IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

UNIVERSITE BRETAGNE LOIRE

THÈSE / IMT Atlantique

sous le sceau de l'Université Bretagne Loire

pour obtenir le grade de

DOCTEUR D'IMT Atlantique

Spécialité : Electronique

École Doctorale Mathématiques et STIC

Présentée par

André Lalevée

Préparée dans le département Electronique

Laboratoire Labsticc

Towards highly flexible hardware architectures for high-speed data processing: a 100 Gbps network case study

Thèse soutenue le 28 novembre 2017

devant le jury composé de :

Christophe Jego
Professeur, INP/ENSEIRB-MATMECA – Talence / président

Daniel Chillet
Professeur, ENSSAT - Lannion / rapporteur

Virginie Fresse
Maître de conférences (HDR), Université de Saint Etienne / rapporteur

Olivier Muller
Maître de conférences, Ensimag/Institut Polytechnique de Grenoble / examinateur

Michael Huebner
Professeur, Ruhr-Universität Bochem / examinateur

Pierre-Henri Horrein
Maître de conférences, IMT Atlantique / examinateur

Matthieu Arzel
Maître de conférences, IMT Atlantique / examinateur

Michel Jezequel
Professeur, IMT Atlantique / directeur de thèse

Remerciements

Je tiens tout d'abord à remercier Michel Jezequel, pour son aide et ses conseils précieux à la fois en tant que directeur de thèse et chef de département. Je remercie également Matthieu Arzel et Pierre-Henri Horrein pour m'avoir offert l'opportunité d'effectuer cette thèse, m'avoir encadré, soutenu et fait profiter de leur expertise tout au long de ces trois années, et sans qui ces travaux n'auraient pû aboutir. Travailler avec vous était une expérience formidable qui je l'espère pourra se reproduire. J'aimerais remercier chaleureusement Michaël Hübner pour avoir accepter de m'acceuillir en séjour dans son équipe pendant quelques mois lors desquels il m'a fait bénéficier de son incroyable expérience dans le domaine. Je remercie aussi le reste des membres de mon jury de thèse: Christophe Jego, Daniel Chillet, Virginie Fresse et Olivier Muller, avec qui j'ai apprécié échanger lors de ma soutenance.

Lors de cette thèse, j'ai également eu l'agréable opportunité d'effectuer plusieurs missions d'enseignement, et je remercie à cet égard Sylvie Kerouédan pour avoir encadré ces missions.

Je remercie également tous mes collègues du département électronique de Télécom Bretagne, pour leur amitié et leur soutien. Merci tout particulièrement à Paul, Erwan, Franck, Valentin, Benoît, Paul 2, Pierre et Bastien. Merci aussi à Charbel pour ses blagues.

Je remercie enfin ma famille et mes proches pour m'avoir toujours soutenu.

Contents

A Abstract **11**

B Résumé **15**

1 Introduction **22**
1.1 Context 22
1.2 Objectives 24
1.3 Thesis structure 27

2 Background and state of the art on flexible designs on FPGAs **29**
2.1 Network monitoring 29
2.2 Reconfigurable designs on FPGAs 31
2.2.1 Structure of FPGAs 31
2.2.2 Typical design flow for FPGAs 32
2.2.3 Dynamic Partial Reconfiguration 34
2.3 Bitstream relocation on Xilinx FPGAs 36
2.4 Network-On-Chips 39
2.4.1 Topologies 42
2.4.2 Routing algorithms 45
2.5 Conclusion 47

3 Automated design flow for bitstream relocation on Xilinx FPGAs **48**
3.1 Motivation 48
3.2 General overview of the proposed design flow 49
3.2.1 Inputs 49
3.2.2 Automated design flow 51
3.3 New algorithms and techniques for previously missing steps . 57

3.3.1 Floorplanning algorithm dedicated to bitstream relocation . . . 58
3.3.2 New timing constraining technique . . . 66
3.4 Tests and implementation status . . . 67
3.4.1 Floorplanning results . . . 67
3.4.2 Layout description and supported targets . . . 68
3.5 Conclusion . . . 70

4 Using Network-on-Chips for network monitoring 72
4.1 Choosing the NoC characteristics and dimensions . . . 73
4.1.1 Overview of the project board . . . 73
4.1.2 NoC characteristics . . . 74
4.1.3 Generation tool . . . 75
4.1.4 Dimensionning the NoC . . . 76
4.2 Protocol overlay . . . 77
4.2.1 Motivation . . . 77
4.2.2 Providing the sequence of treatments to packets . . . 78
4.2.3 Parameterization information . . . 80
4.2.4 Multipackets . . . 83
4.3 NoC/Functional units interface . . . 87
4.3.1 Interface overview . . . 87
4.3.2 Buffers . . . 87
4.3.3 Acquisition module . . . 90
4.3.4 Unit management module . . . 93
4.3.5 Release module . . . 94
4.4 Conclusion . . . 94

5 Test case: traffic generator 97
5.1 Model presentation . . . 99
5.2 Design presentation . . . 102
5.3 Implementation . . . 104
5.3.1 Overview . . . 104
5.3.2 Senders . . . 104
5.3.3 Outputs . . . 106
5.4 Results . . . 106
5.4.1 Static version . . . 107
5.4.2 Issues with relocation . . . 108
5.4.3 Validation of relocation on other designs . . . 111
5.5 Conclusion . . . 113

6 Conclusion **114**
6.1 Automated design flow for bitstream relocation 114
6.2 Network-on-Chips architectures for network monitoring . . . 115
6.3 Traffic generator test case 116
6.4 Perspectives and future work 117

Glossary **119**

Bibliography **121**

List of Figures

2.1 Simplified intern FPGA two-layer structure 32
2.2 Use case example for module based reconfiguration 35
2.3 Bitstream relocation principle 37
2.4 Potential dysfunctions caused by static resources inside relocatable regions 39
2.5 Example of a point-to-point interconnection scheme 40
2.6 Example of a bus interconnection mechanism 41
2.7 Overview of interconnections between routers and functional units in Network-on-Chips 42
2.8 Example of the router interconnection of a 2D Mesh 3×3 Network-on-Chip 43
2.9 Examples of torus (a) and 3D mesh (b) NoCs 44
2.10 Example of the router interconnection of a tree based Network-on-Chip 44
2.11 Example of an irregular structure mixing mesh and tree based NoC 45

3.1 General view of the proposed design flow 52
3.2 Routing of the static part (a) without using the PRIVATE constraint vs (b) using the PRIVATE constraint 56
3.3 Short connections between the added LUTs and partition pins 57
3.4 Clock regions and frames on FPGAs 59
3.5 Region actually reconfigured when using only parts of frames 59
3.6 Example of iterative identification of fitting patterns: needs 8 slice columns, 1 BRAM column and 1 DSP column 62
3.7 Example of distances computation for one region (pattern is slice-slice-BRAM-slice-slice $\times$ one clock region) 64
3.8 Example of timing differences for two relocatable regions . . . 66
3.9 Example of the result of our floorplanning algorithm (8 regions, 1000 slices, 8 BRAMs, 6 DSPs) on a Virtex7 690t . . . 69

4.1 Types of routers in a 2D-mesh Network-on-Chip 77
4.2 Organization of a packet in the Hermes NoC (Nd = number of data flits) 79
4.3 Providing the sequence of treatments in packets 81
4.4 Example of the sequence update on a packet going through 2 functional units 82
4.5 Description of the parameterization flit 83
4.6 Providing the parameterization information in packets 84
4.7 Final organization of packets on the NoC 86
4.8 Overview of our router to functional unit interface 88
4.9 Port description of our FIFOs 89
4.10 Connections of the acquisition module 90
4.11 Connections of the unit management module 94
4.12 Connections of the unit management module 95

5.1 Port scan stream generation example 101
5.2 Example of modifying the streams in the traffic generator using a) the original implementation and b) our architecture . 103
5.3 Overview of our traffic generator 105
5.4 Screenshot of the placement of the AXI Ethernet 10G IPs (highlighted in green) 110
5.5 Placement of our relocatable regions in regions not occupied by the AXI Ethernet 10G IPs 112

List of Tables

3.1 Maximum number of placeable regions based on needed resources on a Virtex7 690t 68

5.1 Resources used by the 5 by 4 traffic generator using increments only 107
5.2 Resources requirements and maximum achievable frequency of the tested reconfigurable modules 111

Abstract

Networking monitoring is a growing trend in the field of network management and computing. It consists in analyzing and performing tasks on the incoming traffic on a given link. This can have several purposes: obtaining statistics about the network in order to improve its *Quality of Service* (QoS), prioritizing types of packets that require low latency at the expense of others that do not, or detecting and mitigating malicious traffic. As a result, processing architectures targeting network monitoring have to be able to support a wide variety of applications, and thus need to be flexible.

Also the sizes and throughputs of current networks tend to grow drastically over the years, with no signs of slowing down. As of today, it is quite common for a single link to support throughputs up to a few hundreds of gigabits per seconds (*Gbps*).

This means that architectures deployed for network monitoring need to support high throughputs as well. As a result, using purely software architectures is not suited for these applications because, while they can offer an unmatched level of flexibility, their maximum achievable rates are not enough. On the other hand, hardware architectures can manage high througputs, as they are optimized for a specific applications. However this optimization often comes at the cost of a really poor flexibility.

This led us to investigate the use of programmable hardware, such as *Field Programmable Gate Arrays* (FPGAs), as these architectures are usually seen as a good compromise between performance and flexibility, thanks to a trending technique called *Dynamic Partial Reconfiguration* (DPR). This technique allows to modify the behavior of a given region of the device during run-time, which in our case can be useful as it means we can adapt the data processing based on the nature of the incoming traffic. However, when used extensively, this technique presents several drawbacks, such as a big memory footprint for storing all the configuration files. This major drawback can be addressed using *bitstream relocation*. This technique allows to use a configuration file (*bitstream*) to configure a functional unit in another

region than the one for which it was implemented, provided that the origin and destination region share the same layout. It is thus possible to have a single partial bitstream for each functional unit that can be used in every predefined region instead of having one bitstream per region, hence drastically decreasing the memory space required to store all the configuration files.

However, designing for FPGAs, and particularly using DPR is quite tedious, and is often out of the skill set of network engineers, which is even more true when using bitstream relocation. Our goal here is to provide a framework that is easy to use by people not familiar with hardware development. This is based on combining the reconfigurable properties of FPGAs with *Network-on-Chips* (NoCs, which are interconnect structures that offer both high throughput and high flexibility at the cost of a heavy resource usage.

In order to ease the use of bitstream relocation, a fully automated design flow has been developed. This design flow requires inputs such as the number of reconfigurable regions to be placed on the fabric, the dynamic functional units to consider, as well as usual FPGA design inputs, and fully automates all the steps from synthesis to bitstreams generation. It is based on techniques already available in the literature, the ISE Design Suite from Xilinx, and new techniques and algorithms that were previously missing features for a fully automated flow to be possible. Particularly, no previous work concerning bitstream relocation addressed the automation of the floorplanning step, which consists in selecting which parts of the fabric will be used as reconfigurable regions. Our proposed approach for this step takes advantage of the fact that using relocation requires homogeneous regions, which drastically limits the exploration space. Thus our algorithm is divided into two steps. The first one is the identification of a *pattern*, *i.e.* the shape, size and resource arrangement which will be used for all of our reconfigurable regions, based on criteria such as size and resource waste. The second uses a heuristic to select which occurrences of the previously identified pattern will be used as reconfigurable. The criteria for the heuristic is that regions should be placed the closest possible to each other, in order to diminish the length of potential communication wires between regions, while still respecting a threshold under which the static part (*i.e.* the part of the design that will not be reconfigured during run-time) would encounter congestion problems. Finally, we also propose a timing constraining technique in order to ensure that timing constraints will be respected among all regions. This flow has been tested and validated on simple reconfigurable modules, and adds negligible computing time compared to a traditional FPGA design flow.

The proposed design flow gives us flexibility regarding where we can place our reconfigurable units on the design. However, these units still have to be able to exchange data efficiently. For this, we need a flexible interconnect structure that can handle high throughputs. However, the most common interconnect paradigms, point-to-point and bus, are both not suited for this, as the former lacks flexibility and the latter is often seen as a bottleneck, limiting the throughput. As a result, we decided to investigate the use of *Network-on-Chips* for our framework. While NoCs have been a very active field of research in the last decade, it has still not been used in conjunction with bitstream relocation yet. As NoCs can come in many forms, we decided to identify and characterize suitable NoCs for relocation. This led us to use Hermes NoCs, as it supports both the topology (2D full mesh) and switching type (packet switching) that are the most suited to both relocation and network monitoring. However, while the functional units are traditionally responsible for providing the destination to which it has to send a data packet, this is tricky when using bitstream relocation, as the destination unit can be placed anywhere on the NoC. Also, bitstream relocation benefits from having regions as small as possible (as it increases the number of candidate regions). For these reasons, we decided to remove the destination specification mechanism from the functional units and developed a dedicated router/unit interface. This interface allows to use a new scheme where the whole sequence of functional units that a packet has to go through is located inside the packet itself. This also allows us to include parameterization information inside the packets as well as stacking together packets that must go through the same sequence of treatments.

Finally, we tested our whole framework on a test case. This test case is based on an existing traffic generator that uses a skeleton/modifier paradigm. A skeleton (*i.e.* a base IP packet) goes through a set of modifiers (*i.e.* treatment units) in order to generate synthetic traffic that can be used to test new equipment or provide statistics about a network reliability. While the original version of this generator needed a whole reimplementation when the modifiers had to be changed (which can take up to a few hours), our approach would only require partial reconfigurations (which typically require a few milliseconds). Using our approach would allow network engineers to save some valuable time when unexpected test scenarios would have to be generated on the fly. However, while both the NoC and relocation parts have been validated separately, some unexpected issues remain unresolved when using both bitstream relocation and the optical links management IP provided by the board's vendor.

While the scope of this specific work is network monitoring, it is worth

noting that the framework proposed in this thesis can be useful for any application that require both high throughput and high flexibility.

Résumé

Introduction

La surveillance de trafic est devenue une part incontournable de la gestion de réseaux. En effet, l'augmentation perpétuelle de la taille des réseaux et de la diversité des applications transitant dessus nécessitent un suivi grandissant. Cette surveillance peut avoir plusieurs buts: obtenir des statistiques concernant le réseau afin d'en améliorer la qualité de service, pouvoir prioritariser certains types de paquets au détriment d'autres pouvant tolérer une plus forte latence, ou également détecter et mitiger de potentielles attaques, de plus en plus fréquentes sur les réseaux modernes. Les architectures destinées à la surveillance de trafic nécessitent donc un haut degré de flexibilité étant donné la diversité des applications concernées.

