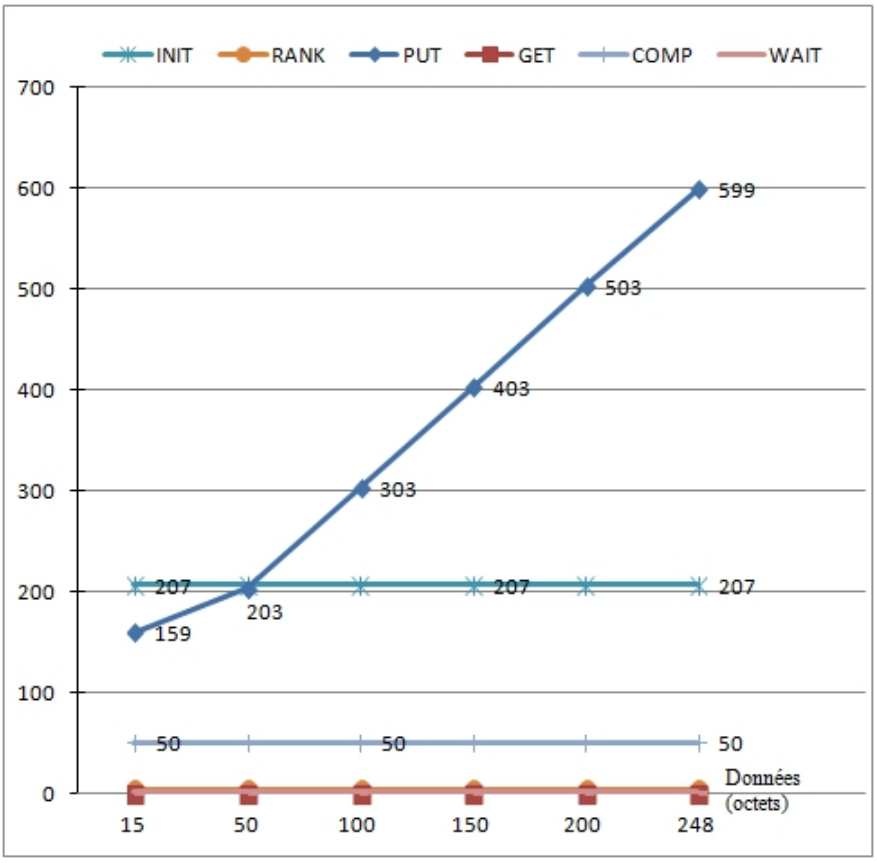



(b) HT1

FIGURE 4.14 : Performance des MVP des tâches matérielles statiques




(a) HT2




(b) HT3

FIGURE 4.15 : Performance des MVP des tâches matérielles dynamiques

primitive est appelée. Toutes les communications MPI-RMA en cours dans le groupe de tâches doivent se terminer et toutes les tâches doivent appeler la primitive Spawn avant le début du traitement. La HT1 envoie des données vers la HT0 puis appelle la primitive *Spawn* tandis que la HT0 envoie des données vers la HT1 et sollicite des données de la HT1 avant d'appeler *Spawn*, les transactions durent plus longtemps et la HT1 passe plus de temps d'attente à la barrière créée par la primitive *Spawn*. Le temps d'exécution de *Spawn* est constant sur la HT0 et nous donne le temps de latence de cette primitive qui est d'environ 400 cycles d'horloge. La version de la primitive `MPI_Comm_modify()` évaluée ici ne charge pas de bitstream dans le FPGA, elle active un module déjà pré-chargé. Le temps d'exécution de `MPI_Comm_modify()` mesuré en cycles d'horloge sur la figure 4.14 rend compte de tout le processus d'activation et d'initialisation des deux tâches dynamiques.

Tableau 4.6: Comparaison entre FOSFOR et MATIP

| | FOSFOR | MATIP |
|---|---|---|
| Paradigme de mémoire | mixte | Distribuée |
| Mode de communications | Canaux virtuels point à point | passage de messages |
| type Intergiciel | OS | Bibliothèque |
| Empreinte de surface | ++ | – |
| Latence des primitives | – | + |
| Passage à l'échelle | Non | Oui |
| Abstraction de la reconfiguration | Oui | Oui |

### 4.4.3 Utilisation de MATIP pour synchroniser des HT

Les primitives fournies par le composant MPI-HCL permettent au concepteur de contrôler la synchronisation des communications entre les tâches. L'application de démonstration comporte quatre tâches matérielles dont les transactions sont représentées par les en traits interrompus sur la figure 4.13. Nous avons produit le chronogramme d'exécution des primitives de communication à la figure 4.16, les tâches matérielles HT0, HT1, HT2 et HT3 s'exécutent en parallèle mais chacune suit un principe séquentiel.

Pour utiliser les primitives de MPI-HCL, il est nécessaire de modéliser chaque tâche (au moins la partie liée aux communications avec les autres tâches) à l'aide d'une machine à états finis.

La phase statique de l'application sur la figure 4.16 consiste pour HT0 à envoyer des données à HT1 et à recevoir des données de HT1 et pour HT1 elle consiste à envoyer des données à HT0.

- les tâches HT0 et HT1 invoquent, dans l'ordre, les primitives de communications en utilisant les MVP correspondantes ;
- pour autoriser la réception des données HT0 et HT1 appellent la primitive `MPI_Win_post()` ;
- pour signaler la fin de l'envoie des données, les tâches HT0 et HT1 utilisent chacune `MPI_Win_complete()` ;
- pour attendre la fin des transferts distants HT0 et HT1 appellent la primitive `MPI_Win_wait` ;
- la fin de `MPI_Win_wait()` sur HT0 est synchronisée par la réception du message `MPI_Win_complete()` venant de HT1.

La phase dynamique de l'application sur la figure 4.16 consiste d'une part, à charger deux tâches HT2 et HT3 puis à initialiser sur HT0 une fenêtre permettant de recevoir des données, et d'autre part à envoyer des données depuis HT2 et HT3 vers HT0.

- toutes les tâches statiques doivent appeler `MPI_Comm_modify()` ;
- les tâches dynamiques HT2 et HT3 ne se chargent qu'après que HT0 et HT1 ont appelé `MPI_Comm_modify()` ;
- la fin de `MPI_Comm_modify()` sur HT0 et HT1 est synchronisée par la fin de l'initialisation des deux tâches HT2 et HT3 ;
- `MPI_init` permet d'initialiser la tâche HT2 (resp. HT3) pour communiquer en utilisant MPI-HCL ;
- `MPI_win_start()` permet de réserver une fenêtre en mémoire de communication pour les transactions du côté de la source ;
- la fin de `MPI_Win_wait()` sur HT0 est synchronisée par la réception du message `MPI_Win_complete()` venant de HT3 ;
- `MPI_finalize` sur HT2 (resp. HT3) désactive la tâche dynamique et interdit toute future communication avec elle ;
- l'appel `MPI_finalize()` sur HT0 (resp HT1) permet de terminer l'application de communication.

Cette description présente une application parallèle ayant 4 modules fonctionnels qui seront transcodés en tâches matérielles.

### 4.4.4 Exemple de description des tâches matérielles en Vhdl

Les tâches de l'application de démonstration sont décrites en Vhdl dans le module ht_process (voir extraits de code du listing 4.1 à 4.7). Les quatre tâches sont décrites à l'aide du même module. Au moment de l'exécution, le rang MPI de chaque tâche permet de

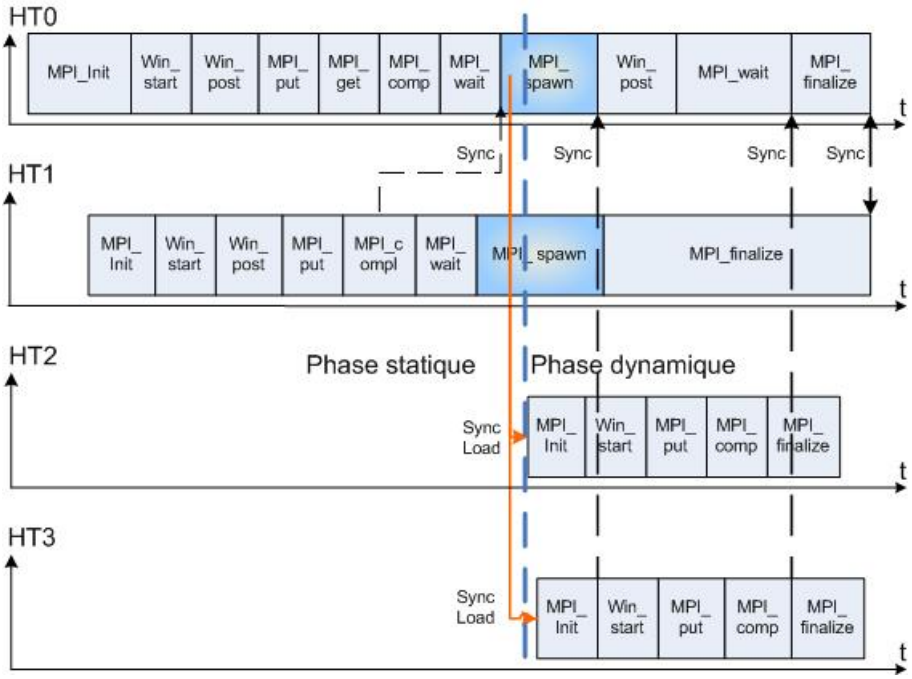



FIGURE 4.16 : Chronogramme des communications dans l'application de démonstration

l'identifier afin de fixer son rôle. Les tâches statiques 0 et 1 s'envoient des données puis créent deux tâches dynamiques et HT1 se termine pendant que HT0 attend les données de HT2 et HT3. Cette application illustre le fonctionnement des principales MVP que nous avons développées. Le concepteur peut personnaliser cet exemple qui est fourni avec le *template* de MATIP pour réaliser sa propre application ayant des tâches matérielles communicantes, il peut instancier par exemple une IP et la faire communiquer à l'aide des primitives de MPI-HCL. Dans la suite nous allons expliquer chaque portion du module Vhdl du démonstrateur MATIP.