Cependant, les réseaux actuels nécessitent de supporter des débits de plus en plus élevés (allant actuellement jusqu'à plusieurs centaines de gigabits par secondes par lien), et les estimations et analyses montrent que cette tendance ne semble pas ralentir. Cette double contrainte flexibilité/débit fait que la plupart des architectures usuelles ne sont pas adaptées à ce genre d'applications. En effet, les architectures purement logicielles, bien qu'offrant un niveau de flexibilité pour l'instant inégalé, ne permettent pas d'atteindre des débits suffisants, alors que les solutions purement matérielles garantissent quant à elles des débits optimisés en échange d'une flexibilité quasi non existante.

Ces limitations des solutions traditionnelles nous ont amenés à nous tourner vers des architectures de type électronique programmable, en particulier les *Field Programmable Gate Arrays* (FPGAs). En effet, ces architectures sont généralement perçues comme un compromis entre débit et flexibilité. Cette flexibilité est notamment rendue possible par la technique de reconfiguration dynamique partielle, qui permet de modifier une fraction du circuit pendant que le reste demeure opérationnel. Cette propriété est particulièrement intéressante dans le cas de la surveillance puisqu'il est pos-

sible d'adapter les unités de calcul présentes sur la carte au trafic observé en entrée de la plateforme. En combinant cette propriété au degré important de parallélisme offert par les FPGAs, il devient alors possible d'atteindre des débits satisfaisants tout en offrant un certain degré de flexibilité. Cependant, l'utilisation extensive de cette technique pose problème quant au stockage des fichiers de configuration. Pour palier ce problème, l'utilisation de la relocation de bitstreams est envisagée.

L'inconvénient du développement sur des architectures matérielles configurables est qu'elles requièrent des compétences qui ne font généralement pas partie de la formation d'un ingénieur réseau. Ainsi, les acteurs du domaine des réseaux ont jusqu'à présent délaissé ces solutions.

L'objectif de cette thèse est de fournir une structure "gros grain" basée sur les architectures reconfigurables qui pourrait être utilisée par des néophytes sans qu'ils aient à déployer les mécanismes inhérents à ces architectures. Pour ce faire une structure de communication flexible et permettant des débits élevés sera également nécessaire pour connecter les unités reconfigurables, ce qui nous a amené à envisager l'utilisation de *Network-on-Chips* (NoCs). Un état de l'art concernant la reconfiguration dynamique partielle, la relocation de bitstreams et les structures de communication de type Réseau-sur-Puce sera présenté. Un flot de conception entièrement automatisé dédié à la relocation de bitstreams sera détaillé permettant de mettre simplement en oeuvre des mécanismes reconfigurables. Une étude et caractèrisation des réseaux sur puce sera effectuée en prenant en compte les problématiques liées à la fois à la relocation de bitstreams et à la surveillance de trafic. Enfin, une étude de cas sera éffectuée en utilisant notre structure pour développer un générateur de trafic flexible.

Etat de l'art sur la flexibilité sur FPGA

Un FPGA est un circuit électronique composé de deux couches. La première, dite couche fonctionnelle, est composé de cellules élémentaires configurables, notamment des *Look-Up Tables* (LUTs), des *Block Random Access Memories* (BRAMs) et des *Digital Signal Processor* (DSPs). Ces cellules sont reliés à un réseau, configurable lui aussi. Ainsi, la configuration des cellules et leurs connexions permettent d'obtenir un circuit plus complexes permettant de réaliser quasiment n'importe quelle application numérique. La deuxième couche, dit couche de configuration, est comme sin nom l'indique directement responsable de la configuration des cellules et du réseau d'interconnexion de la couche fonctionnelle. Ainsi, l'état de la couche

de configuration est donc directement responsable de l'application réalisée par la couche fonctionnelle. La couche de configuration étant basée sur une SRAM, il est alors possible de ne modifier qu'une partie de cette couche tout en laissant le reste intact. Cette propriété a amené une technique intéressante: la reconfiguration dynamique partielle. En effet, puisqu'on peut modifier une partie de la couche de configuration en pleine exécution, on peut donc modifier le comportement d'une partie de la couche fonctionnelle pendant que le reste continue d'opérer. Pour ce faire, des fichiers, appelés bitstreams partiels, représentant le contenu de la zone de la couche de configuration à modifier sont utiliser. Ce qui veut dire qu'il faut un bitstream partiel par fonctionnalité et par zone dans laquelle on veut pouvoir implémenter cette fonctionnalité.

En utilisant extensivement cette technique, on peut alors rencontrer des difficultés à stocker tous les bitstreams partiels. Pour palier ce problème, on peut utiliser la relocation de bitstreams. Cette technique permet d'utiliser un bitstream partiel pour configurer une fonctionnalité mais dans une zone différente que celle initialement prévue, à condition que les zones initiales et finales soient homogènes. Ainsi, on ne nécessite plus qu'un bitstream partiel par fonctionnalité quelque soit le nombre de régions dans lesquelles elle doit pouvoir être implémentée. Cependant, bien que cette technique présente des avantages non négligeables, elle est délicate à mettre en oeuvre et requièrent des connaissances particulières en termes de technologies FPGA. Bien que des efforts aient étés faits dans la littérature pour faciliter ce processus, aucun flot de conception ne l'automatise complètement. Son utilisation reste donc pour l'instant marginale et réservée aux experts.

L'utilisation extensive de la reconfiguration dynamique partielle amène également un problème de connectivité entre les différentes unités de calcul reconfigurables. En effet, avoir un grand nombre de possibilités quant au placement de ces unités implique également qu'il y a un grand de connexions potentielles entre les différentes zones reconfigurables du FPGA. Ainsi, il est nécessaire de déployer une structure de communication qui soit à la fois flexible et performante en termes de débit pour connecter les unités de calculs. Nous nous sommes pour cela intéressés aux NoCs. Bien que les NoCs aient étés un sujet de recherche très actif au cours de la dernière décennie, aucune implémentation n'a été réalisée conjointement à de la relocation de bitstreams. Ces réseaux ayant beaucoup de caractèristiques paramétrables, il faudra alors les caractériser de manière adéquate aux problématiques de relocation et de surveillance de réseau.

Ainsi, les trois prochaines parties de cette thèse concerneront:

- un flot de conception complèment automatisé dédié à la relocation de bitstreams;
- la caractérisation et le dimensionnement d'un NoC adapté à la relocation et à la surveillance de trafic;
- le développement d'un cas d'étude permettant de valider notre approche sur un exemple concret.

Flot de conception automatisé pour la relocation de bitstreams sur FPGAs Xilinx

Afin de pouvoir efficacement utiliser la relocation de bitstreams, et ainsi pouvoir éviter des problèmes de stockage des bitstreams partiels dans le cas d'une utilisation extensive de la reconfiguration dynamique partielle, il paraît nécessaire de développer un flot de conception permettant d'automatiser ce processus. En effet, les étapes de conception liées à cette technique sont longues, fastidieuses et propices aux erreurs. Certaines de ces étapes ont déjà été détaillées dans la littérature, bien que non scriptées. Cependant, d'autres étapes restes manquantes dans l'état de l'art, notamment un algorithme de floorplanning dédié à la relocation, et une méthode pour assurer le respect des contraintes temporelles aux interfaces des zones relogeables.

Le flot ainsi développé se base sur des éléments détaillés au préalable dans l'état de l'art, la suite ISE Design Suite de Xilinx, et des nouveaux algorithmes traitant les étapes jusqu'alors manquantes. Ce flot entièrement automatisé demande en entrée le nombre de régions relogeables souhaitée par le concepteur, la bibliothèque des unités reconfigurables à implémenter ainsi que les paramètres de flot de conception classique sur FPGA. Une fois ces entrées fourniés, aucune intervention n'est nécessaire à l'exécution du flot entièrement articulé à l'aide de scripts, qui produit automatiquement tous les bitstreams nécessaires à l'exécution de l'application.

L'algorithme de floorplanning (*i.e.* la sélection des zones reconfigurables sur la cible FPGA) prend avantage du fait que les régions doivent être homogènes (en termes de taille, de forme, et d'agencement des ressources), ce qui limite grandement l'espace des solutions potentielles. Ainsi, cet algorithme se divise en deux étapes. La première consiste à identifier un *pattern* (*i.e.* (taille, forme, agencement des ressources) qui sera commun à toutes les régions relogeables du design. La deuxième étape consiste à sélectionner, grâce à une heuristique, quelles occurrences de ce pattern sur la carte seront

effectivement utilisées en temps que régions reconfigurables. Cette heuristique, plus précisément un recuit simulé, cherche à minimiser la distance entre les régions retenues (afin de diminuer les délais entre deux zones) tout en s'assurant que ces distances reste supérieures à un seuil (afin d'éviter tout problème de congestion pour les ressources devant être placées ou routées entre ces zones).

Le flot complet a été testé et validé sur des exemples simples et produit des circuits fonctionnels, tout en rajoutant des temps de calculs liés aux étapes de relocation négligeables comparés au temps d'implémentation d'un flot de conception traditionnel.

Utilisation des Network-on-Chips pour la surveillance de trafic

Bien que le flot de conception présenté dans la section précédente offre de la flexbilité au niveau du placement des unités de calculs sur le FPGA, il est également nécessaire pour ces unités de pouvoir communiquer entre elles. Il faut pour cela choisir une structure d'interconnexions adaptée. La structure choisie devra évidemment pouvoir gérer des débits élevés tels que le requièrent les applications de surveillance de trafic. Ainsi, les structures de type bus sont à proscrire car elles représentent généralement un goulot en terme de débit. Egalement, l'utilisation massive de reconfiguration dynamique partielle implique que les unités peuvent être placées n'importe où sur la cible. Ainsi, il est nécessaire d'avoir une infrastructure de communication flexible. Pour cette raison, les connexions de type point à point sont également à proscrire.

Nous nous sommes donc intéressés à l'utilisation de structures de type NoC pour interconnecter les unités de calcul. Les NoCs sont en effet des structures de communication flexibles permettant des débits élevés, au prix d'un coût élevé en termes de ressources électroniques. Les NoCs étant des structures pouvant varier selon diverses caractéristiques, il a fallu caractériser et dimensionner un NoC en adéquation avec les problématiques de relocation et de surveillance de trafic, ce qui nous poussé à retenir des NoCs de type Hermes.

Cependant, bien que l'usage classique des NoCs requièrent que ce soit les unités connectées qui décident de la destination des paquets qu'elles envoient, l'utilisation de relocation de bitstreams rend ce mécanisme difficile. En effet, l'unité de destination pouvant alors être connectée n'importe où sur le NoC, il devient compliqué pour l'unité d'émission de connaître les adresses

des autres unités. Ainsi, une interface dédiée routeur/unité de calcul a été développée afin de décharger les unités de ce mécanisme. Cette interface étend le protocole des NoCs Hermes pour intégrer plusieurs fonctionnalités simplifiant l'utilisation de relocation avec les NoCs, notamment la possibilité d'indiquer la suite des unités à traverser par un paquet à l'intérieur même de ce paquet, l'ajout de données de paramètrisation au sein de entêtes des paquets, et la possibilités d'empiler plusieurs paquets ayant des caractéristiques communes.

Ainsi, nous avons mis au point une structure d'interconnexion dédiée à la relocation de bitstreams à la fois flexibles et haut débit.

Cas d'étude: générateur de trafic

Afin de valider notre approche complète, un cas d'étude a été mené. Nous nous sommes intéressés à un générateur de trafic pour plusieurs raisons. La première raison est que ce type d'application ne requière pas de s'adapter aux trafic entrant puique celui-ci est inexistant. Ainsi, il est possible de valider les mécanismes proposés dans un contexte simplifié, sans avoir à tenir compte de problèmes tels que la gestion des temps de reconfiguration, souvent critique dans les circuits reconfigurables. Egalement, le développement d'une telle application nous permettrait également par la suite de la réutiliser pour tester d'autres circuits plus complexes.

Le générateur ainsi réalisé se base sur un modèle pré-existant reposant sur une structure squelette/modificateurs. Un squelette (*i.e.* un paquet IP de base) est émis à une série de modificateurs (*i.e.* des unités de calcul) agissant sur le squelette pour obtenir un trafic réaliste. L'inconvénient de l'implémentation initiale de ce modèle est qu'elle n'utilisait pas de reconfiguration, ainsi, lorsqu'un modificateur devait être changé, une réimplémentation complète était nécessaire, ce qui peut prendre jusqu'à plusieurs heures. En utilisant notre approche, le changement de la série de modificateurs à utiliser ne requière que des reconfiguration partielles, ce qui prend en général quelques millisecondes.

Lors du développement de ce cas d'étude, les parties NoC et relocation de notre approche ont chacune été validées séparément. Cependant des problèmes restent non résolus lorsqu'on utilise notre flot de conception conjointement au module gérant les interfaces optiques fourni par le constructeur du FPGA. Ainsi, le générateur n'a pas pu être entièrement validé, bien chaque partie de notre approche le soit.

Conclusion

Cette thèse propose une approche “gros grain” visant à simplifier l'utilisation de circuits reconfigurables pour des personnes extérieures à ce domaine. Cette approche utilise conjointement la relocation de bitstreams et les Network-on-Chips pour offrir une architectures flexible supportant de hauts débits. Pour mettre en place cette architecture, un flot de conception entièrement automatisé dédié à le relocation de bitstreams a été développé. Egalement, une étude des NoCs en adéquation aux problématiques de relocation et de surveillance de trafic a été menée. Cette approche a été partiellement validée grâce à un cas d'étude sur le développement d'un générateur de trafic.

Bien que cette approche ait été détaillée spécifiquement dans le cadre de la surveillance de trafic, il pourrait être intéressant de l'adapter à des domaines d'application présentant des contraintes similaires, à savoir flexibilité et haut débit.

Chapter 1

Introduction

1.1 Context

The growing nature of current networks comes with the need for measuring or analyzing various characteristics on a given network. This can be done to fulfill different goals. Having detailed information on the traffic going through a network at a given time can be used in order to improve the *Quality of Service* (QoS) of the network. Critical packets can for example be tagged with a high priority while packets that do not have strong latency constraints can be postponed. Another objective of network monitoring is to detect and mitigate attacks, as networks are often key targets for always more complex attacks.

However, up-to-date networks are often required to be able to support very high rates that can sometimes go up to a few hundreds of Gigabytes per second. This makes using purely software solutions for network monitoring rather impractical. Indeed, while software architectures (such as *Global Purpose Processors* (GPPs) or *Network Processing Units* (NPUs)) benefit from unmatched flexibility as well as easier and quicker developments, these come with a major drawback in terms of achievable rates. Their sequential nature means that only one instruction can be run at a time, making it impossible to target networks whose rates are above a few Gigabytes per second. On the other hand, hardware solutions can achieve really high throughputs thanks to the ability to massively parallelize computations. Purely hardware architectures (such as *Application Specific Integrated Circuit* (ASICs)) are often designed in a way that optimizes every inner part for the targeted application. This usually allows to achieve high performance at the cost of 1) long development times, 2) close to no flexibility, as no modification can be done

once the design has been implemented and 3) really high development and production costs. However, networking applications usually require short development times, as the network landscape is in constant evolution. Also, as monitoring applications can depend on the nature of the monitored traffic, architectures that target these applications either have to support every application at once or be able to give enough flexibility to adapt to the traffic. As the number of different applications makes it impractical to implement all of them on a single chip, architectures considered for network monitoring should offer enough flexibility to adapt to the incoming traffic. Thus, hardware-only solutions are not well suited to network monitoring because of both development times and lack of flexibility.

As software and hardware solutions both have strong limitations in regards to network monitoring, it could be interesting to explore the use of firmwares such as *Field Programmable Gate Arrays* (FPGAs) for such applications. Indeed, these types of solutions are often regarded as a good compromise between hardware and software architectures. FPGAs can be seen as matrices of elementary cells called *Look Up Tables* (LUTs), which can all independently be configured to perform any basic logic operation. Other cells are also usually available on an FPGA fabric, such as *Block RAMs* (BRAMs) used to store information and *Digital Signal Processing* (DSP) units that can perform complex arithmetic computations. All these cells are connected to a configurable routing structure, allowing them to be combined in order to perform complex applications. While these architectures can not achieve such a high level of optimization as a purely hardware solution, they can still benefit from massive parallelism properties, allowing them to achieve much higher rates than software solutions. Also, the fact that FPGAs can not be as much optimized as ASICs also means that development times are typically smaller for the former, but it still can not compete with software solutions in that regard. Their configurable nature also makes it possible to reset the fabric's behavior even after a design has already been implemented, while such a feat is not possible for ASICs. A relatively recent feature called *Dynamic Partial Reconfiguration* (DPR) takes it even further by making it possible to reconfigure only a portion of the fabric while the rest is still operating. This seems particularly interesting in the case of network monitoring for two reasons: 1) it makes it possible to add new features to an existing system without having to reset the whole device, which is well suited for applications that are likely to need updates and 2) it also allows the fabric to modify its behavior depending on the traffic being monitored. So FPGAs, while not benefiting from a level of flexibility as high as software solutions, as well as not being able to achieve

such high levels of performance as purely hardware architectures, seem to be well suited for network monitoring, where data has to be processed at high throughput and in a flexible way

1.2 Objectives

One of the primary objectives of this thesis is to find dedicated architectures for network monitoring using FPGAs. These architectures have to be able to meet the strong throughput constraints required by up-to-date networks, while providing enough flexibility to be able to host different processing units based on the traffic being monitored, as well as to be able to be updated without having to stop the whole system.

In order to achieve this, Dynamic Partial Reconfiguration of FPGAs will be explored as it offers great possibilities for implementing flexible designs on a platform that is likely to support high rate processing. However, when dealing with highly flexible designs, DPR often comes at the cost of increased compilation times and memory usage. Usually, reconfigurable designs are divided into two distinct parts: 1) the static part, which comprises all the parts of the design that will always have the same functionality throughout the course of execution of the application, and 2) dynamic regions, which can host precompiled units and swap between them during the execution using configuration files called partial bitstreams. When a reconfigurable module has to be implemented in several reconfigurable regions, one partial bitstream is needed per region. So, in case of a design where many different modules have to be implemented in a lot of regions, the compilation times and memory space needed to store each partial bitstream can greatly increase. For example, if 20 modules all have to be implemented in 20 regions, 400 (20×20) partial bitstreams will have to be compiled and stored in memory. While memory storage is usually critical in embedded systems, where the available memory space is limited, it can seem to be irrelevant in the case of network monitoring devices that are likely to be placed in servers where memory space is not a problem. However, some critical applications may need a really low latency for a reconfiguration. For example, we can consider an application where a platform is hosting algorithms for detecting and mitigating network attacks. While the detection part of these algorithms would have to be implemented in the static part of a design, the mitigation part would more likely be inactive most of the time, thus allowing the designer to implement that part as a reconfigurable module. So, in this scenario, when an attack is detected, a reconfiguration request would

be triggered to implement the mitigation part, which obviously should be done as quickly as possible. Storing the partial bitstreams in a fast memory placed close to the FPGA fabric is then needed to decrease the time between a reconfiguration request and the availability of the module on chip. However, fast memories are often really expensive, so decreasing the memory footprint of partial bitstreams becomes crucial. Thus, we have investigated bitstream relocation in order to reduce the memory space required to store our partial bitstreams. This technique allows a partial bitstream to be used to configure a dynamic module in a region it was not implemented for. With bitstream relocation, only one partial bitstream is needed per dynamic module whatever the number of regions in which it has to be implemented is. Taking back the previous example with 20 modules and 20 regions, only 20 partial bitstream are now required instead of 400, resulting in shorter compilation times and smaller memory space. However, bitstream relocation comes with very strong constraints which makes it a long and error-prone process. Unfortunately, no effort that we are aware of has been made to automate this process. Another problem with bitstream relocation is that it requires extensive knowledge of the considered FPGA fabric which would not be needed otherwise when working with non-relocatable designs. As a result, bitstream relocation is still a very rarely used technique due to the fact that it is quite difficult to perform. Developing an automated design flow dedicated to bitstream relocation while keeping the steps related to relocation entirely transparent to the user is thus another key objective of this thesis.

While Dynamic Partial Reconfiguration offers flexibility in terms of the available algorithms on the FPGA fabric at a given time, it seems unlikely that every packet on a 100 Gbps stream would have to be applied the same treatments. Instead, network monitoring applications usually start with a classification step whose results can be used to identify which algorithm should be applied to each packet. This means that each packet is likely to only need to go through a subset of all algorithms implemented on the fabric. Hence, point-to-point connections between treatment units is not suited for such applications, as these routing architectures force a unique path for all packets. As a result, we also need a flexible routing structure. However, this is impractical using DPR. Indeed, the latency introduced by a reconfiguration implies that this technique is not suited for applications that require a lot of connection switching. For that reason, using DPR on routing structures is usually considered bad practice. Among existing techniques for flexible routing, the most common are bus interconnections and *Network-on-Chips* (NoCs). While bus interconnects are usually well-documented and

easy to use, they suffer from the fact that only one connection between two elements is allowed at any given time, greatly impeding the massive parallelism offered by FPGA fabrics and, as a result, creating a bottleneck that makes it unlikely to meet the throughput requirements of network monitoring applications. On the other hand, NoCs seem well-suited to fully exploit the massive parallelism of hardware architectures. Indeed, a NoC consists in a group of routers, each interfaced with its neighbors, and that all have a local connection, typically interfaced with a computational unit. This structure allows for multiple connections to be active simultaneously. NoCs have been a very active field of research in the past decade, and many topologies and routing algorithms have already been proposed. Finding suitable topologies and routing techniques that allows to meet both flexibility and throughput criteria will be a challenge to investigate in this thesis.

Also, while in NoCs the computational units that are interfaced with the routers are usually responsible for determining which router they should send their packets to, relocatable designs usually require the dynamic regions to be as small as possible in order to limit the area waste introduced by relocation. Moreover, relocation constraints usually makes it so that the regions need to be small for a relocation to be performed in a high number of regions. To simplify, the smaller the regions are, the higher the number of possible relocations is. As smaller regions are tied with less complex modules, this means that reconfigurable modules would really benefit from being as less complex as possible. As a result, it would be unoptimal to include a mechanism responsible for determining the next router in the reconfigurable units. As this information has to be given to the routers interfaced with the computational units, this means that we need to insert mechanisms to handle this between the routers and the computational units. Finding and implementing such mechanisms is essential to ensure enough flexibility in our architectures.

Finally, network monitoring architectures have to be tested before being deployed. In this regard, a realistic and configurable traffic generator is needed in order to ensure the validity of our approach. Fortunately, some traffic generation models have already been developed, that are based on elementary modifiers that are applied to a common packet. However, current implementation of such models are highly static, in the sense that once a behavior has been set, changing it needs a complete recompilation of the design, which can take up to several hours. Our approach is particularly well suited for this kind of model, as we can compile every elementary modifier as a reconfigurable module. Changing the behavior of the generator would then only require a few reconfigurations (which typically take a few millisec-

onds) instead of a complete recompilation. Implementing these generation models using our architectures would then not only validate our approach, but also be used to test more complex future applications that would use these architectures.

1.3 Thesis structure

This document is structured as follows.

Chapter 2 will provide a background on network monitoring applications and architectures to identify the constraints we will have to meet in our designs. It also presents a state-of-the-art on reconfigurable designs and Dynamic Partial Reconfiguration on FPGAs, as well as an introduction to bitstream relocation and the constraints that have to be respected for this technique to work. An overview of Network-on-Chips and their various characteristics on which we will base our design choices will finally be presented.

Chapter 3 will present a new fully automated design flow dedicated to bitstream relocation on Xilinx FPGAs. This design flow is based on existing design flows, techniques already described on the literature, and new algorithms specifically developed to address steps that were previously missing in order for a relocation automated design flow to work. Specifically, the lack of a floorplanning algorithm dedicated to relocation as well as a timing constraining technique for relocatable modules have been addressed. This tool aims at being user-friendly, as no prior knowledge of the design target is needed by the designer, and no intervention is required once all sources and parameters have been provided. This tool provides successful implementations in only a few additional minutes over a whole traditional FPGA design flow.

In chapter 4, the use of Network-on-Chip architectures for network monitoring applications will be investigated. The choice of an Hermes NoC and its characteristics will be justified according to constraints induced by network monitoring requirements and the FPGA available for this project. This chapter will also present an extension to the protocol used in the Hermes NoC in order to ease and optimize the use of bitstream relocation on this architecture. A generic interface between routers and relocatable modules will also be presented, which makes all the new mechanisms induced by our protocol extension transparent to both the NoC and the relocatable modules. This interface also moves heavy mechanisms such as buffering away from the relocatable modules, in order to decrease the complexity of the

reconfigurable parts of our designs.

Chapter 5 will present a case study for our architecture that combines both bitstream relocation and Network-on-Chip. This case study is a traffic generator based on an existing generation model that was previously lacking in terms of flexibility. While the original implementation of that generator required a full reimplementation as soon as the model needed to be changed, using our architecture could potentially only require a few reconfigurations, which can drastically reduce the time required to generate a new traffic model. However, issues are still unresolved when using both bitstream relocation and the *AXI 10G Ethernet Interface* [1] *Intellectual Property* (IP) from Xilinx, which is required for our application. In that regard, we thus provided simpler test cases to validate our NoC architecture and our bitstream relocation design flow separately.

Finally, chapter 6 will summarize our contributions and give conclusions based on the results obtained with our architecture. It will also present the perspectives about the future work and improvement that can be considered based on this thesis.

Chapter 2

Background and state of the art on flexible designs on FPGAs

2.1 Network monitoring

Network monitoring is becoming a necessary task for network managers, as the more and more extensive use of IP networks brings the need to both gather and interpret information about the state of a network as well as to detect and possibly mitigate some potentially malicious packets on a that network.

Network monitoring applications can cover various purposes. Network engineering can be done through the gathering of information that can be used in order to improve the *Quality of Service* (QoS) of a given network based on the incoming traffic. A big part of those applications usually involves a traffic classification step. Traffic classification consists in sorting the incoming traffic following predefined criteria. One goal of classification can be to get information on which applications the incoming traffic is made of in order to get statistics about a network's usage. Another goal can be to filter the traffic in order to apply treatments to some packets if they correspond to a specific set of characteristics. Also, using traffic classification, it is possible to prioritize some critical parts of the traffic in order to reduce their latency, which usually comes at the cost of delaying other parts of the traffic. For example, some applications types such as streaming can require to have very low latency, while it is acceptable to delay the sending of a webpage, in which case delaying the latter to send the former earlier can

improve the global QoS of a network.

Another important part of network monitoring is to improve the security of a network. More and more attacks are being observed over the internet. For example, the number of *Distribute Denial Of Service* (DDoS) attacks has seen a 140% growth between 2015 and 2016 [2]. While most recent trends indicate that this number has started decreasing in early 2017, their average volume continues to increase [3]. Providing solutions to detect and mitigate these attacks is thus a task that is becoming more and more important for network providers.

This leads to two major requirements for architectures supporting network monitoring applications. First, the architecture must be able to support high throughputs, as traffic volumes on IP networks are constantly increasing. Second, the architecture has to be flexible enough to support a wide variety of applications. Indeed, the diversity that can be observed on a single link makes that architectures would greatly benefit from being able to adapt the the currently incoming traffic.

However, existing solutions for network monitoring usually focus on only one of these aspects. Indeed, most available flexible solutions are usually based on software systems, which offer really poor throughput performances. For example, [63] provides an highly flexible traffic measurement platform, however it can only achieve about 12Gbps using 8 Intel Xeon CPUs. [28] presents a framework for real-time monitoring capable to adapt to any type of incoming traffic, but is only able to reach 762Mbps for minimal size packets (64 bytes) on a single core Intel Centrino CPU. On the other hand, solutions based on *Field Programmable Gate Arrays* (FPGAs) are usually able to sustain higher data rates. For example, [24] is able to implement a *Network Intrusion Detection System* (NIDS) able to reach 23.76Gbps on a VirtexII-Pro (which should increase with up-to-date technologies). [27] is able to implement a packet classifier that is able to sustain a rate of 100Gbps on a Virtex5 vsx240t. [39] implements an NIDS based on the Snort [50] rule set at 10Gbps on an Altera DE2-70 development board. However, all these implementations are very rigid, as no adaptation of the designs can be made at run-time.

As a result, it can be interesting to investigate the use of flexible architectures on FPGA so that network monitoring applications can benefit from both high throughputs and high flexibility. In this chapter, we will present existing methods for implementing flexible designs on FPGAs. Section 2.2 will present a background and state-of-the-art on *Dynamic Partial Reconfiguration* (DPR) and bitstream relocation (section 2.3, while section 2.4 will present an overview on *Network-on-Chips* (NoCs), a flexible communication

architecture for digital systems.

2.2 Reconfigurable designs on FPGAs

Network monitoring applications are likely to need flexibility in terms of the different treatments that can have to be performed on the incoming traffic. Indeed, supporting all possible treatments at once on a same hardware target is highly unlikely, especially considering the ever-growing diversity of network applications or attacks. Moreover, designing applications that are likely to have to be updated during its lifetime means that architectures that support them can not conceivably be unadaptable. Fortunately, FPGAs offer a mechanism, called Dynamic Partial Reconfiguration, that allows for only a part of a design to be modified during run-time while the rest of the fabric is still operating. This makes FPGAs a prime target for applications that require high levels of both flexibility and throughput, such as network monitoring.