Dans l'extrait de code 4.1, L'entité de la tâche utilisateur contient les signaux pour l'interface avec le TIC.

Listing 4.1: ht_process : en-tête et entité des tâches du démonstrateur MATIP

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;
library NocLib ;
use NocLib.CoreTypes.all;
Library MPI_HCL;
use MPI_HCL.MPI_RMA.all;
entity HT_process is
generic (Task_id : natural:=0);
Port ( clk : in STD_LOGIC;
reset : in STD_LOGIC;
```

```
  en : in std_logic; -- active la tâche
Interf_i : in core_i; --signaux pour l'interface IO
  Interf_o : out core_o; --signaux pour l'interface IO
mem_o : out typ_dpram_o; -- signaux pour l'accès à la mémoire
  mem_i : in typ_dpram_i -- signaux pour l'accès à la mémoire
  );
end HT_process;
--...
```

Dans l'extrait de code 4.2, la partie déclarative de l'architecture du module, montre tous les signaux et les types utilisés dans le process de traitement des communications.

Listing 4.2: ht_process : déclaration des signaux utilisés dans la HT du démonstrateur MATIP

```
architecture Behavioral of HT_process is

  type typ_mae is
      (start,Fillmem,NextFill,InitApp,GetRank,WInCreate,WinPost,WinStart,
  putdata,getdata,WinWait,WinCompleted,MpiSpawn,finalize,st_timeout);

   signal sram : typ_dpram;
   signal MyGroup:MPI_group;
   signal MyWin : MPI_win;
   signal MyRank :std_logic_vector(3 downto 0);
   signal intercomm,array_of_errcodes : natural;
   signal Libr : Core_io; --regroupe tous les signaux IO de la bibliothèque
   signal RunState : typ_mae;
--...
```

Dans l'extrait de code 4.3, la partie déclarative du *process* de communication `comm_proc` renferme des variables qui servent à gérer les arguments des MVP. La partie traitement commence par l'initialisation et l'affectation des signaux manipulées par les MVP.

Listing 4.3: ht_process : Initialisation et affectation des signaux

```
--...
begin
--============================================================================
--début du process de communication MPI
 comm_proc:process(clk,reset,en)

--partie déclarative du process
variable destrank : natural range 0 to 15;
variable timeout,ct,dlen,dcount : natural range 0 to 255;
```

```
variable SrcAdr,DestAdr : std_logic_vector(ADRLEN-1 downto 0);
variable mywin : MPI_win;
variable command , argv , maxprocs , info , root : natural:=0; --variable pour le
    spawn
variable adresse,adresse_rd :natural range 0 to 65536;

-- partie traitement du process
begin

if (rising_edge(clk) and en='1' then

if reset='1' then --le reset est synchrone
-- Initialisation des signaux du module de la tâche matérielle
      Libr.i.Instr_ack<='0';
      Libr.i.InitOk<='0';
      Libr.i.Hold_Req<='0';
      Libr.i.RamSel<='0';
      Libr.O.MemBusy<='0';
      Libr.O.Instr_En<='0';
      sram.O.we<='0';
      sram.O.ena<='0';
      sram.O.enb<='0';
      mem_o.addr_wr<=(others=>'1');
      mem_o.addr_rd<=(others=>'1');
      mem_o.we<='0';
      mem_o.ena<='0';
      mem_o.enb<='0';
      mem_o.data_in<=(others=>'0');
   --initialisation des entrées TIC
   Interf_o.Instr_EN<='0';
   Interf_o.Membusy<='0';
   Interf_o.Instruction<=(others=>'0');
      RunState<=start; --jump to the first state of the FSM
else
   --============================================================
   -- affectation des entrées sorties mémoires
   --These signals binding must remain unchanged for proper functionning of the
       template !
   sram.i.data_out<=mem_I.data_out;
   mem_o.addr_wr<=sram.O.addr_wr;
   mem_o.addr_rd<=sram.O.addr_rd ;
```

```
   mem_o.we<=sram.0.we;
   mem_o.ena<=sram.0.ena;
   mem_o.enb<=sram.0.enb;
   mem_o.data_in<=sram.0.data_in;
   --affectation des entrées/sortie TIC
   Interf_o.Instr_EN<=Libr.0.instr_en;
   Interf_o.Membusy<=Libr.0.MemBusy;
   Interf_o.Instruction<=Libr.0.Instruction;
   --affectation des sorties TIC
   Libr.i.Instr_ack<=Interf_i.Instr_ack;
   Libr.i.InitOk<=Interf_i.InitOk;
   Libr.i.Hold_Req<=Interf_i.Hold_req;
   Libr.i.RamSel<=Interf_i.RamSel;
--...
```

Dans l'extrait de code 4.4, le signal `RunState` identifie chaque état du processus de communication. l'état `start` permet de réserver une zone de stockage dans la mémoire de communication pour les données. Les états `Writedata` et `test_i` permettent de remplir cette zone avec des données de test. l'état initApp permet d'initialiser les communications en utilisant la MVP `pMPI_init()` enfin l'état GetRank permet de récupérer le rang de la tâche. Jusqu'ici, toutes les tâches suivent les mêmes étapes dans cette application. Les différences de comportement entre chaque tâche apparaîtront dans la suite du module.

Listing 4.4: ht_process : Initialisation des communications

```
--...
--============================================================
--gestion de l'état des communications
   case RunState is
   when start =>
      adresse:=10;dcount:=0;

      MPI_Alloc_mem(ct,Libr,Clk,SRam,200,MPI_INFO_NULL,adresse);
      --réserve 200 octets pour les données
      if ct=0 then
      RunState<=Writedata;
      end if;
When Writedata =>
      adresse:=adresse+i;
      Writemem(ct,Libr,sram,adresse,i);--écrire 0,1,2,3,4... à partir de l'adresse
          retournée
      if ct= 0 then
```

```
      RunState<=test_i;
   end if;
When test_i =>
      if i=100 then
         RunState<=InitApp;
      else
         i:=i+1;
         RunState<=Writedata;
      end if;
when InitApp => --la phase d'initialisation
      dlen:=15;

      pMPI_Init(ct,Libr,Clk,SRam);

      if ct=0 then
         RunState<=GetRank;
      end if;

when GetRank => --récupération du rang MPI

   pMPI_Comm_rank(ct,Libr,sram,MPI_COMM_WORLD,MyRank);

   if ct=0 then
      RunState<=WinCreate;

   end if;
--...
```

Dans l'extrait de code 4.5, l'état `Wincreate` sert à créer les groupes de tâches qui seront sollicitées par les opérations de communication. Les groupes sont créés en fonction du rang de la tâche. la tâche de rang 0 reçoit, d'après les transactions de la figure 4.13, un message de la tâche de rang 1 c'est pourquoi la variable `Mygroup.grp` est initialisée à la valeur `x0002`. Chaque bit de la variable `Mygroup.grp` indique le rang d'une tâche qui est sollicitée lorsqu'il est à '1'. Toutes les autres tâches communiquent avec la tâche de rang 1 ainsi leur variable `Mygroup.grp` est initialisée à `x"0001"`.

L'état `WinPost` permet d'appeler la MVP qui gère la réception des messages.

L'état `WinStart` permet d'appeler la MVP qui gère l'envoie des messages.

Listing 4.5: ht_process : Création des groupes et initialisation de la fenêtre de communication

```
--...
when Wincreate => --ici nous personnalisons les HT en fonction de leur rang
```