2.2.1 Structure of FPGAs

An FPGA is a configurable electronic structure that is divided into two interconnected layers: the functional layer and the configuration layer.

The functional can be seen as a matrix of configurable elements that are all connected to a routing structure that is configurable as well. These configurable elements can either be *Look Up Tables* (LUTs), *i.e.* logical cells that can perform any logical computation for a fixed number of inputs, that can be paired with flip-flops, *Block Random Access Memories* (BRAMs) or *Digital Signal Processors* (DSPs) that are used for complex arithmetic computations and provide dedicated logic for accelerating additions. Other components can be found depending on the target, which usually consists in interfaces supporting various communication standards. As all these cells, as well as the routing structure that connects them, are configurable, they can be combined to achieve complex digital designs.

The configuration layer, as its name suggests, is responsible for the configuration of the functional layer, *i.e.* the way the cells of the functional layer are configured and interconnected. This layer is based on a *Static Random Access Memory* (SRAM), which is arranged in a way so that a contiguous space of that memory is used to configure a configurable cell of the functional layer (see figure 2.1).

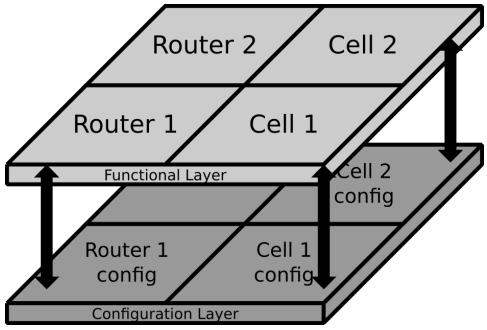


Figure 2.1 – Simplified intern FPGA two-layer structure

2.2.2 Typical design flow for FPGAs

RTL description

The first step to implement a design on an FPGA is for the designer to provide a *Register Transfer Level* (RTL) description of the design. RTL languages aim at describing the behavior of the logic signals of a design between its registers. Common RTL languages are Verilog or VHDL. Both of them allow the designer to organize its design in a hierarchical and behavioral description. Every hierarchical unit is called a component and consists in a list of ports (*input-outputs* (IOs)) paired with a behavioral description, and can contain other components inside them. They can also be parameterized in order to be reused with a different behavior without having to be redeveloped. The highest component in a design's hierarchy is usually referred to as the *top module* of the design.

Synthesis

The goal of the synthesis step is to translate the RTL description into elementary logic operations, *e.g.* identifying adders and multipliers, extracting *Finite State Machines* (FSMs) or memory units such as RAMs. This step is usually fully automated by *Electronic Design Automation* (EDA) tools. In theory, this step is independent of the FPGA target considered for the design, however, modern synthesis tools take the target into account in order to operate pre-treatments or target-dependant optimizations and provide timing and resource usage estimations. The result of this step is called a logical netlist.

Implementation

The goal of the implementation step is to translate the logical netlist(s) previously obtained during the synthesis step into a functioning design on a specific FPGA fabric. This step is usually divided into 3 sub-steps. While there is standard denomination for these sub-steps, there goals differ very little among available design flows. Thus, while the names of the sub-steps we will present here are only specific to the Xilinx ISE design suite (as this is the design flow we will base our work upon), similar tasks can be observed on other EDA tools.

Translate This task merges all the input logical netlists and constraints into a Xilinx design file called *Native Generic Database* (NGD) . The constraint file(s) (in this design flow the *User Constraint File(s)* (UCF)) is a user-defined target dependant file that can contain:

- bindings between the ports of the top module of the design and the physical pins of the FPGA target
- timings information or constraints on the IOs or intern signals of the design
- various constraints allowing the designer to lock some signals to specific resources, allocate some regions for hierarchical instances of the design, *etc...*

Map This task maps the logical elements and signals of the NGD file obtained during the translate step to actual resources available on the considered target. During this process, only the type of resource for each element is given, not their actual location on the fabric.

Place and Route The *Place and Route* (PaR) finally places every element of the design on a specific location of the design, and determines the state of the routing structure in order for each signal to be correctly routed. The result of this process is thus a full description of the functional layer of the FPGA for the design.

Configuration

Usually, when loading a new design on the FPGA fabric, the whole configuration layer is modified by loading a file, called *bitstream*, into its SRAM.

The bitstream contains, besides a small header mainly used to ensure data integrity, all the words that have to be stored in the SRAM for the functional layer to perform the desired design. However, the fact that SRAMs' contents can be independently accessed has led to an interesting opportunity, which is to modify only a part of the configuration without changing the rest of the fabric.

2.2.3 Dynamic Partial Reconfiguration

Dynamic partial reconfiguration of FPGAs is a technique that allows a predefined portion the fabric to be reconfigured during run-time while the rest is still under execution. This process is made possible thanks to the inner structure of FPGAs, as the configuration layer is based on an SRAM. Indeed, it is possible to modify only a portion of the SRAM without affecting the rest, which equals to modifying only a portion of the functional layer without affecting the rest.

In order to modify the behavior of the fabric, files that describe the new configurations have to be loaded in the configuration SRAM. These files are called partial bitstreams, and, similarly to the usual bitstreams, contain the information that has to be loaded in the SRAM, but only relative to the portions that have to be reconfigured, thanks to an addressing mechanism that allows the information to be written in the desired location.

Early reconfigurable designs on FPGAs used an approach called *difference based* partial reconfiguration. With this approach, the designer had to fully implement a first design, then had to manually modify the functional layer (using for example tools like *FPGA Editor* in the case of Xilinx FPGAs) and generate the new resulting bitstream. The original and modified bitstreams would then be compared and a partial bitstream containing only the differences between the two was generated. While this approach is well suited for punctual modifications, such as for example changing the logical equation of a single or a few LUTs, it is inadequate to use this method for more reconfiguration-heavy designs. Moreover, manually modifying the functional layer is a long and tedious task that can not take advantage of common designing methods, such as using RTL languages. Difference-based reconfiguration has thus become obsolete.

The now commonly adopted approach for partial reconfiguration is called *module based* (or sometimes *slot based*) reconfiguration. With this approach, the functional layer is usually divided into two distinct parts: the static part, and the dynamic part. The dynamic part consists in predefined regions that can be subject to reconfiguration while the static part is every the part of

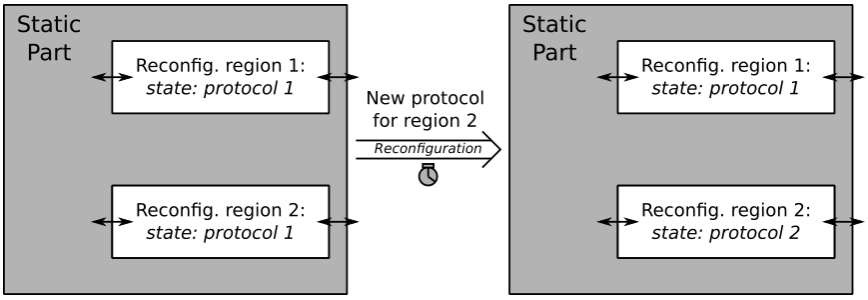


Figure 2.2 – Use case example for module based reconfiguration

the design which behavior will not be modified throughout the execution.

A classic example that illustrates the benefits of module based reconfiguration is the use case of a network device that has several interfaces that can all implement different communication protocols. In this case, each interface will be connected to a reconfigurable region in which each communication protocol can be implemented. If at any time one of the interfaces has to change its communication protocol, not using partial reconfiguration would likely have required one of these two solutions: 1) have every protocol be implemented simultaneously for each interface and have a multiplexer control which protocol to use, resulting in a huge area waste (as each would need its own logical resources to be implemented) or 2) stop the FPGA and reconfigure the whole fabric, which means that all the other interfaces will be stopped as well. However, with module based partial reconfiguration, the communication protocol of every interface can be changed without having to stop the others nor having to have every protocol be implemented simultaneously (see 2.2).

With this method, a designer has to first determine which part(s) of his design will have to be able to be reconfigured. The RTL description of the static contains empty components for which only the ports are defined. A "black box" attribute is given to these components so that EDA tools can treat them as parts of the design that will be filled later on. By synthesizing reconfigurable modules apart from the static part, it is then possible to switch between these modules inside the black boxes. As the layouts of FPGAs are usually non-homogeneous, the designer has to select regions on the fabric according to the number of each type of resources required by the reconfigurable modules. This step is called *floorplanning*. Usually, EDA tools will automatically place interfaces (called *partition pins*) to ensure communication between the static part and these regions based on the

number of IOs required by the reconfigurable modules. The place and route step of the dynamic regions is then done based on the selected regions and their interfaces and on the timing identified on the previously implemented static part.

After the place and route process of each dynamic part of the design, a full bitstream is generated that contains the static part and an initial configuration, as well as a partial bitstream for each dynamic module per reconfigurable region in which it is implemented. These partial bitstream have the same structure as a full bitstream, except that the start address is set to the address of the considered reconfigurable region in the SRAM, as well as a decreased size field.

2.3 Bitstream relocation on Xilinx FPGAs

While partial reconfiguration offers huge advantages in terms of flexibility, using it extensively can come with several inconvenients. Indeed, when a design requires that a lot of reconfigurable modules have to be able to be implemented in a lot of reconfigurable regions, the number of partial bitstreams that have to be generated and stored can drastically increase. For example, if 20 reconfigurable modules all have to be able to be implemented in 20 reconfigurable regions, 400 (20×20) partial bitstreams will have to be generated and stored in a remote memory accessible by the FPGA fabric. While bitstream generation time is not seen as a problem for a final design, it can become problematic when prototyping. Indeed, implementing a module in one region can take up to a few hours. Reiterating this in 20 regions becomes unacceptable just to verify the module functionality in every region, even more so when this has to be done for several modules. Also, while memory space is usually seen as a problem for embedded systems, it is not the case for network applications, where the FPGA target is likely to be hosted in servers where memory is abundant. However, reconfiguration latency can be highly critical for several treatments. For example, in a case where an attack is detected, the functional module that reacts to that attack has to reconfigurable with as little latency as possible, else it is possible that many fraudulous packets would be able not to be treated. Reconfiguration latency can also be an issue due to the high throughput involved: for example, on a 100Gbps link, having to wait 1ms for a module to be configured means that potentially 100Mb of data have to be buffered before the module can start treating it. This means that partial bitstreams have to ideally be stored in fast on-chip memories in order to reduce reconfiguration latency. As fast

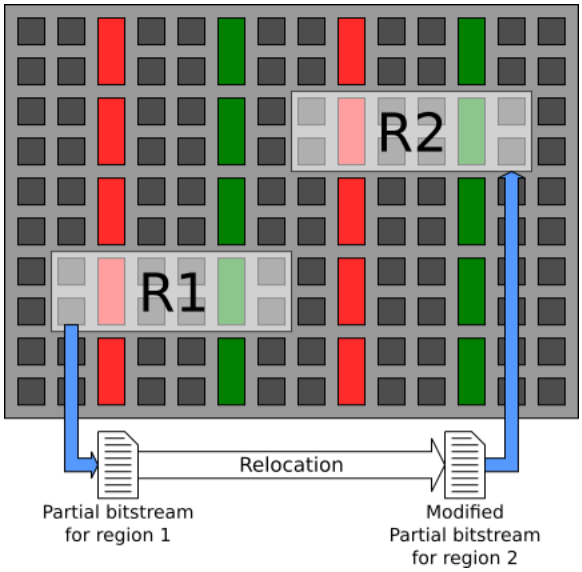


Figure 2.3 – Bitstream relocation principle

memories are usually really expensive, reducing the memory space needed to store partial bitstreams can be highly advantageous. So, decreasing the number of partial bitstreams required in a design is highly encouraged.

Bitstream relocation [62] is a technique that allows to use a partial bitstream that was generated for a specific region to implement the same functionality in a region it was not generated for (see figure 2.3).

Using this technique can limit the number of partial bitstreams required for a reconfigurable module to one whatever the number of regions it has to be implemented in. Looking back at our previous example, only 20 partial bitstreams would then be needed instead of 400, drastically reducing both the generation time and the memory space that are required.

While early works on relocation showed promising results [21] [41] thanks to the homogeneous structure of FPGAs at that time, the trend towards more and more heterogeneous FPGAs makes this technique more complicated to use. Indeed, relocating a partial module on an homogeneous structure only requires the start address to be modified for the module to be successfully implemented anywhere on the fabric. However, this is not the case for heterogeneous fabrics, as the configuration data of the partial bitstream of the origin region will not necessarily match the resources inside the destination region.

This means that regions have to fulfill requirements in order to perform a

relocation. These constraints have been described in [25] and [26]. The first one is that the origin and destination regions must be identical in terms of size and resource arrangements in order for the configuration data of the partial bitstream to match the resources inside the destination region. Though some techniques focus on bitstream relocation between non-identical regions [15], doing so often requires specific information about the fabric and bitstream encoding that is usually not provided by the manufacturers. Identifying compatible regions for bitstream relocation is thus a key step when using this technique, which is made tougher and tougher by the always increasing heterogeneity of FPGA fabrics.

The second requirement is that all the interfaces between the static part and the relocatable regions (*i.e.* partition pins) must be placed at the same relative locations among all regions. This is pretty straightforward since designers can use location constraints that are provided by the manufacturers' EDA tools.

Finally, the static part must use any resources inside the relocatable regions in the same manner regardless of the region, may them be logic or routing resources, which is not the case in traditional reconfigurable design flows. Indeed, current design flows allows the static part to borrow resources inside the relocatable, primarily in order to ease up the routing of the static part. When not relocating, this is not a problem as the partial bitstreams of a region will contain the information about the portion of the static part inside that region. However, when relocating, that information will not match the static part around the new region, potentially causing dysfunctions (see figure 2.4). As it can be really tedious to constrain the routing of the static part, the adopted solution usually consists in preventing the static part from using any resources inside the relocatable regions altogether. By making sure that the static part can not use resources inside reconfigurable regions, we in fact make sure that no information can potentially be written at the wrong place, or overwrite critical other information.

Once all these constraints have been applied to the relocatable regions, each reconfigurable module has to be implemented in only one region. Only one partial bitstream per module will then be generated, which can be used to configure that module in every region, provided a modification of the *Frame Address Register* (FAR) in the partial bitstream header, which corresponds to the starting address at which the bitstream will be written in the fabric SRAM, *i.e.* the region of the functional layer that will be reconfigured. While this modification can be done easily, as the new addresses can be computed offline, modifying the bitstream on the fly can be crucial in the case of applications where reconfiguration time is critical. Hardware

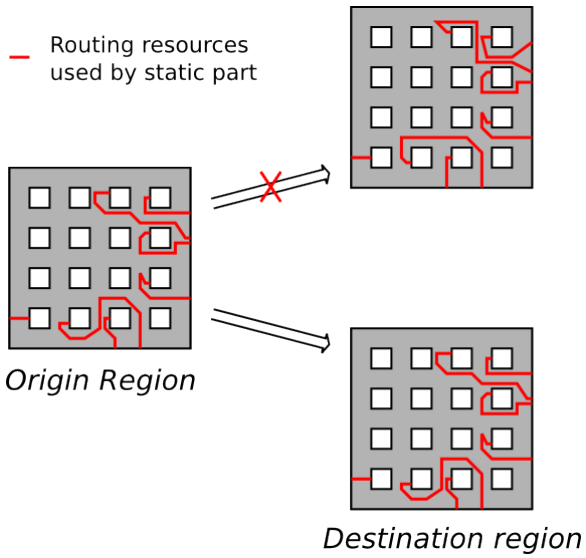


Figure 2.4 – Potential dysfunctions caused by static resources inside relocatable regions

versions of this modification have been proposed in [35] and [36], however, as at that time the reconfigurable regions on FPGAs had to span the entire height of the fabric, these versions only manage horizontal relocations.

However, while relocation can provide interesting advantages in terms of both compilation times and memory space required for partial bitstream, this technique has not been automated yet. As relocation can be long and error-prone to do manually, it can be quite interesting to develop a design flow that would entirely automate this process. The lack of automated tools for bitstream relocation has been a serious impediment to its adoption by designers, especially considering the diversity and complexity of modern FPGAs, as relocation requires a deep knowledge of the considered fabric.

2.4 Network-On-Chips

While flexibility in terms of what applications can be run at any time is required, flexibility in terms of what treatment each packet has to be applied is also needed. Indeed, the traffic diversity on a 100Gbps link can be huge, so it is very unlikely that all the packets will need the same treatment. Instead, almost all network monitoring systems have a classification process at

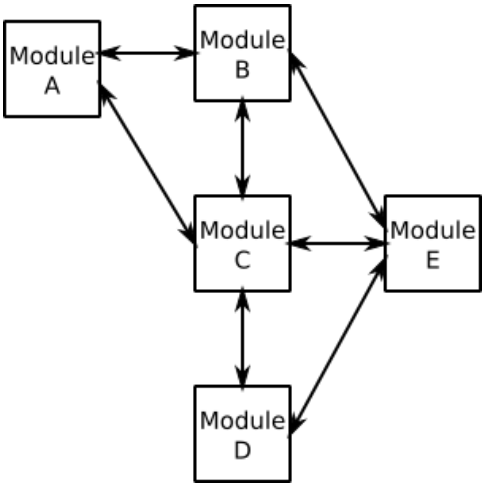


Figure 2.5 – Example of a point-to-point interconnection scheme

the start of the chain that decides what treatment has to be applied to each single packet. For example, classification results might indicate that packet 1 must go to processing module A, packet 2 to module B, and packet 3 to both modules A and B. This means that network monitoring applications require a communication structure that allows for flexible module-to-module interconnections while supporting high data rates and concurrent communications.

Among existing communication structures in digital systems, three main types can be distinguished. The first one is point-to-point (see figure 2.5), which connects each component's interface to another one. While this type of connection is fairly straightforward to implement as well as resource efficient, it severely lacks flexibility, as every component can only communicate to the one it is directly connected.

Another common type of communication architecture is the bus mechanism (see figure 2.6). In this architecture, each component is connected to the same communication medium, called bus, and has a unique address in order to identify itself on the bus. When a component has to transmit data to another, it has to specify to which address (*i.e.* component) the data has to be sent. The receiving component accepts the incoming data as the address field matches its own, and all other will not perform any task as their own address will mismatch the one specified on the bus. This architecture offers great flexibility, as all components can potentially communicate with each other, and is the most commonly found in processor-based systems.

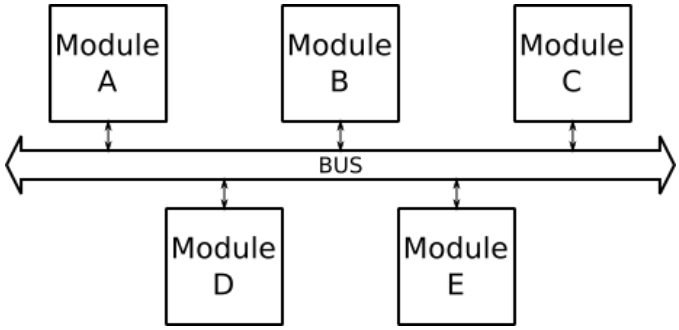


Figure 2.6 – Example of a bus interconnection mechanism

However, this architecture is not well-suited for applications where extensive communications are required. Indeed, this architecture only allows one module-to-module connection at any given time, as the bus is shared by all components. This means that buses are usually bottlenecks for applications where a lot of data has to be exchanged between several modules.

Network-on-Chips [33, 38, 17] provide both flexibility and high rate possibilities. These architectures consist in a set of interconnected routers. Each router is connected to a single functional unit, and vice-versa, and to a subset of the other routers of the NoC (see figure 2.7). Each component connected to the NoC also gets a unique address on the network, which also corresponds to the router to which it is connected. When a component has to transmit data to another one, it releases that data to its router, along with the address of the destination router, *i.e.* the address of the destination component. The data packets are then transmitted from router to router until they reach their destination. The way the routers are interconnected (topology 2.4.1) and the way the packets are transmitted from router to router (routing algorithm 2.4.2) can vary depending on the NoC. NoCs can also vary in the way their routers implement more advanced functionalities, which are beyond the scope of this document.

This architecture can often sustain higher data rates than a bus mechanism, as several paths can be active at the same time, *i.e.* several pairs of component can simultaneously exchange data without losing any flexibility, as all components can still communicate with each other. This means that NoCs offer a much better scalability than its bus counterpart. However, it has several drawbacks, such as an increased resource cost (as the number of wires is duplicated several times compared to a bus), as well as a slightly increased latency for an active connection, as each packet is likely to have

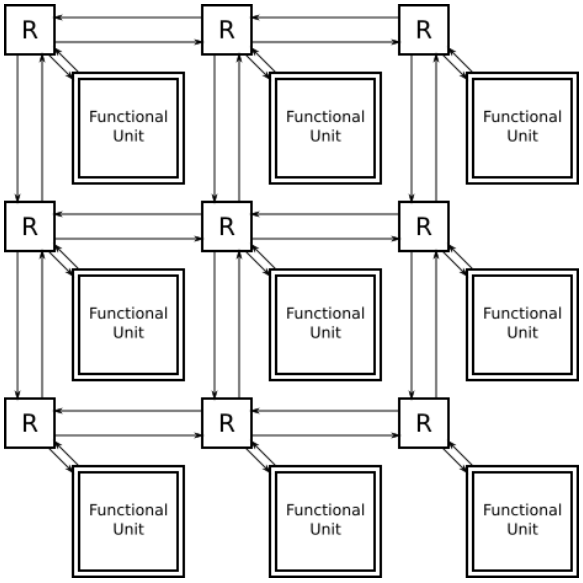


Figure 2.7 – Overview of interconnections between routers and functional units in Network-on-Chips

to go through several routers before achieving its destination.

2.4.1 Topologies

The topology of a Network-on-Chip is the way the routers are interconnected.

One of the most popular and generic topology is the 2D mesh topology (see figure 2.8). In this case, the topology can be seen as a regular discrete orthogonal grid on which each node is a router. Each router is then interconnected with each of his neighbours. Thanks to its regular structure, addresses of routers on such a topology are really straightforward, as they usually correspond to the abscissas and ordinates of routers on the grid. This topology is well suited for applications where high flexibility is needed, as the high number of paths on that topology leads to very unlikely congestion problems for any communication between any pair of routers. Its regular structure also allows relatively simple routing algorithms to be used (see section 2.4.2). However, this topology is quite heavy in terms of hardware resources, leading to large wastes for applications where certain pairs of routers are unlikely to exchange data. Though the most popular mesh type is a square grid (meaning that each router has 4 neighbours), some

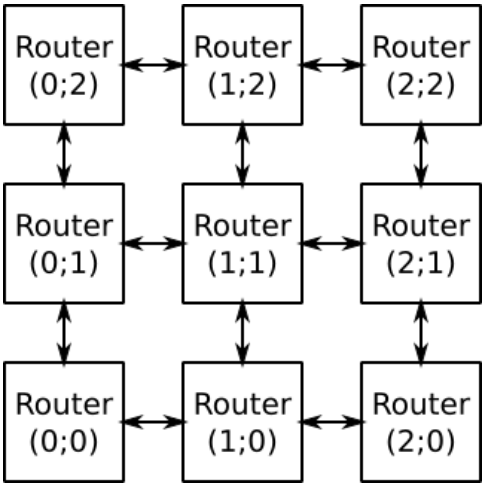


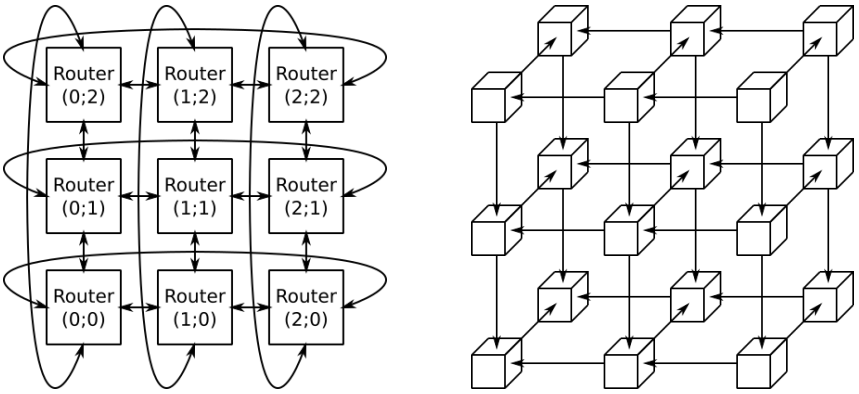
Figure 2.8 – Example of the router interconnection of a 2D Mesh 3×3 Network-on-Chip

other versions exists where the number of neighbours for each router differs. For example, the honeycomb-mesh consists in an hexagonal structure where each router has 3 neighbours.

Extensions of the 2D mesh topology include the torus [23] and the 3D mesh topologies [42, 32, 12] (see figure 2.9). The torus architecture as the same router positions as a 2D mesh except that routers that are on the borders of the structure are also interconnected. While further improving the flexibility of the 2D mesh architecture, the torus structure has the huge drawback of having long delays on these new interconnections. As its name implies, the 3D mesh adds a new dimension to the 2D mesh, meaning that most routers (*i.e* routers that are not on the borders) now have 6 neighbours instead of 4. However, these routers can be hard to implement on 2-dimensional fabrics such as FPGAs.

Another relatively common topology is a tree-based Network-on-Chip (see figure 2.10). While these structures can have great performances if well designed, finding a well suited layout for a given application is a quite tedious task. Also, these structures have to be used with great care, and are not suited for every type of application, since the root of the tree is very much likely to create a bottleneck. Examples of tree-based NoCs can be found in [31] and [48].

Even though common topologies are the most frequently used structures, coming up with a topology specifically designed for a given application can



(a) 3×3 torus (b) $3 \times 3 \times 2$ 3D mesh

Figure 2.9 – Examples of torus (a) and 3D mesh (b) NoCs

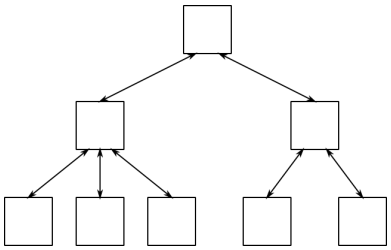


Figure 2.10 – Example of the router interconnection of a tree based Network-on-Chip

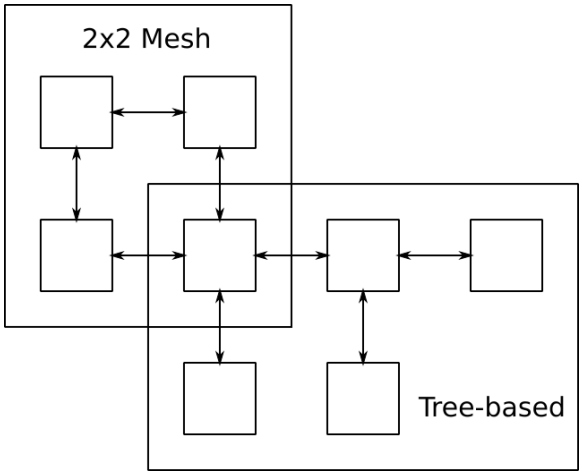


Figure 2.11 – Example of an irregular structure mixing mesh and tree based NoC

be advantageous in order to optimize both the potential throughput a the NoC as well as its resource footprint. In that regard, irregular structures can be used for some designs. Those structures are usually a mix up between subsets of existing topologies. As a result, finding an adequate addressing scheme, as well as a optimal routing algorithm for the topologies can be very tricky. An example of such an *ad hoc* topology mixing mesh and tree based NoCs is provided in figure 2.11.

2.4.2 Routing algorithms

In Network-on-Chips, the routing algorithm corresponds to the way data is transmitted from its origin router to its destination router. The most common characteristics for routing algorithms are explained below.

Switching type

Two main types of switching on NoCs can be distinguished: circuit switching and packet switching. In the circuit switching case, the whole path from the origin to the destination router is reserved beforehand. All router interconnections involved in this path remain closed to other communication until the origin is done sending data. An example of a NoC based on circuit switching is the Æthernal NoC [29]. On the other hand, algorithms that implement a packet switching (for example the Hermes NoC [46]) strategy

require the routing information to be provided for each data packet. The routing scheme is then re-evaluated on a per-router basis. Circuit switching can be quite advantageous when large chunks of data often have to be transferred from one router to another, as the routing path is set only once whatever the data size is instead of updating it for each packet in the case of packet switching. However, the fact that many paths are closed during the whole transactions means that it can potentially lead to congestion problems. In the case where most packets do not have the same destination, a packet switching approach is thus preferred.

Deterministic or adaptive routing

In the case of a deterministic approach, the path that a packet will follow is only determined by its origin and destination routers. This means that two packets that have to be transmitted from the same origin router to the the same destination router will necessarily have the same routing, regardless of if that path is currently overloaded. Two popular methods for a deterministic routing are the source routing (for example the *Source Routing for NoC* algorithm [37]) and the X-Y algorithm [20]. In the case of source routing, the origin router specifies the path that a packet must follow from one end to another. For a X-Y algorithm in the case of a 2D mesh topology, all routers addresses are specified as their X-Y coordinates on the grid. Each router that a packet reaches compares its own address to the destination address of the packet. If its X coordinate is less (resp. more) than the one of the destination of the packet, it sends the packet to the router to its right (resp. left). If both X coordinates are equal, the same test is ran on the Y coordinate (*i.e.* sends the packet to the router above (resp. below) if its Y coordinate is smaller (resp. greater) than the one of the destination router). If both Y coordinates are also equal, that means the packet has reached its destination and is thus sent to the local interface of the router (*i.e* usually the functional unit connected to that router). This algorithm is really straightforward to implement, as well as resource friendly, as only two comparators are needed for each router.