```
case MyRank is
when x"0"=>
  Mygroup.grp<=x"0002";-- rank 1 in this group ==> Win_Post
  MyGroup.nb<=2;
  destRank:=1;
when x"1" =>
  Mygroup.grp<=x"0001";-- rank 0 in this group
  destRank:=0;       -- not use by this HT instance
  DestAdr:=X"043F";
when x"2" =>
  Mygroup.grp<=x"0001";-- rank 0 in this group
  destRank:=0;
  DestAdr:=X"044F";
when x"3" =>
  Mygroup.grp<=x"0001";-- rank 0 in this group
  destRank:=0;   -- not use by this HT instance
  DestAdr:=X"045F";
when others =>
  Mygroup.grp<=x"0001";
  destRank:=0;
end case;
      RunState<=WinPost;
when WinPost =>

   pMPI_Win_Post(ct,Libr,sram,MyGroup,0,MyWin);
   if ct=0 then
   RunState<=WinStart;

   end if;

when WinStart =>
   pMPI_Win_start(ct,Libr,sram,MyGroup,0,MyWin);

   if ct=0 then
      RunState<=PutData;
   end if;
--...
```

Dans l'extrait de code 4.6, les états `putdata` et `getdata` utilisent les MVP pour les communications. Seule la tâche HT0 exécute la MVP `pMPI_GET()`.

Listing 4.6: ht_process : utilisation des MVP pour l'envoi et la réception des données

```
--...
when putdata => -- phase envoi des données
   SrcAdr:=std_logic_vector(to_unsigned(adresse,16));--X"000A";
   pMPI_put(ct,Libr,Clk,Sram,SrcAdr,Dlen,MPI_int,destrank, DestAdr, Dlen, MPI_int,
       Default_win);
   if ct=0 then
      RunState<=GetData;
   end if;


when getdata =>
   SrcAdr:=std_logic_vector(to_unsigned(adresse+100,16));
   destrank:=1;
   if unsigned(MyRank) = 0 then --réception des données par HT 0 seulement
      pMPI_GET(ct,Libr,Clk,Sram,SrcAdr,Dlen,MPI_int, destrank, DestAdr, Dlen,
          MPI_int, Default_win);
   end if;

   if ct=0 then

      RunState<=wincompleted;

   end if;
--...
```

Dans l'extrait de code 4.7, les états `WinCompleted` et `WinWait` sont utilisés pour synchroniser les transferts. A la fin de l'exécution de la MVP `pMPI_Win_wait()` l'état `MpiSpawn` est activé en fonction de la nature statique ou dynamique de la tâche matérielle ou d'un précédent appel de la MVP.

Listing 4.7: ht_process : Synchronisation de l'envoi et de la réception

```
--...
when WinCompleted => --Fin des transferts côté source

    pMPI_Win_Complete(ct,Libr,sram,MyWin );
   if ct=0 then
      RunState<=WinWait;

   end if;
when WinWait => --Fin des transferts côté cible
```

```
    pMPI_Win_wait(ct,Libr,sram,MyWin );

   if ct=0 then
      if Libr.I.Spawned='1' then
         RunState<=Finalize; --cas de HT dynamiques
      elsif spawn_on='1' then
         RunState<=Finalize; -- cas de spawn déjà appelé par HT0
      else
         RunState<=MpiSpawn; --appel de spawn par HT 0 et HT1
      end if;
   end if;
--...
```

Dans l'extrait de code 4.8, l'état `MpiSpawn` utilise la MVP `MPI_Comm_modify()` pour charger et activer deux HT dynamiques. la HT0 retourne à l'état `WinPost` pour initialiser la réception des données venant de HT2 et HT3 tandis que la HT1 passe à l'état `finalize` qui est l'état de fin des transactions pour une tâche.

Listing 4.8: Description VHDL des tâches statiques du démonstrateur MATIP

```
--...
when MpiSpawn => --Phase appel des tâches dynamiques

   maxprocs:=2; --nombre de HT
   pMPI_Comm_modify(ct,Libr,sram,command,argv,maxprocs, info, root, comm,
       intercomm, array_of_errcodes);

   if ct=0 then
      if unsigned(MyRank)=0 then
         Spawn_on:='1'; --signale qu'un spawn a eu lieu
          Mygroup.grp<=x"000C";-- code pour rank 3,4
            destRank:=0;
         runstate<=WinPost; --la tâche HT0 attend des données
      else
               RunState<=Finalize; --la tâche HT1 se termine directement
      end if;
   end if;
when finalize =>
   pMPI_finalize(ct,Libr,Clk,SRam);
      if ct=0 then
         RunState<=finalize;
      end if;
when others =>
```

```
   RunState<=start;
end case;

end if;
end if;

end process comm_proc;