On the other hand, adaptive algorithms route the packets based on the current state of the Network-on-Chip. For example, an adaptive routing algorithm can be based on the congestion of the links in the NoC [40, 22]. In that case, a packet can be sent on a longer path (in terms of nodes) if the shorter one is saturated, reducing its latency, and also decreasing the global congestion of the NoC. Other adaptive algorithms can be used in order to take broken links into account in circuits where faults or failures are

likely to occur [52, 34]. These algorithms are typically more complex than the deterministic ones, and also usually require some heavy mechanisms to monitor the state of the NoC.

Finding a suitable Network-on-Chip architecture for network monitoring applications, as well as dimensionning it to allow enough functional units and high enough data rates considering our flexibility and throughput constraints will be a key part in conceiving our architectures.

2.5 Conclusion

In this section we presented the background and state-of-the-art on flexible designs on FPGAs. This led us to experiment with bitstream relocation to add flexibility for architectures supporting network monitoring applications. However, bitstream relocation remains a very rarely used technique because of the difficulty of performing this task. As a result, providing an automated design flow dedicated to this technique will be one the key challenges of this thesis, as making this technique transparent to a designer would allow for the use of its potential.

In order to be able to efficiently host network monitoring applications, which have both high flexibility and throughput requirements, a suitable communication structure is needed. As a result, Network-on-Chips will be investigated for network monitoring. As NoCs have not been used in the past for network monitoring applications, the dimensionning and choosing the characteristics of our NoC accordingly to characteristics of these applications will have to be done. Also, to our knowledge, no existing architecture combines bitstream relocation with Network-on-Chips. As bitstream relocation involves strong constraints regarding the reconfigurable modules, adapting the NoC architecture in order to ease up the use of relocation will be required.

Chapter 3

Automated design flow for bitstream relocation on Xilinx FPGAs

As seen in 2.3, bitstream relocation is a promising technique for highly flexible reconfigurable designs on FPGAs, as it can drastically decrease the bitstream footprint, both in terms of generation time and memory space required. In application domains such as network monitoring, both generation time and memory space can become critical. Indeed, generation time can greatly impact development and prototyping time, which is not acceptable for network applications that are always evolving. Memory space can impact reconfiguration latency, as many partial bitstreams to store results in bigger memories yet usually further away from the target. As network monitoring covers critical applications such as attack mitigation, keeping the reconfiguration latency low is required. Integrating bitstream relocation in architectures dedicated to network monitoring can thus be highly advantageous.

3.1 Motivation

While bitstream relocation is a promising technique to enhance flexibility on highly reconfigurable designs, strong constraints are required for it to be performed. Meeting these requirements is a long and error prone task and often requires a deep knowledge of reconfigurable systems and the specific FPGA target. Moreover, it is very likely that this task would have to be performed again when a modification is made to a reconfigurable module,

or when a new module is added, as changing the resources requirements of the modules can make the former results of this task obsolete. This means that manually fulfilling the constraints related to relocation is highly discouraged, which in our case is further emphasized by the fact that designs targeting network monitoring applications are likely to be updated quite often. Relocation is thus still a very underused technique because of its deployment difficulty. Making this process transparent to a designer then feels mandatory in order to make efficient use of bitstream relocation. However, this has still not been automated yet. Providing a fully automated design flow that supports bitstream relocation could thus be highly interesting in order to be able to fully take advantage of the benefits that this technique can offer.

In this chapter, we will present a new fully automated design flow dedicated to bitstream relocation on Xilinx FPGAs. As relocation involves many steps that are technology reliant, it is possible that the flow would not be compliant with other manufacturers FPGAs or even with future Xilinx technologies. This design flow is based on the Xilinx ISE Design Suite (as more recent tools from this vendor are missing some key tools in order for relocating to work), scripted versions of techniques already available in the literature, and new algorithms and techniques for steps that still haven't been addressed yet. The general overview is described in section 3.2, our new techniques that address previously unresolved steps specific to relocation are presented in section 3.3, while tests and results are provided in section 3.4.

3.2 General overview of the proposed design flow

In order for bitstream relocation to become a viable technique, an automated design flow that makes it transparent to the user is required. Our design flow aims at being user friendly, as no prior knowledge of relocation is needed, and it requires even less skills than a traditional DPR design flow. It is also worth noting that once all inputs have been provided and a configuration has been run, no user intervention is required, a simple building command based on the standard `make` is enough to provide all the needed outputs to run the design on the target.

3.2.1 Inputs

In order to use our design flow, a designer has to provide several specific files.

An RTL description of the static part , in which all reconfigurable instances must be instantiated as black boxes is required. In order for the synthesis tool not to trim these instances, this attribute has to be specified (in VHDL in this example): `attribute box_type of reconfigurable_module : component is "black_box";` In order to be able to constrain the locations of the interfaces with the relocatable regions, this `black_box` attribute must also be specified to the added_LUTs component (see section 3.2.2) as well as a lock_pins attribute: `attribute lock_pins of added_luts_8_v7 : component is "true";`

An RTL description of every dynamic module also has to be provided. The port list of that description must be identical to the reconfigurable component previously declared as a black box in the static part. As a consequence, all reconfigurable modules must have the same ports. This means that if a reconfigurable module requires less inputs (or outputs) than another one, useless ones would still have to be declared.

A *User Constraint File* (UCF) is required, that only contains the same information as a traditional non-reconfigurable design flow, *i.e.* physical-logical ports bindings and timing constraints. It is important to note that, while UCFs usually allow the designer to use location constraints in order to fix instances of the design into specific locations, it is highly discouraged to do this while using our design flow, as our floorplanning algorithm (see section 3.3.1) will add location constraints of its own without taking these ones into consideration, potentially leading to conflicts. It is also worth noting that, while traditional DPR design flows require the designer to provide floorplanning information in the constraint file, this is not required with our design flow, as the floorplanning will be automated.

Finally, a configuration file has to be filled with several parameters. Those parameters include:

- the name of the top module of the static part;
- the names of the reconfigurable component and their instances as stated in the static part;
- the number of relocatable regions in the design;
- the name of the UCF of the design;

- the part, package and speed grade of the targeted FPGA;
- the clock period of the design (as explained in section 3.3.1, only single clock designs are supported for now);
- the name of the hard macro used at the static/dynamic interfaces, as well as its required number for both inputs and outputs;
- finally, the list of every reconfigurable module on which a relocation will have to be performed.

3.2.2 Automated design flow

Once all inputs have been provided, a configuration script has to be run in order to set every parameter in all other scripts according to the ones given by the designer in the configuration file. Then, a simple make command is enough to launch the design flow. The general overview of the design flow can be found in figure 3.1. In that figure, the steps that are integrated in the traditional Xilinx DPR flow are highlighted in blue. The steps that are available in the literature and that we have scripted are highlighted in green with dashed lines. Finally, the steps that have not been addressed before are highlighted in red with thick lines.

First, the static and dynamic parts are synthesized separately , similarly to a usual DPR flow. In our case, the tool we use for that step is *Xilinx Synthesis Tool* XST [4].

A floorplanning algorithm then uses the information located both in the synthesis reports of the dynamic modules and in the configuration file in order to find a valid floorplan for the design. The main goal of a floorplanning algorithm is to find a decent placement of the reconfiguration regions on the targeted fabric. Although many algorithms (see [16, 53, 54, 55, 45, 18, 60, 47, 59]) exist for floorplanning in the case of traditional reconfigurable designs, two specificities of relocatable designs prevent us from using them for relocation. First, these algorithms do not have the constraint of identifying identical regions, which means that the output of these floorplanning algorithms will not likely be suitable for relocation. Second, most of these floorplanning algorithms tend to limit the fragmentation of the static part in order to provide a decreased resource waste. This leads to floorplans where reconfigurable regions are located right next to each other. However, as relocation forces us to apply the `PRIVATE` constraint on relocatable regions

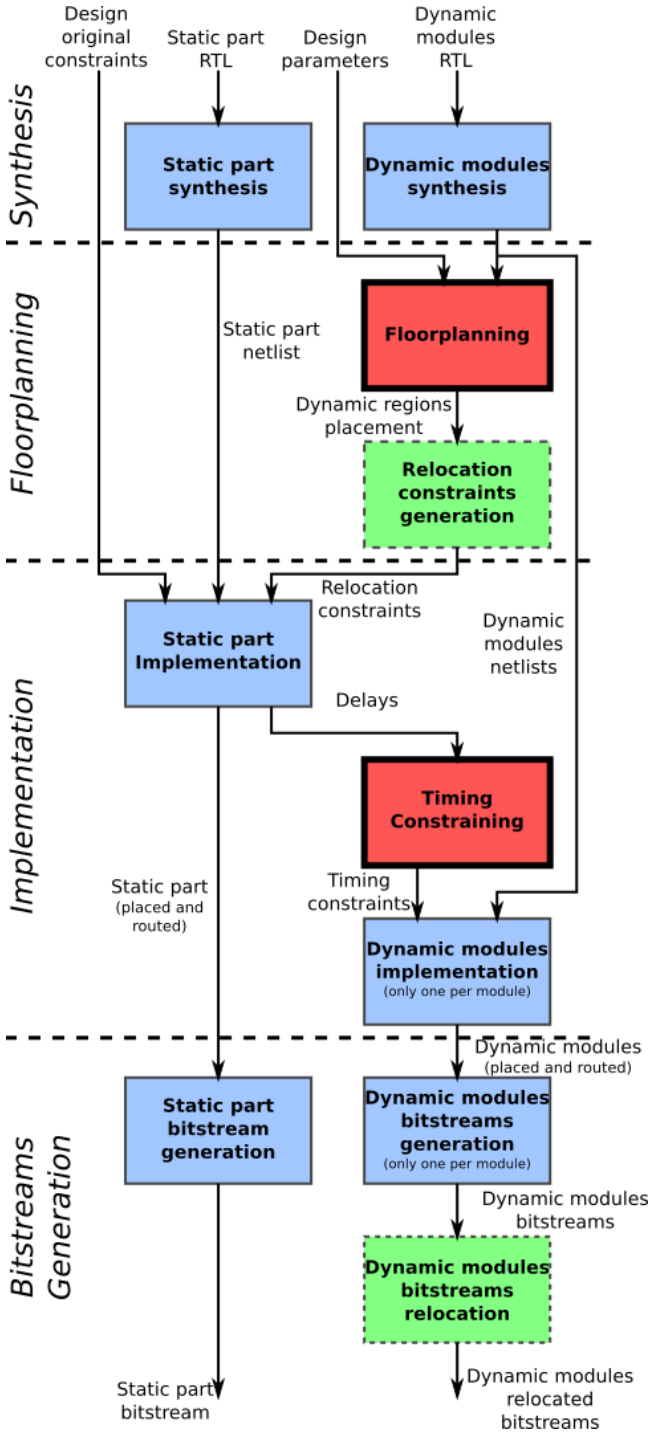


Figure 3.1 – General view of the proposed design flow

(see further), the static part will not be able to use any resource inside them. This can be problematic in the case where regions are adjacent, as it can cause severe congestion problem to the static part if some of its elements have to be placed between the relocatable regions.

Thus, we decided to develop a new floorplanning algorithm specially thought for bitstream relocation. More details about that algorithm can be found in section 3.3.1.

A constraints generation script then sets all constraints relative to the relocatable regions accordingly to the method detailed in [25]. According to this paper, these requirements are:

- (1) origin and destination must have the same shape, size, and arrangement in terms of resources;
- (2) the partition pins between the static part and the dynamic regions must be placed at the same place relatively to the relocatable regions;
- (3) the static part must use the same resources (routing and logic) among all relocatable regions.

The constraint (1) is taken care of by our floorplanning algorithm, as fulfilling it is actually its purpose.

The constraint (2) can be solved by using location constraints for both inputs and outputs. In our case we decided to put all inputs on the left border of the regions, and all outputs on the right border. As we prevent all non-reconfigurable resources from being inside a relocatable region (see constraint (3)), we are sure that all resources are homogeneous among a single column, which means that we can make sure that both the left and right borders of a region are entirely made of slices. For inputs, the script looks for the starting slice of the region written by the floorplanning algorithm in the UCF, and the ending slice for outputs (reconfigurable regions are described as follows: `"AREA_GROUP "partial_region_$region_index" RANGE=SLICE_X$starting_abscissaY$starting_ordinate:SLICE_X$ending_abscissaY$ending_ordinate;"`).

Each pin can be set to a specific location using the following constraint: `PIN "$instance_name.$pin_name" LOC=SLICE_X$slice_abscissaY$slice_ordinate;` . However, as each slice contains 4 LUTS that can be used as partition pins, specifying which one will be used is also mandatory, which can be done as follows: `PIN "$instance_name.pin_name" BEL=$bel_name;`.

This also means that we can stack up to 4 partition pins per slice. Thus, the following is run for all regions:

```
$ # current_x and current_y are set to the starting
$ # abscissa and starting ordinate of the current region
$ # i is the index of the current region
$ while [ $placed_inputs -lt $number_inputs ]
$  do
$     free_luts=4
$     while [ $free_luts -gt 0 ] && [ $placed_inputs -lt
$number_inputs ]
$     do
$         echo "PIN \"rp_inst_$i.module_in<$placed_inputs>\"
LOC=SLICE_X${current_x}Y${current_y};" >> $UCF
$         case $free_luts in
$             4) BEL=A6LUT ;;
$             3) BEL=B6LUT ;;
$             2) BEL=C6LUT ;;
$             1) BEL=D6LUT ;;
$             *) echo "invalid number of free luts" ;;
$         esac
$         echo "PIN \"rp_inst_$i.module\_in<$placed_inputs>\"
BEL=${BEL};" >> $UCF
$         free_luts=`expr $free_luts - 1`
$         placed_inputs=`expr $placed_inputs + 1`
$     done
$     current_y=`expr $current_x + 1`
$ done
```

It is worth noting that the clock input of the reconfigurable part does not need to be constrained, as clock signals use a dedicated routing structure.

The constraint (3) is caused by the fact that, when reconfiguring a region, every previous part will be overwritten. In the case of a traditional reconfiguration, this is not a problem, as the resources inside the region that belong to the static part will just be reconfigured the same way they were. However, when relocating a bitstream, the resources that belong to the static part but located inside the relocatable region will be overwritten by resources that belong to the static part but in the origin region, potentially causing a mismatch. In order to prevent this from happening, we have to make sure that if the static uses resources inside the relocatable regions,

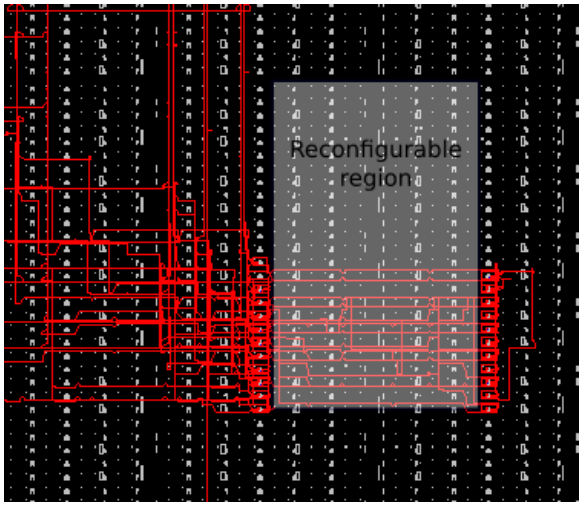
it will use the same resources the same way among all relocatable regions. However, this can be really tricky, as designers usually do not have that much control over the placement and routing of the static part. So, it is far more convenient to prevent the static from using any resources inside the relocatable regions altogether. This can be achieved by using the `PRIVATE` constraint from the PlanAhead tool available in the ISE design suite (figure 3.2 shows how the PRIVATE constraint acts on the routing of the static part). This is actually the sole reason why our design flow is not compliant with the more recent tools from Xilinx Vivado, as this constraint still does not have an equivalent in this design suite.

However, this can not prevent some signals to cross the static/dynamic boundary. Indeed, the signals that form the interface between the static part and the dynamic regions (*i.e.* the inputs of the input partition pins, and the outputs of the output partition pins) have to cross that border, and belong to the static part. The solution proposed to this in [25] is to add LUTs in the signals path right next to the partition pins. As the part of these signals that will be in the dynamic regions will always have to be routed to a LUT right next to them, they will be routed the same way across all relocatable regions. In our design flow, this is done by using a hard macro (see figure 3.3) next to each partition pin. These hard macros will have to be instantiated in the static part, and the interfaces signals will be split so that the output (resp. input) of the hard macros are connected to the input (resp. output) partition pins, and the signals that were previously connected to the reconfigurable region will now be connected to the hard macros instead. The macros locations are constrained using scripts similar to the ones we used to constrain the locations of the partition pins.

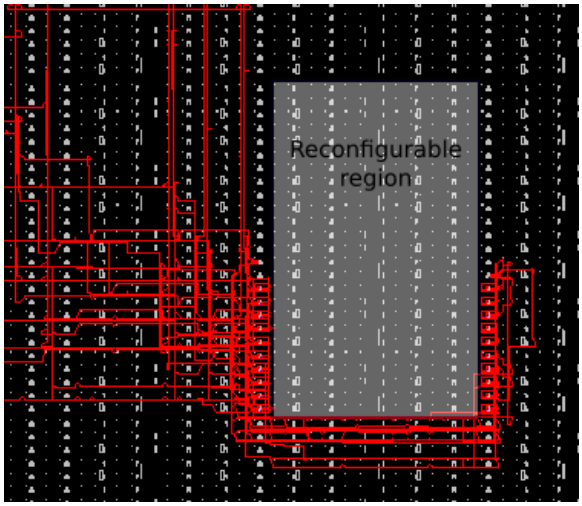
The static part is then implemented by taking into account the constraints previously identified, using the Xilinx ISE design suite implementation tools.

A timing constraining step is then done in order to ensure that the timing constraints for the regions interfaces will be respected for all regions. More details about this are provided in 3.3.2.

The dynamic modules are then implemented in only one region using Xilinx ISE design suite implementation tools, and taking into account the new timing constraints.



(a) usual static part routing



(b) using the PRIVATE constraint

Figure 3.2 – Routing of the static part (a) without using the PRIVATE constraint vs (b) using the PRIVATE constraint

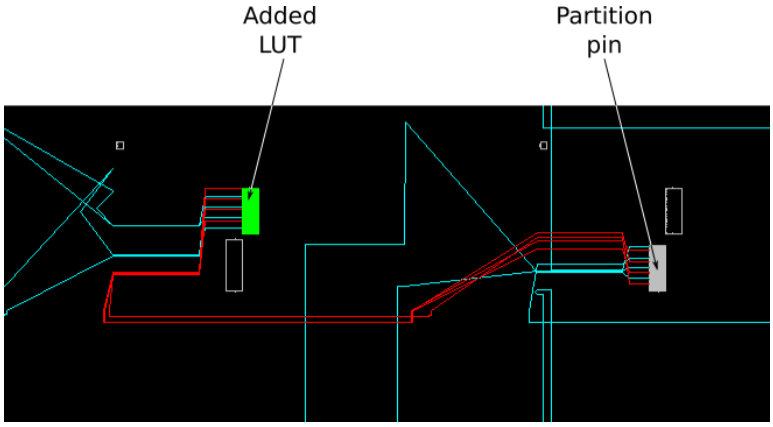


Figure 3.3 – Short connections between the added LUTs and partition pins

Outputs

Finally, the needed bitstreams are generated, *i.e.* the bitstream of the static part, and only one partial bitstream per module, which can be used directly in the region in which they were implemented. However, as modifications will have to be performed on them when a relocation is required, the CRC part of the partial bitstreams has been disabled.

A relocation algorithm is provided, which only changes the FAR(s) in the partial bitstream in order for it to match the one(s) of the destination region. While this algorithm is only available in a software version for now, its code can easily be translated into a hardware streaming module that can be interface with the *Internal Configuration Access Port* (ICAP) which could potentially offer a relocation mechanism with only 1 clock cycle latency.

3.3 New algorithms and techniques for previously missing steps

While existing work allow for a relocation to be possible, there are still some tasks that have not been yet addressed in terms of automation. This means that, while designers can use this technique, it is still quite tedious and require some long efforts, especially since its most difficult part, *i.e.* the floorplanning, has still not been automated.

3.3.1 Floorplanning algorithm dedicated to bitstream relocation

As no algorithm is currently available for floorplanning dedicated to bitstream relocation, we had to develop a new one. As all regions must be identical, we can divide our algorithm into two steps: *pattern* (*i.e.* shape and resources arrangement) choice, and regions selection (*i.e.* selection of occurrences of the selected pattern will be used as relocatable regions).

Pattern Choice

General pattern requirements In order for a pattern to allow bitstream relocation, it has to fulfill several requirements due to limitations on the current FPGA fabrics. On Xilinx FPGAs, the reconfigurable fabric is divided into clock regions. Clock regions are groups of resources that are all connected to the same dedicated clock network. This means that all the synchronous resources that are located in a same clock region will have to share the same clock. This is also the reason that our design flow only supports single clock designs, as our floorplanning algorithm still does not allow for specific clock regions not to be selected to implement relocatable regions, preventing the designer from having any control on clock domains. It is also important to note that every clock region has the same height.

On these FPGAs, the smallest reconfigurable element is called a frame, which is one clock region high (see figure 3.4). A frame actually corresponds to a single word in the configuration SRAM. Since frames are the smallest reconfigurable elements, this means that if a part of a frame is located in a reconfigurable region, the whole frame will have to be reconfigured (see figure 3.5). In traditional reconfigurable designs, only adding parts of frames inside a reconfigurable region would not be a problem since the part of the frame that does not belong to the region would be reconfigured the same way it already was (assuming the reconfiguration is glitch-free, and that involved LUTs are not used in a carry state). However, in case of a relocation, the part of frames that does not belong to the relocatable region would then be reconfigured in another way than it was, likely causing dysfunctions in the design. By forcing each frame to belong entirely either to the static part or to reconfigurable regions, we can then ensure that relocation will not alter the state of static resources. As a result, relocation oriented design flows should always consider only regions that span entire clock regions.

Also, on fairly recent FPGA technologies, such as 7 series, interconnect tiles are added between slices in an horizontal manner, in order for the clock

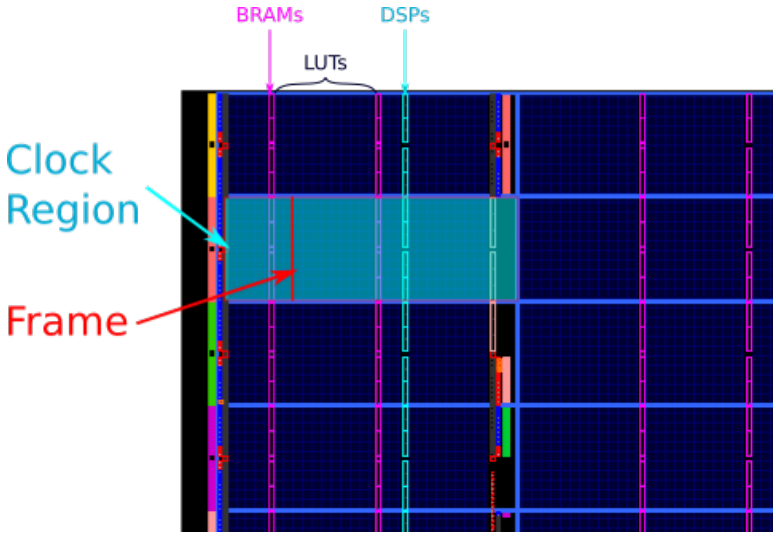


Figure 3.4 – Clock regions and frames on FPGAs

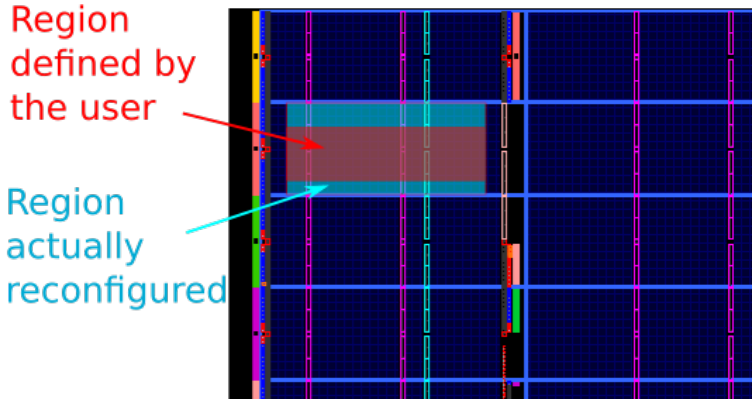


Figure 3.5 – Region actually reconfigured when using only parts of frames

network to be more easily routed. These interconnect tiles come as pairs that can not be in two distinct partitions of the fabric. This means that if one end of a tile is located in a reconfigurable region, the other end must be as well. As a result, each slice column is either on the left side or the right side of a interconnect tile, and thus each slice column is only compatible with a left or right border of a relocatable region (of course, this issue only concerns columns that are located on the border of a region). Thus, for these FPGA technologies, the layout description that we use for our pattern selection must contain information about the compatibility for each slice column with a left or right border. As a result, our pattern finder must only consider patterns where the borders are compatible with the interconnect tiles.

Also, as we decided to place our partition pins on the left and right borders of the relocatable regions, our pattern must have slice columns on their borders.

The last requirement is that each region must be rectangular. This requirement is only induced by our region selection algorithm, and future work could potentially add the possibility of non-rectangular regions, as there is no technology barrier to it.

Identifying potential patterns The only data required for pattern identification is the resource footprint of all relocatable modules in the design. These footprints can be found in the synthesis reports of most synthesis tools. In order for all modules to have enough resources to be implemented in the relocatable regions, only the maximum number estimated for each resource type (LUTs, BRAMs and DSPs) is needed. We also add a 10 percent overhead for LUTs, as modules that needs the maximum number of LUTs in a reconfigurable region will most likely not be able to be successfully routed. That percentage is based on experimental results, as we so far have not seen unsuccessfull implementations of reconfigurable modules using this overhead.

Few pattern identification algorithms based on resource estimations have already been proposed in the literature. The solution presented in [13] looks for all possible patterns that are one clock region high only, and then eliminates the ones that are included in another one. Another solution proposed in [14] consists in expanding a region that is one clock region high until it has enough of each resource type, then shrinks it while it still has enough resources. However, both solutions only consider regions that are only one clock region high. While the latter stops as soon as one valid pattern is

found (regardless of if enough occurrences of the pattern are present on the fabric), we still decided to base our method on this one, as it is easily improved, simple and efficient, and as the former involves many unnecessary steps and is rather complex. Also, this method does not take into account the left/right border compatibility.

An example of our method is presented in figure 3.6. We start on the top left column (one clock region high) of the layout of the FPGA, and shift it to the right until we find a slice column that can be placed on a left border (step a), and we extend the region to the right until it meets all the required resources constraints (step b). We re-extend it until we meet a slice column that can be placed on a right border (step c). We then shrink the obtained region from the left until one resource type is not sufficient enough anymore (step d), and we re-extend it to the left until the next slice column that can be placed on a left border (step e). By shrinking the pattern from the left until one resource is missing, we can make sure that this obtained pattern will not include any other one. This also means that we can search for the next potential pattern starting from the slice column after the left border of the previous pattern (step f).

This process is iterated until we meet the right border of the fabric. Every time a pattern is identical to a previously identified one, it is discarded. This is then iterated again on each row (a row is one clock region high) of the fabric. At that point we are sure that we identified every potential pattern that is one clock region high. This whole process is then reiterated each time by increasing the height by one clock region until we span the whole height of the fabric.

Finally, for each pattern identified, we count its number of occurrences on the target, and the ones that does not appear at least a number of times equal to the number of relocatable regions specified by the designer are discarded.

Choosing the final pattern Once all potential patterns have been identified, we have to choose the one that will eventually be used for relocatable regions. Since there is no available standard metric for evaluating which pattern will provide the best results, we had to come up with empiric criteria to eliminate patterns that could possibly lead to implementation problems.