end Behavioral;
```

Le code Vhdl du module `ht_process` que nous venons de présenter démontre la facilité d'utilisation des MVP pour ajouter des primitives de communications à une tâche matérielle ou pour charger dynamiquement une tâche à partir du langage VHDL. Pour déployer une application ayant des tâches matérielles, le concepteur peut modifier les différents états du signal `RunState` pour décrire le comportement de chaque tâche matérielle.

## 4.5 Conclusion

Dans ce chapitre, nous avons étendu les capacités de la plateforme MATIP afin de déployer dynamiquement des tâches matérielles. Nous avons présenté et mis en œuvre la primitive `MPI_Comm_modify()` du standard MPI-RMA qui permet au concepteur d'une application parallèle de charger une ou plusieurs tâches de façon dynamique. Nous avons utilisé le flot de conception de Xilinx pour mettre en œuvre la reconfiguration dynamique partielle des FPGAS Xilinx et avons intégré à la plateforme MATIP une primitive qui configure le FPGA Xilinx Artix 7 lorsque la primitive Vhdl `MPI_Comm_modify()` est invoquée.

La reconfiguration du FPGA à partir de l'appel d'une MVP apporte un comportement dynamique au langage VHDL. Les résultats de latence des différentes primitives de la plateforme MATIP ont été présentés en utilisant une application de démonstration. Lorsqu'on compare l'architecture MATIP à l'architecture FOSFOR, on note que les deux plateformes permettent de déployer dynamiquement des tâches matérielles, toutefois, FOSFOR utilise un système d'exploitation qui occupe deux fois plus de ressources que la plateforme MATIP. Les ressources utilisées par le middleware de FOSFOR pour les communications et le wrapper pour intégrer les tâches matérielles (HW threads) sont 10 % plus imporantes que les ressources utilisées par le TIC de MATIP et la couche de communication. L'utilisation de la mémoire partagée sur FOSFOR donne de bonne performances temporelles mais rend le passage à l'échelle plus complexe que sur MATIP qui utilise un système à mémoire distribuée. La plateforme MATIP occupe moins de ressources sur le FPGA que la plateforme FOSFOR.

Le délai de reconfiguration d'une tâche est fonction de la taille des données de configuration, des performances de l'unité de stockage des données de configurations et d'un délai

de synchronisation de la primitive *Spawn* qui dure environ deux cent cycles d'horloge. Les performances temporelles des primitives de communication MPI de MATIP sont meilleures comparées à celles des autres plateformes MPI que nous avons trouvées dans la littérature (voir tableau 3.8 en page 104). La plateforme MATIP permet de concevoir et déployer de façon autonome des tâches matérielles utilisant la reconfiguration dynamique et le langage VHDL.

# Conclusion générale

Dans cette thèse nous avons étudié le déploiement d'application parallèle sur un MP-RSoC. Nous avons proposé dans ce cadre une plateforme à mémoire distribuée que nous avons appelée MATIP. MATIP est structurée en trois couches qui peuvent être améliorées indépendamment les unes des autres. MATIP permet de déployer des applications parallèles conçues à l'aide de tâches matérielles dont certaines utilisent la reconfigurabilité du MP-RSoC. Pour réaliser MATIP, nous avons conçu un processeur de communication nommé MPI-HCL qui met en œuvre le standard MPI-RMA et qui contribue à la réduction des efforts nécessaires pour développer une application parallèle en environnement MP-RSoC. MPI-HCL met en œuvre une version matérielle de primitives de communication essentielles de la bibliothèque MPI version 2. Ces primitives permettent d'effectuer des communications non bloquantes entre tâches et de charger dynamiquement une tâche pendant l'exécution de l'application. Une version de MATIP supportant 16 tâches a été réalisée et nous avons déployé une application de démonstration pour valider le fonctionnement de la plateforme sur un FPGA Xilinx Artix-7. MATIP fournit aux concepteurs d'applications parallèles une plateforme avec un *template* pour intégrer des IPs sans se préoccuper des détails d'interconnexion et de reconfiguration dynamique du MP-RSoC. Avec MATIP, le concepteur peut appliquer l'approche *Platform Based Design* en construisant son application à l'aide d'un langage de conception tel que le VHDL. La couche de communication de MATIP prend en charge de façon transparente la reconfiguration du FPGA pour ajouter de nouvelles tâches à l'application, donnant au concepteur VHDL la capacité de décrire la dynamicité dans son application. Le projet TMD-MPI a montré que MPI pouvait être utilisé comme une API standard pour développer des applications pour HPRC[7], MATIP a prouvé la même chose avec le MP-RSoC et va plus loin en activant la reconfiguration dynamique partielle du MP-RSoC à partir des primitives MPI-2 RMA. L'overhead de MATIP pour la gestion des tâches matérielles communicantes est 10 % moins important que celui de la plateforme réalisée lors du projet FOSFOR.

7. High Performance Reconfigurable Computer

# Perspectives

Pour compléter ce travail, il est envisagé dans la suite de ce projet de réaliser une version logicielle de MPI-HCL afin d'avoir un environnement de déploiement d'applications ayant des tâches logicielles et des tâches matérielles. Cette extension apportera davantage de flexibilité.

Nous avons vu que l'interconnexion est responsable de la limitation en nombre de tâches matérielles gérable par MATIP. Nous envisageons d'étendre notre NoC en un réseau multi-étage de type arbre comme celui de la figure 4.17. Cette structure consomme moins de ressources et permet de connecter plus de tâches matérielles.

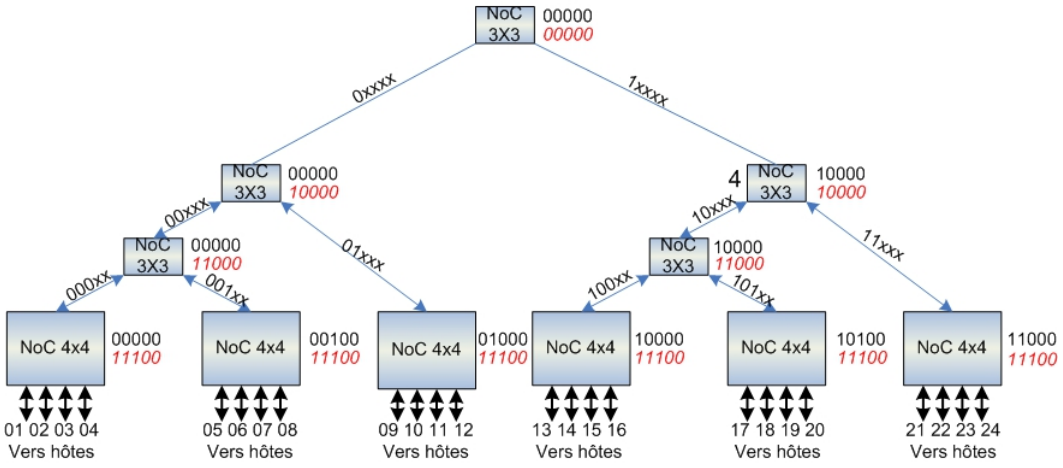



FIGURE 4.17 : NoC hierarchique

Pour faciliter la génération des codes Vhdl du *template* de notre plateforme, un outil de configuration d'IP comme CoreGenerator de Xilinx ISE pourra être proposé pour MPI-HCL. Ce travail permet également d'envisager la construction d'un *Cloud SoC* réalisé à partir de multiples puces FPGA. Dans une telle architecture, le présent projet sera considéré comme une brique initiale. La figure 4.18 illustre ce que pourrait être cette architecture. Cette architecture pourra apporter une meilleur gestion de l'énergie car il sera envisageable d'arrêter complètement un FPGA et de l'activer uniquement lorsque cela sera nécessaire.

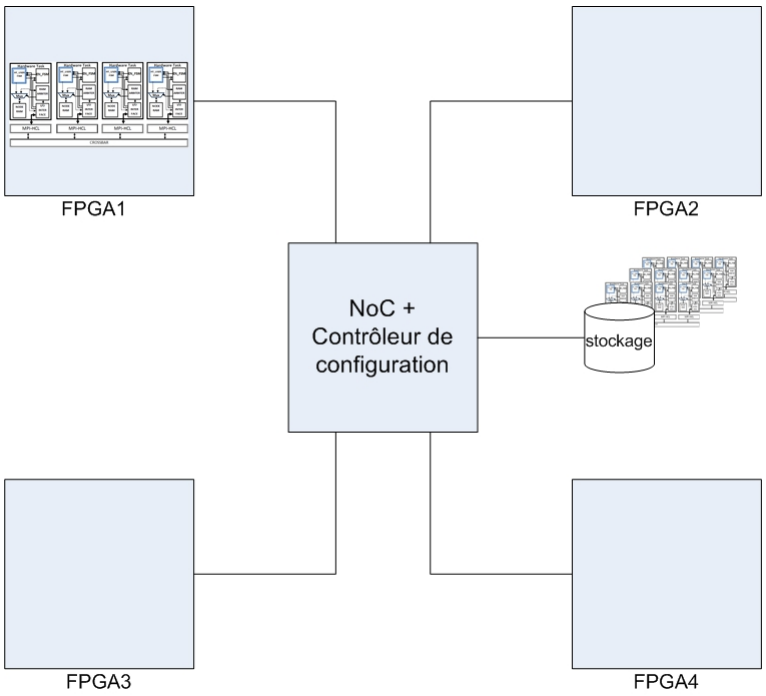



FIGURE 4.18 : Architecture Cloud SoC

# Publications

1. GAMOM NGOUNOU EWO, Roland Christian, KIEGAING, Emmanuel, MBOUENDA, Martin, et al. Hardware mpi-2 functions for multi-processing reconfigurable system on chip. In : *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International.* IEEE, 2013. p. 273-280.
2. GAMOM NGOUNOU EWO, R., PINNA, Andrea, GRANADO, Bertrand, et al. A Hardware MPI Spawn for Distributed Multiprocessing Reconfigurable System on Chip (MP-RSoC). In : *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on.* IEEE, 2014. p. 238-238.
3. Implémentation d'une primitive MPI Spawn pour le déploiement dynamique de tâches dans un Système multiprocesseurs sur puce reconfigurable (MP-RSoC) à mémoire distribuée, Gamom Ngounou Ewo Roland Christian, Granado Bertrand, Pinna Andrea, Fotsin Bertrand Hilaire, Mbouenda Martin *colloque GDR SoC SiP* 2014

# Annexe A

# Codes sources

## A.1 Code source en langage C du programme send_prg

Listing A.1: Code source C du programme send_prg

```
#include "mpi.h"
#include "stdio.h"

/* test de put post/start/complete/wait avec deux instances */

#define SIZE1 10
#define SIZE2 10

int main(int argc, char *argv[])
{
    int rank, destrank, nprocs, i;
   char *A, *B;
    MPI_Group comm_group, group;
    MPI_Win win;
    int errs = 0;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if (nprocs != 2) {
        printf("Attention il faut 2 instances du programme\n");fflush(stdout);
        MPI_Abort(MPI_COMM_WORLD,1);
    }

    i = MPI_Alloc_mem(SIZE2 * sizeof(char), MPI_INFO_NULL, &A);
```

```
if (i) {
    printf("Impossible d'allouer la mémoire\n");fflush(stdout);
    MPI_Abort(MPI_COMM_WORLD, 1);
}
i = MPI_Alloc_mem(SIZE2 * sizeof(char), MPI_INFO_NULL, &B);
if (i) {
    printf("Impossible d'allouer la mémoire\n");fflush(stdout);
    MPI_Abort(MPI_COMM_WORLD, 1);
}

MPI_Comm_group(MPI_COMM_WORLD, &comm_group);

if (rank == 0) {
    for (i=0; i<SIZE2; i++) A[i] = i;
    MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &win);
    destrank = 1;
    MPI_Group_incl(comm_group, 1, &destrank, &group);
    MPI_Win_start(group, 0, win);
    for (i=0; i<SIZE1; i++)
        MPI_Put(A+i, 1, MPI_INT, 1, i, 1, MPI_CHAR, win);

    MPI_Win_complete(win);
}
else { /* rank=1 */
    for (i=0; i<SIZE2; i++) B[i] = (-4)*i;
    MPI_Win_create(B, SIZE2*sizeof(int), sizeof(int), MPI_INFO_NULL,
        MPI_COMM_WORLD, &win);
    destrank = 0;
    MPI_Group_incl(comm_group, 1, &destrank, &group);
    MPI_Win_post(group, 0, win);
    MPI_Win_wait(win);

    for (i=0; i<SIZE1; i++) {
        if (B[i] != i) {
            printf("Erreur: B[i] est %d, au lieu de %d\n", B[i],
                i);fflush(stdout);
            errs++;
        }
    }
}
```

```
    MPI_Group_free(&group);
    MPI_Group_free(&comm_group);
    MPI_Win_free(&win);
    MPI_Free_mem(A);
    MPI_Free_mem(B);

    MPI_Finalize();
    return errs;
}
```

## A.2 Constantes associées à la zone de transfert des instructions

Listing A.2: Déclaration des constantes associées à la zone de transfert des instructions

```
CONSTANT CORE_BASE_ADR :natural range 0 to 65535:=4096; --l'adresse de base de la
    zone de registres du composant MPI (0x1000)
CONSTANT core_INIT_ADR : natural:=CORE_BASE_ADR+512; --l'adresse àpartir de
    laquelle les paramètres de la MVP MPI_INIT et les valeurs de retour sont
    stockées (0x1200)
CONSTANT CORE_PUT_ADR :natural:=CORE_BASE_ADR+516;--l'adresse àpartir de laquelle
    les paramètres de la MVP MPI_PUT et les valeurs de retour sont stockées
    (0x1204)
CONSTANT CORE_GET_ADR:natural:=CORE_BASE_ADR+526; --l'adresse àpartir de laquelle
    les paramètres de la MVP MPI_GET et les valeurs de retour sont stockées
    (0x120E)
CONSTANT CORE_WCREATE_ADR : natural := CORE_BASE_ADR+536;--l'adresse àpartir de
    laquelle les paramètres de la MVP MPI_WIN_CREATE et les valeurs de retour sont
    stockées
CONSTANT CORE_WCOMPL_ADR : natural := CORE_BASE_ADR+546;--l'adresse àpartir de
    laquelle les paramètres de la MVP MPI_WIN_COMPLETE et les valeurs de retour
    sont stockées
CONSTANT CORE_WPOST_ADR : natural := CORE_BASE_ADR+556;
CONSTANT CORE_WWAIT_ADR : natural := CORE_BASE_ADR+566;
CONSTANT CORE_SPAWN_ADR : natural := CORE_BASE_ADR+576;--l'adresse àpartir de
    laquelle les paramètres de la MVP MPI_WIN_SPAWN et les valeurs de retour sont
    stockées
```

## A.3 Code source du programme send_prg Vhdl

Listing A.3: Code source Vhdl du programme send_prg

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;
library NocLib ;
use NocLib.CoreTypes.all;
Library MPI_HCL;
use MPI_HCL.MPI_RMA.all;
--...
entity HT_stat is
      generic (Task_id : natural:=0);
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           en : in std_logic; -- active la tâche
           Interf_i : in core_i; --signaux pour l'interface d'entrée
           Interf_o : out core_o; --signaux pour l'interface de sortie
           mem_o : out typ_dpram_o; -- signaux pour l'accès àla mémoire(sortie)
           mem_i : in typ_dpram_i -- signaux pour l'accès àla mémoire (entrée)
           );
end HT_dyn;
architecture Behavioral of HT_stat is
   --déclarer les diférents états du programme
  type typ_mae is (start, Writedata_0, Writedata_1, Init, GetRank,
      WInCreate,test_i_0,test_i_1, test_i_2, WinPost, WinStart, put_0, WinWait,
      WinCompleted, MpiSpawn, finalize, freeMem);

signal MyGroup:mpi_group;
signal MyWin : mpi_win;

signal ct_state:natural:=0; --Compteur d'état interne pour les MVPs
signal Libr : Core_io; --regroupe tous les signaux IO de la bibliothèque
signal sram : typ_dpram; --type Dual port Ram défini dans le package Corestypes
signal RunState : typ_mae; --déclarer le signal de parcours des différents états
    du programme
--...
send_prg:process(clk)
--
variable i : natural:=0; --compteur de boucle
variable adr : std_logic_vector(15 downto 0);
```

```
begin
if rising_edge(clk) then

case RunState is
when start =>

   MPI_Alloc_mem(ct,Libr,Clk,SRam,10,MPI_INFO_NULL,adr); --reserve 10 octets pour
       les données
   if ct=0 then
   RunState<=Init;
   end if;
when Init =>

pMPI_Init(ct,Libr,Clk,SRam);
if ct= 0 then
  RunState<=GetRank;--success move to next step
end if;
when GetRank =>
pMPI_Comm_rank(ct, Libr, sram, MPI_COMM_WORLD, MyRank);
   if ct= 0 then

      RunState<=Wincreate;

   end if;

end if;
when Wincreate => --par défaut la fenêtre 0 est utilisée !
       case MyRank is --le rang détermine le rôle spécifique de chaque tâche dans
           une application SPMD
         when x"0" =>
              Mygroup.grp<=X"0002";-- rang 1 dans le groupe
              MyGroup.nb<=2;
              destRank:=1;
              DestAdr:=X"043F";
              nb_rma:=1;
               RunState<=WriteData; --le rôle source
      when others =>
           Mygroup.grp<=x"0001";-- rang 0 dans le groupe
           destRank:=0;
           DestAdr:=X"043F";
            RunState<=WriteData; -- le rôle cible
```

```
      end case;

When Writedata_0 =>
      adr:=adr+i;
      Writemem(ct,Libr,sram,adr,i);--écrire 0,1,2,3,4...9 àpartir de l'adresse adr
      if ct= 0 then
      RunState<=test_i;
   end if;
When test_i_0 =>
      if i=10 then
         RunState<=Put_0;
      else
         i:=i+1;
         RunState<=Writedata_0;
      end if;
When Writedata_1 =>
      adr:=adr+i;
      Writemem((ct,Libr,sram,adr,0); --écrire 0 àtoutes les adresses
      if ct= 0 then
      RunState<=test_i;
   end if;
When test_i_1 =>
      if i=10 then
         RunState<=WinPost;
      else
         i:=i+1;
         RunState<=Writedata_1;
      end if;

when WinPost =>

   pMPI_Win_Post(ct,Libr,sram,MyGroup,0,MyWin);
   if ct= 0 then
      RunState<=WinWait;
   end if;
when WinStart =>

   pMPI_Win_Start(ct,Libr,sram,MyGroup,0,MyWin);
   if ct= 0 then
      RunState<=Put_0;
      i:=0;
```

```
   end if;

when put_0 =>
src_adr:=adr+i;dest_adr:=200+i;
pMPI_put(ct,Libr,Clk,Sram,src_adr,1,MPI_char,0, dest_adr,1,Mpi_char,Default_win);
   if ct= 0 then
      RunState<=test_i_2;
   end if;

When test_i_2 =>
      if i=10 then
         RunState<=WinComplete;
      else
         i:=i+1;
         RunState<=put_0;
      end if;
when WinComplete =>
   pMPI_Win_Complete(ct,Libr,sram,MyWin );
   if ct=0 then
      RunState<=finalize;
   end if;
when Winwait=> --attente de la fin des transferts
pMPI_Win_Wait(ct,Libr,sram,MyWin );
   if ct= 0 then
      RunState<=finalize;
   end if;
when FreeMem=> --libère le bloc mémoire de données
pMPI_free_mem(ct,Libr,sram,adr );
   if ct= 0 then
      RunState<=finalize;
   end if;
when finalize =>
--
RunState<=finalize;
end case;
end if;
end process Put_Get;
```

## A.4 Exemple de description d'une tâche matérielle faisant appel à des tâches dynamiques

Listing A.4: Exemple de description d'une tâche dans ht_stat.vhd

```
entity HT_process is
generic (Task_id : natural:=0);
Port ( clk : in STD_LOGIC;
reset : in STD_LOGIC;
  en : in std_logic; -- active la tâche
Interf_i : in core_i; --signaux pour l'interface IO
  Interf_o : out core_o; --signaux pour l'interface IO
mem_o : out typ_dpram_o; -- signaux pour l'accès á la mémoire
  mem_i : in typ_dpram_i -- signaux pour l'accès à la mémoire
  );
end HT_process;

architecture Behavioral of HT_process is

  type typ_mae is
     (start,Fillmem,NextFill,InitApp,GetRank,WInCreate,WinPost,WinStart,
  putdata,getdata,WinWait,WinCompleted,MpiSpawn,finalize,st_timeout);
   signal sram : typ_dpram;
   signal MyGroup:MPI_group;
   signal MyWin : MPI_win;
   signal MyRank :std_logic_vector(3 downto 0);
   signal intercomm,array_of_errcodes : natural;
signal Libr : Core_io; --regroupe tous les signaux IO de la bibliothèque
signal RunState : typ_mae;

begin

--MAE du PE
--=============================================================================

 pPutGet:process(clk,reset,en)


variable destrank : natural range 0 to 15;
variable timeout,ct,dlen,dcount : natural range 0 to 255;
variable SrcAdr,DestAdr : std_logic_vector(ADRLEN-1 downto 0);
```

```
variable mywin : MPI_win;
variable command , argv , maxprocs , info , root : natural:=0; --variable pour le
    spawn
variable adresse,adresse_rd :natural range 0 to 65536;

begin

if (clk'event and clk='1') and en='1' then


if reset='1' then
--set the default signals values

      Libr.O.MemBusy<='0';
      Libr.O.Instr_En<='0';
      sram.O.we<='0';
      sram.O.ena<='0';
      sram.O.enb<='0';
      RunState<=start; --jump to the first state of the FSM
else
   --============================================================
   -- affectation des entrées sorties mémoires
   --These signals binding must remain unchanged for proper functionning of the
       template !
   sram.i.data_out<=mem_I.data_out;
   mem_o.addr_wr<=sram.O.addr_wr;
   mem_o.addr_rd<=sram.O.addr_rd ;
   mem_o.we<=sram.O.we;
   mem_o.ena<=sram.O.ena;
   mem_o.enb<=sram.O.enb;
   mem_o.data_in<=sram.O.data_in;
   --affectation des entrées sorties MPI_HCL
   Interf_o.Instr_EN<=Libr.O.instr_en;
   Interf_o.Membusy<=Libr.O.MemBusy;
   Interf_o.Instruction<=Libr.O.Instruction;
   Libr.i.Instr_ack<=Interf_i.Instr_ack;
   Libr.i.InitOk<=Interf_i.InitOk;
   Libr.i.Hold_Req<=Interf_i.Hold_req;
   Libr.i.RamSel<=Interf_i.RamSel;
   --============================================================
```

```
   case RunState is
   when start =>
      adresse:=10;dcount:=0;
      RunState<=Fillmem;
    when Fillmem => --this state demonstrate a simple example task that writes
        data in the communication RAM
         sram.O.we<='1'; --enables writting
         sram.O.ena<='1';
         sram.O.enb<='0'; --disable reading on port B
      if Libr.I.Ramsel='0' then  --test if the RAM is free
         sram.O.addr_wr<=std_logic_vector(to_unsigned(adresse,ADRLEN));
         sram.O.data_in<=std_logic_vector(to_unsigned(dcount,8)); -- x"0f";
         Libr.O.MemBusy<='1'; --prevent the MPI-HCL to gain control on the RAM
         dcount:=dcount+1;

         if dcount=250 then
              RunState<=InitApp;
            Libr.O.MemBusy<='0'; --release the RAM access
         else
            adresse:=adresse+1;
            RunState<=Fillmem;
         end if;
   else -- attente de la libéraion de la mémoire/waiting for Ram to be free
         timeout:=timeout+1;
           if timeout=200 then
            report "The RAM is been busy for too long!!!";
            timeout:=0;
           end if;

   end if;

when InitApp =>
      dlen:=15;

      pMPI_Init(ct,Libr,Clk,SRam);

      if ct=0 then
         RunState<=GetRank;
      end if;

when GetRank =>
```

```
   pMPI_Comm_rank(ct,Libr,sram,MPI_COMM_WORLD,MyRank);

   if ct=0 then
      RunState<=WinCreate;

   end if;

when Wincreate => --ici nous personalisons les HTs en fonction de leur rang
--we use this state to fix HT parameters in regard of thier rank
case MyRank is
when x"0"=>
  Mygroup.grp<=x"0002";-- rank 1 in this group ==> Win_Post
  MyGroup.nb<=2;
  destRank:=1;
when x"1" =>
  Mygroup.grp<=x"0001";-- rank 0 in this group
  destRank:=0;       -- not use by this HT instance
  DestAdr:=X"043F";
when x"2" =>
  Mygroup.grp<=x"0001";-- rank 0 in this group
  destRank:=0;
  DestAdr:=X"044F";
when x"3" =>
  Mygroup.grp<=x"0001";-- rank 0 in this group
  destRank:=0;   -- not use by this HT instance
  DestAdr:=X"045F";
when others =>
  Mygroup.grp<=x"0001";
  destRank:=0;
end case;
       RunState<=WinPost;
when WinPost =>

   pMPI_Win_Post(ct,Libr,sram,MyGroup,0,MyWin);
   if ct=0 then
   RunState<=WinStart;

   end if;

when WinStart =>
```

```
   pMPI_Win_start(ct,Libr,sram,MyGroup,0,MyWin);

   if ct=0 then
      RunState<=PutData;
   end if;
when putdata => --
   SrcAdr:=X"000A";
   pMPI_put(ct,Libr,Clk,Sram,SrcAdr,Dlen,MPI_int, destrank, DestAdr, Dlen,
       MPI_int, Default_win);
   if ct=0 then
      RunState<=GetData;
   end if;


when getdata =>
   SrcAdr:=X"006D";
   destrank:=1;
   if unsigned(MyRank) = 0 then
      pMPI_GET(ct,Libr,Clk,Sram,SrcAdr,Dlen,MPI_int, destrank, DestAdr, Dlen,
          MPI_int, Default_win);
   end if;

   if ct=0 then

      RunState<=wincompleted;

   end if;

when WinCompleted =>

    pMPI_Win_Complete(ct,Libr,sram,MyWin );
   if ct=0 then
      RunState<=WinWait;

   end if;
when WinWait =>

    pMPI_Win_wait(ct,Libr,sram,MyWin );

   if ct=0 then
      if Libr.I.Spawned='1' then
```

```
        RunState<=Finalize; --cas de processus dynamique
      elsif spawn_on='1' then
        RunState<=Finalize; -- cas de de spawn déjà appelé par HT0
      else
        RunState<=MpiSpawn; --appel de spawn par HT 0 et HT1
      end if;
   end if;
when MpiSpawn =>

   maxprocs:=2;
   pMPI_Comm_Spawn(ct,Libr,sram,command,argv,maxprocs, info, root, comm,
       intercomm, array_of_errcodes);

   if ct=0 then
      if unsigned(MyRank)=0 then
        Spawn_on:='1'; --signal qu'un spawn a eu lieu
         Mygroup.grp<=x"000C";-- rank 3,4
           destRank:=0;
        runstate<=WinPost; --la tâche HT0 attend des données
      else
              RunState<=Finalize; --l'autre tâche HT1 se terminent directement
      end if;
   end if;
when finalize =>

      if ct=0 then
        RunState<=finalize;
      end if;
when others =>
   RunState<=start;
end case;

end if;
end if;

  end process pPutGet;

end Behavioral;
```

# Bibliographie


[1] ITRS et TWG, "The international technology roadmap for semiconductors (itrs) : 2013 edition", *White paper*, 2013. adresse : `http://www.itrs.net/Links/2013ITRS/2013Chapters/2013Overview.pdf`.

[2] I. Xilinx. (2015). All programmable 7 series product selection guide, Xilinx Inc, adresse : `http://www.xilinx.com/publications/prod_mktg/7-series-product-selection-guide.pdf`.

[3] W. H. W. Tuttlebee, *Software Defined Radio : Baseband Technologies for 3G Handsets and Basestations*, en. John Wiley & Sons, fév. 2006, isbn : 9780470867716.

[4] R. David, D. Chillet, S. Pillement et O. Sentieys, "A dynamically reconfigurable architecture for low. power multimedia terminals", *SOC Design Methodologies*, t. 90, p. 51, 2002.

[5] E. Mirsky et A. DeHon, "Matrix : a reconfigurable computing architecture with configurable instruction distribution and deployable resources", in *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, IEEE, 1996, p. 157–166.

[6] Wikipedia, *Google glass — wikipedia, the free encyclopedia*, [Online ; accessed 2-January-2015], 2014. adresse : `http://en.wikipedia.org/w/index.php?title=Google_Glass&oldid=638346255`.

[7] I. Mhedhbi, F. Kaddouh, K. Hachicha, D. Heudes, S. Hochberg et P. Garda, "Mask motion adaptive medical image coding", in *Biomedical and Health Informatics (BHI), 2014 IEEE-EMBS International Conference on*, IEEE, 2014, p. 408–411.

[8] M. Gries et K. Keutzer, *Building ASIPs : The Mescal Methodology : The MESCAL Methodology*. Springer, 2006.

[9] R. Dobai et L. Sekanina, "Towards evolvable systems based on the xilinx zynq platform", in *Evolvable Systems (ICES), 2013 IEEE International Conference on*, IEEE, 2013, p. 89–95.

[10] Z. Or-Bach. (). Is the cost reduction associated with scaling over ?, adresse : `http://www.monolithic3d.com/2/post/2012/06/is-the-cost-reduction-associated-with-scaling-over.html` (visité le 13/11/2014).

[11] S. Pillement, "Dynamically reconfigurable architectures : from silicon to system management", Habilitation à diriger des recherches, Université Rennes 1, oct. 2010. adresse : https://tel.archives-ouvertes.fr/tel-00554210.

[12] A. Greenfield, *Everyware : The dawning age of ubiquitous computing*. New Riders, 2010.

[13] T. Steve, "Beyong moore. beyong programmable logic", *Keynote FPL2012 Oslo, Norway, Aug. 29-31, 2012*, 2012. adresse : http://www.fpl2012.org/Presentations/Keynote_Steve_Trimberger.pdf.

[14] R. Hartenstein, "Coarse grain reconfigurable architecture (embedded tutorial)", in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ACM, 2001, p. 564–570.

[15] R. Hartenstein, M. Herz, T. Hoffmann et U. Nageldinger, "Kressarray xplorer : a new cad environment to optimize reconfigurable datapath array architectures", in *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, IEEE, 2000, p. 163–168.

[16] R. David, D. Chillet, S. Pillement et O. Sentieys, "Dart : a dynamically reconfigurable architecture dealing with future mobile telecommunications constraints", ENSSAT-LASTI - University of Rennes, Résumé, 2003.

[17] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach et M. Weinhardt, "Pact xpp—a self-reconfigurable data processing architecture", *The Journal of Supercomputing*, t. 26, n° 2, p. 167–184, 2003.

[18] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh et E. M. Chaves Filho, "Morphosys : an integrated reconfigurable system for data-parallel and computation-intensive applications", *Computers, IEEE Transactions on*, t. 49, n° 5, p. 465–481, 2000.

[19] J. Becker, T. Pionteck et M. Glesner, "Dream : a dynamically reconfigurable architecture for future mobile communication applications", in *Field-Programmable Logic and Applications : The Roadmap to reconfigurable Computing*, Springer, 2000, p. 312–321.

[20] T. Miyamori et K. Olukotun, "Remarc : reconfigurable multimedia array coprocessor", *IEICE Transactions on information and systems*, t. 82, n° 2, p. 389–397, 1999.

[21] L. Gantel, S. Layouni, M Benkhelifa, F. Verdier et S. Chauvet, "Multiprocessor task migration implementation in a reconfigurable platform", in *Reconfigurable Computing and FPGAs, 2009. ReConFig'09. International Conference on*, IEEE, 2009, p. 362–367.

[22] V. Alliance, *Virtual component interface standard (ocb 2 1.0) on-chip bus development working group*, 2000. adresse : http://www.vsi.org..

[23] *Ocp 3.0 specification*, 2014. adresse : http://www.accellera.org/downloads/standards/.

[24] T. Dorta, Jim, J. Nez, Mart, J. N, Luis, U. Bidarte et A. Astarloa, "Reconfigurable multiprocessor systems : a review", en, *International Journal of Reconfigurable Computing*, t. 2010, e570279, déc. 2010, issn : 1687-7195. doi : 10.1155/2010/570279. adresse : http://www.hindawi.com/journals/ijrc/2010/570279/abs/ (visité le 03/06/2014).

[25] X. Inc, *Partial reconfiguration user guide ug702 (v14.1)*, Xilinx Inc, 2012. adresse : http://www.xilinx.com/tools/partial-reconfiguration.htm.

[26] M. Flynn, "Some computer organizations and their effectiveness", *Computers, IEEE Transactions on*, t. C-21, n° 9, p. 948–960, 1972, issn : 0018-9340. doi : 10.1109/TC.1972.5009071.

[27] A. Burns et A. J. Wellings, *Real-time systems and programming languages : Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[28] M. Sato, "Openmp : parallel programming api for shared memory multiprocessors and on-chip multiprocessors", in *System Synthesis, 2002. 15th International Symposium on*, 2002, p. 109–111.

[29] J. Diaz, C. Munoz-Caro et A. Nino, "A survey of parallel programming models and tools in the multi and many-core era", *Parallel and Distributed Systems, IEEE Transactions on*, t. 23, n° 8, p. 1369–1386, 2012.

[30] S. A. Christe, M Vignesh et A Kandaswamy, "An efficient fpga implementation of mri image filtering and tumor characterization using xilinx system generator", *ArXiv preprint arXiv :1201.2542*, 2012.

[31] P. K. Bandyopadhyay, A. Biswas, P. K. Bandyopadhyay, D. Mandal et R. Kar, "Fpga based high frequency noise elimination system from speech signal using xilinx system generator", *Journal of Electron Devices*, t. 17, p. 1423–1426, 2013.

[32] L. Zhang, M. Glab, N. Ballmann et J. Teich, "Bridging algorithm and esl design : matlab/simulink model transformation and validation", in *Specification & Design Languages (FDL), 2013 Forum on*, IEEE, 2013, p. 1–8.

[33] O. S. Initiative et al., "Ieee standard systemc language reference manual", *IEEE Computer Society*, 2006.

[34] D. Dye, *Partial reconfiguration of xilinx fpgas using ise design suite*, Xilinx Inc., 2012. adresse : http://www.xilinx.com/tools/partial-reconfiguration.htm.

[35] M. Bourgeault, *Alteras partial reconfiguration flow*, Altera Inc., 2011. adresse : http://www.eecg.utoronto.ca/~jayar/FPGAseminar/FPGA_Bourgeault_June23_2011.pdf.

[36] P. J. Ashenden et J. Lewis, *VHDL-2008 : just the new stuff*. Morgan Kaufmann, 2007.

[37] D. E. Ott et T. J. Wilderotter, *A Designer's Guide to VHDL Synthesis*, 1st. Springer Publishing Company, Incorporated, 2010, isbn : 1441951431, 9781441951434.

[38] K. Cameron, "Dynamic connectivity", VHDL Analysis et Standardization Group, Proposal, 2013. adresse : http://www.eda.org/twiki/bin/view.cgi/P1076/DynamicRewiring.

[39] C. Steve et A. Peter, *Dynamic hardware construct*, 2003. adresse : http://www.eda.org/isac/IRs-VHDL-2002/IR2021.txt.

[40] G. Martin, B. Bailey et A. Piziali, *ESL design and verification : a prescription for electronic system level methodology*. Morgan Kaufmann, 2010.

[41] A. Otero, E. de la Torre et T. Riesgo, "Dreams : a tool for the design of dynamically reconfigurable embedded and modular systems", in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, IEEE, 2012, p. 1–8.

[42] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson et B. Hutchings, "Rapid prototyping tools for fpga designs : rapidsmith", in *Field-Programmable Technology (FPT), 2010 International Conference on*, IEEE, 2010, p. 353–356.

[43] S. Bayar et A. Yurdakul, "Self-reconfiguration on spartan-iii fpgas with compressed partial bitstreams via a parallel configuration access port (cpcap) core", in *Research in Microelectronics and Electronics, 2008. PRIME 2008. Ph.D.*, 2008, p. 137–140. doi : 10.1109/RME.2008.4595744.

[44] D. Koch, C. Beckhoff et J. Teich, "Recobus-builder ; a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas", in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, 2008, p. 119–124. doi : 10.1109/FPL.2008.4629918.

[45] D. Koch, C. Beckhoff et J. Teich, *Recobus-builder—a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas*, IEEE, 2008.

[46] C. Beckhoff, D. Koch et T. Jim, "Goahead : a partial reconfiguration framework", in *20th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Toronto, Canada : IEEE, 2012, p. 37–44.

[47] D. E. Taylor, J. S. Turner et J. W. Lockwood, "Dynamic hardware plugins (dhp) : exploiting reconfigurable hardware for high-performance programmable routers", *Computer Networks*, p. 295–310, 2002.

[48] E. Horta et J. W. Lockwood, "Parbit : a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (fpgas)", *Dept. Comput. Sci., Washington Univ., Saint Louis, MO, Tech. Rep. WUCS-01-13*, 2001.

[49] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown et J. H. Anderson, "Legup : an open-source high-level synthesis tool for fpga-based processor/accelerator systems", *ACM Transactions on Embedded Computing Systems (TECS)*, t. 13, n° 2, p. 24, 2013.

[50] V. Adhangale et R. Daruwala, "Design and implementation of soft core processor on fpga based on avalon bus and sopc technology", *International Journal of Computer Applications*, t. 63, n° 16, 2013.

[51] C Synphony, *Compiler reference manual*, Synopsys Inc, 2012.

[52] P. Coussy et A. Morawiec, *High-Level Synthesis : from Algorithm to Digital Circuit*. Springer Science & Business Media, 2008.

[53] M. Santarini, "Xilinx unveils vivado design suite for the next decade of'all programmable'devices", *Xcell Journal-Solutions for a Programmable World*, n° 79, 2012.

[54] A. Syed Zahid, F. Sébastien et G. Bertrand, "Vivado's esl capabilities speed ip design on zynq soc : automated methodology delivers results similar to hand-coded rtl for two image-processing ip cores", in *Xilinx Xcell Journal, issue 84*, 2013, p. 34–41.

[55] C. Economakos, H. Sidiropoulos et G. Economakos, "Rapid prototyping of digital controllers using fpgas and esl/hls design methodologies", in *Automation and Computing (ICAC), 2013 19th International Conference on*, 2013, p. 1–6.

[56] G. Martin et G. Smith, "High-level synthesis : past, present, and future", *IEEE Design & Test of Computers*, t. 26, n° 4, p. 18–25, 2009.

[57] A. Cornu, S. Derrien et D. Lavenier, "Hls tools for fpga : faster development with better performance", in *Reconfigurable Computing : Architectures, Tools and Applications*, Springer, 2011, p. 67–78.

[58] L. Gantel, A. Khiar, B. Miramond, A Benkhelifa, F. Lemonnier et L. Kessal, "Dataflow programming model for reconfigurable computing", in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, IEEE, 2011, p. 1–8.

[59] F Muller, C Foucher et al., "Implémentation d'un système d'exploitation matériel compatible rtems", in *Colloque GDR SoC/SiP*, 2009.

[60] D. Cederman, D. Hellström, J. Sherrill, G. Bloom, M. Patte et M. Zulianello, "Rtems smp for leon3/leon4 multi-processor devices", *Data Systems In Aerospace*, 2014.

[61] L. Gantel, A. Khiar, B. Miramond, M. E. A. Benkhelifa, L. Kessal, F. Lemonnier et J. Le Rhun, "Enhancing reconfigurable platforms programmability for synchronous data-flow applications", *ACM Trans. Reconfigurable Technol. Syst.*, t. 5, n° 3, 14 :1–14 :16, oct. 2012, ISSN : 1936-7406. DOI : 10.1145/2362374.2362378. adresse : http://doi.acm.org/10.1145/2362374.2362378.

[62] M. Saldana, D. Nunes, E. Ramalho et P. Chow, "Configuration and programming of heterogeneous multiprocessors on a multi-fpga system using tmd-mpi", in *Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference on*, 2006, p. 1–10. DOI : 10.1109/RECONF.2006.307779.

[63] P. Mahr, C. Lorchner, H. Ishebabi et C. Bobda, "Soc-mpi : a flexible message passing library for multiprocessor systems-on-chips", in *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on*, 2008, p. 187–192. DOI : 10.1109/ReConFig.2008.27.

[64] *BORPH : an operating system for FPGA-Based reconfigurable computers...* 2007. adresse : http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-92.pdf.

[65] R. Brodersen, A. Tkachenko et H. K.-H. So, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH", in *Hardware/software codesign and system synthesis, International conference on*, t. 0, Los Alamitos, CA, USA : IEEE Computer Society, 2006, p. 259–264, ISBN : 1-59593-370-0. DOI : 10.1145/1176254.1176316.

[66] C. Foucher, "Design methodology for virtualization and deployment of parallel applications on reconfigurable hardware platform", Theses, Université Nice Sophia Antipolis, oct. 2012. adresse : https://tel.archives-ouvertes.fr/tel-00777511.

[67] C. Foucher, F. Muller et A. Giulieri, "Exploring fpgas capability to host a hpc design", in *NORCHIP, 2010*, IEEE, 2010, p. 1–4.

[68] ——, "Fast integration of hardware accelerators for dynamically reconfigurable architecture", in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, IEEE, 2012, p. 1–7.

[69] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki et K. Kuusilinna, "Uml-based multiprocessor soc design framework", *ACM Trans. Embed. Comput. Syst.*, t. 5, n° 2, p. 281–320, mai 2006, ISSN : 1539-9087. DOI : 10.1145/1151074.1151077. adresse : http://doi.acm.org/10.1145/1151074.1151077.

[70] T. Kangas, *Methods and implementations for automated system on chip architecture exploration*. Tampere University of Technology, 2006.

[71] É. Piel, R. B. Atitallah, P. Marquet, S. Meftali, S. Niar, A. Etien, J.-L. Dekeyser, P. Boulet et I. L.-N. Europe, "Gaspard2 : from marte to systemc simulation", in *Proc. of the DATE*, t. 8, 2008.

[72] F. THOMAS, H. ESPINOZA, S. TAHA et S. GERARD, "Marte : le futur standard omg pour le développement dirigé par les modèles des systèmes embarqués temps réel", *Génie logiciel*, n° 80, p. 27–31, 2007.

[73] S. Guillet, F. de Lamotte, N. Le Griguer, E. Rutten, G. Gogniat et J.-P. Diguet, "Designing formal reconfiguration control using uml/marte", in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, IEEE, 2012, p. 1–8.

[74] P. Wettebled et J.-P. Diguet, "Méthodologie basée sur des membranes pour la gestion de la reconfiguration dynamique dans les systèmes embarquées parallèles", *COMPAS*, 2013.

[75] A. Koudri, J. Champeau, J.-C. Le Lann et V. Leilde, "Mopcom methodology : focus on models of computation", in *Modelling Foundations and Applications*, Springer, 2010, p. 189–200.

[76] C. Paredis, Y. Bernard, R. M. Burkhart, H.-P. de Koning, S. Friedenthal, P. Fritzson, N. F. Rouquette et W. Schamai, "An overview of the sysml-modelica transformation specification", in *INCOSE International Symposium*, 2010.

[77] P. Zaykov, "Mimd implementation with picoblaze microprocessor using mpi functions", in *Proceedings of the 2007 International Conference on Computer Systems and Technologies*, sér. CompSysTech '07, Bulgaria : ACM, 2007, 4 :1–4 :7, isbn : 978-954-9641-50-9. doi : 10.1145/1330598.1330604. adresse : http://doi.acm.org/10.1145/1330598.1330604.

[78] M. Snir, *MPI–the Complete Reference : The MPI core*. MIT press, 1998, t. 1.

[79] A. Lumsdaine, J. M. Squyres et B. Barrett, "Reliability in lam/mpi requirements specification", Technical Report TR563, Department of Computer Science, Indiana University, rapp. tech., 2002.

[80] H. Ong et P. A. Farrell, "Performance comparison of lam/mpi, mpich, and mvich on a linux cluster connected by a gigabit ethernet network", in *Proceedings of the 4th Annual Linux Showcase & Conference*, 2000, p. 10–14.

[81] R. Barriuso et A. Knies, "Shmem user's guide for c", Technical report, Cray Research Inc, rapp. tech., 1994.

[82] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola et C. Bender, "Performance and experience with lapi-a new high-performance communication library for the ibm rs/6000 sp", in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, IEEE, 1998, p. 260–266.

[83] J.-P. Sansonnet, *Architectures des machines parallèles - les réseaux d'interconnexion*, 2014. adresse : `http://perso.limsi.fr/jps/enseignement/tutoriels/archi/ARCHI.03.Reseaux/ch3reseaux.html`.

[84] C. Germain-Renaud et J.-P. Sansonnet, *Les ordinateurs massivement parallèles*, sér. 2AI. Paris : Armand Colin, cop., 1991, isbn : 2-200-42034-X. adresse : `http://opac.inria.fr/record=b1076720`.

[85] T. AMBA, "Specification (ahb)(rev 2.0)", *ARM Ltd*, t. 1, p. 1, 1999.

[86] A. ARM, "Axi protocol specification (rev 2.0)", *Availableat http ://www. arm. com*, 2010.

[87] M. B. Stensgaard et J. Sparso, "Renoc : a network-on-chip architecture with reconfigurable topology", in *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, IEEE, 2008, p. 55–64.

[88] J. Hurt, A. May, X. Zhu et B. Lin, "Design and implementation of high-speed symmetric crossbar schedulers", in *Communications, 1999. ICC'99. 1999 IEEE International Conference on*, IEEE, t. 3, 1999, p. 1478–1483.

[89] S. Xing et W. W. Yu, "Fpga adders : performance evaluation and optimal design", *IEEE Design & Test of Computers*, t. 15, n° 1, p. 24–29, 1998.

[90] R. Thakur, W. Gropp et B. Toonen, "Optimizing the synchronization operations in message passing interface one-sided communication", *International Journal of High Performance Computing Applications*, t. 19, n° 2, p. 119–128, 2005.

[91] W. H. Minhass, J. Öberg et I. Sander, "Design and implementation of a plesiochronous multi-core 4x4 network-on-chip fpga platform with mpi hal support", in *Proceedings of the 6th FPGAworld Conference*, ACM, 2009, p. 52–57.

[92] G. N. Ewo, R. Christian, E. Kiegaing, M. Mbouenda, H. B. Fotsin et B. Granado, "Hardware mpi-2 functions for multi-processing reconfigurable system on chip", in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, IEEE, 2013, p. 273–280.

[93] É. Goubault et C. à l'énergie atomique (France)., *Informatique parallèle et distribution*. École polytechnique, Département d'informatique, 2002.

[94] W. Gropp et E. Lusk, "Dynamic process management in an mpi setting", in *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on*, IEEE, 1995, p. 530–533.

[95] N. P. Richard, *Partial reconfiguration : a simple tutorial*, 2012. adresse : `http://research.microsoft.com/pubs/159545/prtutorial_1.pdf`.

[96] X. Inc, *7 series fpgas configuration ug470 (v1.7)*, 2013. adresse : `http://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf`.