The first problem we noticed is that a lot of patterns only use a few columns but span a lot of clock regions, resulting in very narrow regions, especially in cases where one resource type is unused. Since narrow regions are likely to lead to congestion problems during implementations, we decided

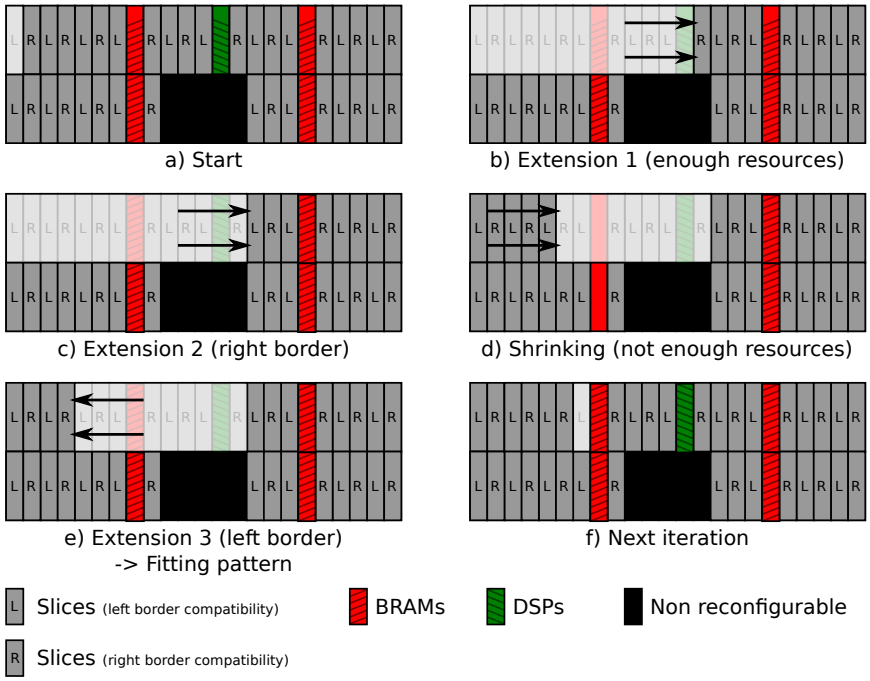


Figure 3.6 – Example of iterative identification of fitting patterns: needs 8 slice columns, 1 BRAM column and 1 DSP column

to first eliminate patterns that do not span the minimum number of clock regions among all patterns.

Another problem that we found is that if the number of occurrences of a pattern on the fabric is low, it is likely that most of these occurrences will be located on the same columns of the FPGA but on different clock regions. This means that the regions that will be selected for relocation will likely be adjacent to each other, causing congestion problems to the static part since it can not be placed nor routed between the regions. Thus, we only retain the patterns that provide the largest number of occurrences on the fabric.

We finally keep the remaining pattern that offers the least wasted resources. In the unlikely scenario where several of the remaining patterns have the same wasted resources, one of them is randomly chosen.

While these criteria have led to successful implementations so far, finding a way to weight all these accordingly to the design's requirements or giving a designer control over the pattern choice could potentially improve the design flow.

Regions selection

Once a pattern has been chosen for our relocatable regions, we have to select, among all occurrences of that pattern on the fabric, which ones will be used as relocatable regions.

Floorplan criteria The selected configuration must respect two criteria. First, regions must not be placed too close from each other, once again to prevent any congestion problems from happening to the static part. Indeed, as the static part has to respect the `PRIVATE` constraint, it is likely that configurations where regions are too close to each other would not give enough space for the static part to be successfully placed and routed. Second, while it is ideal to give a lot of space to the static part, placing regions too far away from each other could also increase delays for signals that must potentially connect two regions together. Placing the regions close to each other would likely improve performances. As a result, our floorplanning algorithm has to find floorplans where relocatable regions are fairly close to each other while still giving enough space for the static part to be implemented. However, the notion of "close" and "far away" in this case is highly dependant on both the FPGA technology and the complexity of the static part. This means that for now, the metrics we present in the next paragraphs are based on experimental results, and that improvements such as taking into account the static part complexity can be made.

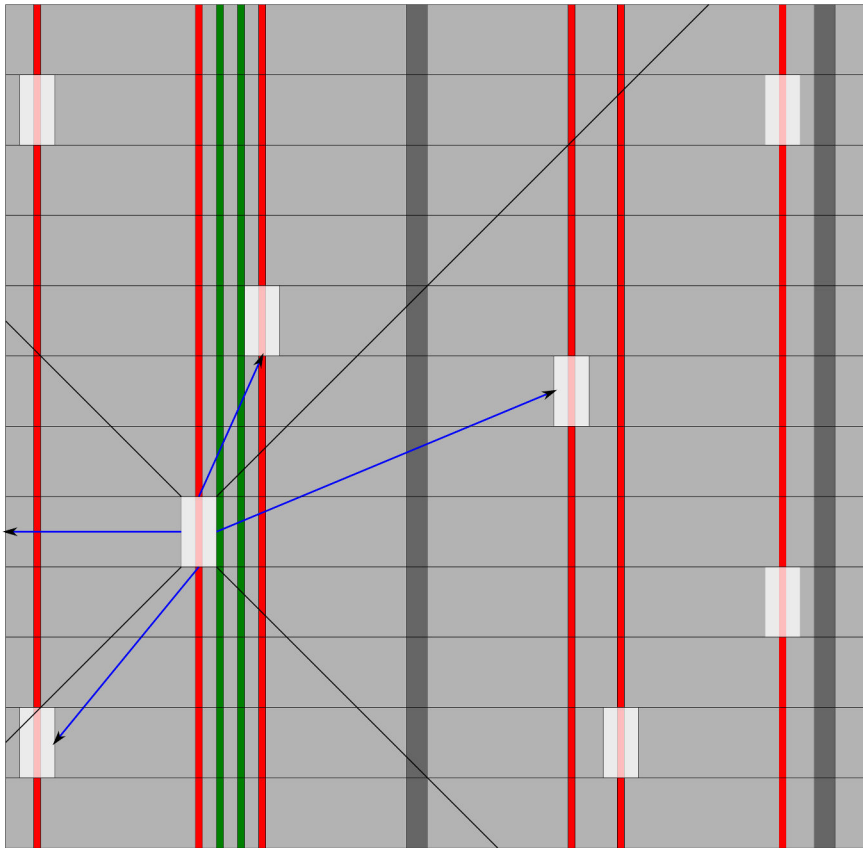


Figure 3.7 – Example of distances computation for one region (pattern is slice-slice-BRAM-slice-slice × one clock region)

Evaluating configurations In order for us to select which configuration will be used for our design, we have to find a way to compare them efficiently. For this purpose, we decided to find a function that we need to minimize. Our goal is that each distance between a region and its closest neighbours are as close as possible to a value that would let the static part be routed. For each region that belongs to the configuration being estimated, we compute the minimal euclidean distance between each border of that region (4, as reconfigurable regions have to be rectangular) and the closest region (or target border) inside a quarter plan located between two rays starting from the border ends and inclined by +/- 45°from an horizontal line (see an example on figure 3.7).

We then run two tests for each distance that we thus obtained. First, if

the distance is less than an empirically defined threshold, a penalty equal to $(distance - threshold)^2$ is added. This way it is unlikely that a design where two regions are too close to each other will minimize the objective function. Second, if the considered distance is the distance between a region and the fabric's border, and if it is greater than the threshold, it is discarded. Indeed, while it is not advantageous to have long distances between regions, there is no drawback for having a lot of space between a region and a fabric's border, as that would greatly ease the routing of the static part.

The function we decided to minimize is the sum of the mean of all kept distances and their standard deviation. Using the mean of the distances ensures that floorplans where distances are too long will be discarded. However, it is still possible to have floorplan where most distances are short, but one or few distance(s) are critically long, hence the addition of the standard deviation.

The threshold we used has been empirically set to the number of slices in a slice column (one clock region high). Indeed, any greater threshold would cause any vertically aligned regions to have at least two clock region heights between them (as relocatable regions are aligned on clock regions). One possible improvement could be to estimate the complexity of the static part so that the threshold could be decreased for designs where the static part is likely to be easily routed.

More penalty functions could also be investigated, because while the square function provides satisfying results, other functions could be more adapted to this approach.

Reducing the computing time Since the number of different configurations to be evaluated can be really high (N regions amongst M occurrences, which has a $\mathcal{O}(M!)$ complexity), doing an exhausting search can take up to a few days. Hence, we decided to perform a simulated annealing in order for our floorplanner to find a valid placement in a decent computation time. At each iteration of the algorithm, one region is randomly swapped with another one which did not belong to the previous configuration, and the temperature follows a geometric progression. Of course, configurations where some regions overlap are immediately discarded, as resources on the fabric can not belong to more than one region. In that case the swap is randomly done on one of the regions that overlap.

We have thus obtained a new floorplanning algorithm dedicated to bitstream relocation which, while still being based on empirically defined met-

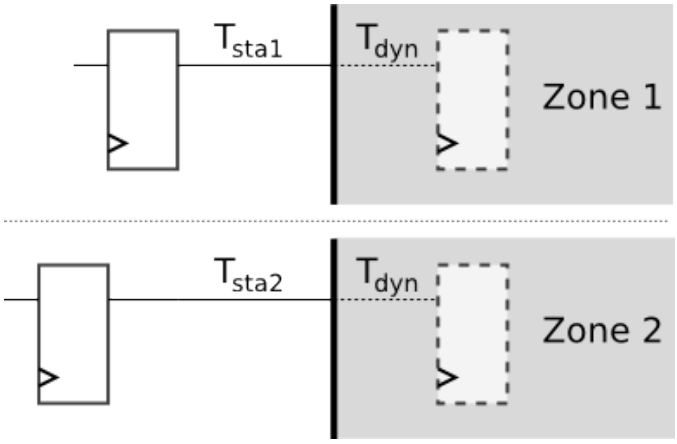


Figure 3.8 – Example of timing differences for two relocatable regions

rics, is able to provide valid floorplans in a quite respectable time.

3.3.2 New timing constraining technique

When performing a relocation without taking any precaution, it is possible to encounter timing problems at the static/dynamic interfaces. It is, as a matter of fact, possible that the timing constraints, that are valid for the origin region, will not be satisfied in the destination region. Indeed, while we can be sure that the delays inside the relocatable regions will be the same for both the origin and the destination region (since both regions are identical and will be configured the same way as the bitstream is the same), there is no guarantee that the delays from the static part to the reconfigurable region, or vice-versa, will be the same across all regions, as the designer does not have control over the placement and routing of the static part (see figure 3.8).

One simple solution to this problem would be to use synchronous interfaces between the static part and relocatable regions, but this would prevent the user from using asynchronous communication protocols (such as a simple req/ack protocol) between the static and dynamic parts.

Instead we propose a new technique that ensures that all the interface delays (*i.e.* from input partition pin to first register, of from last register to output partition pin) inside the reconfigurable regions would be small enough to allow each reconfigurable module to be successfully implemented in any relocatable region of the design.

Once the static part has been implemented using Xilinx *PaR*, the tool

uses *FPGA Editor* in order to get all the delays in the design. Then, for each pin that belongs to the interface between a relocatable region and the static part, the tool finds the maximum delay for that pin among all the reconfigurable regions. The maximum delay found is then used to constrain delays for the dynamic modules implementation using the UCF constraints:

PIN "rp_inst_region_number.module_out<*pin_number*>" TPSYNC = regions_output_*pin_number*;
TIMESPEC TS_from_RM_to_PP_output_*pin_number* = TO "regions_output_*pin_number*" (*clock_period* - *max_delay*) ns;
PIN "rp_inst_region_number.module_in<*pin_number*>" TPSYNC = regions_input_*pin_number*;
TIMESPEC TS_from_PP_input_to_RM_*pin_number* = TO "regions_input_*pin_number*" (*clock_period* - *max_delay*) ns;

As the actual implementation of each relocatable module will now take into account the worst delays for all partition pins, we can be sure that a relocated module will respect the timing constraints in each relocatable region.

3.4 Tests and implementation status

This design flow has been fully implemented, and tested on several Xilinx FPGA targets.

3.4.1 Floorplanning results

As our pattern finder does an exhaustive search of all potential patterns for a given number of resources, we were able to use it in order to find the maximum number of relocatable regions that a design can achieve depending of the resources needed for reconfigurable modules, or, on the other hand, the typical number of resources allowed for reconfigurable modules for a set number of regions. Table 3.1 gives an example of the maximum number of relocatable regions allowed for a design given various resources constraints. It can be worth noting that bigger or more homogeneous targets allow for more relocatable regions, as the probability for a pattern to appear more times on the fabric increases. It is also fairly obvious that smaller modules lead to more potential relocatable regions.

#Slices	#BRAMs	#DSPs	Max #Regions
1000	10	10	30
		40	20
	40	10	10
		40	10
2000	10	10	14
		40	14
	40	10	10
		40	10
3500	10	10	8
		40	8
	40	10	6
		40	6
8000	[0-100]	[0-100]	4
9000	[0-100]	[0-100]	1

Table 3.1 – Maximum number of placeable regions based on needed resources on a Virtex7 690t

An example result of our floorplanner is given in figure 3.9 for a design on a Virtex7 690t, on which the reconfigurable modules need a maximum of 1000 slices, 8 BRAMs and 6 DSPs on 8 relocatable regions. We can see on this example that our floorplanner outputs a fairly regular floorplan, with regions being relatively close while still giving decent space for the static part to be routed.

We finally ran some tests regarding computation time of our floorplanning algorithm. It took 58 seconds to find a pattern and place 15 times on a Virtex7 690t for a resources need of 1000 slices, 10 BRAMs and 10 DSPs (out of 30 possible locations for each region) using an Intel Core I5-2500 CPU @3.3GHz. This computation time is highly acceptable compared to both the time it would have taken to do it manually (typically a few hours) and to the whole design flow time (from synthesis to bitstream generation).

3.4.2 Layout description and supported targets

In order for our floorplanning algorithm to work, we need a description of the layout of the considered target to find patterns and place them on the FPGA. We first considered to use the databases provided by the Torc framework [56]. However these databases do not make the difference between slices M

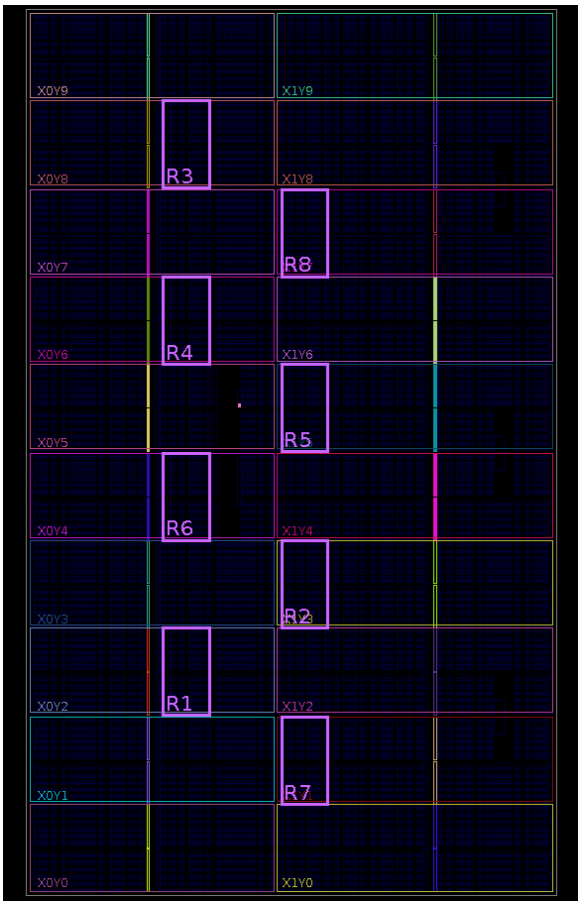


Figure 3.9 – Example of the result of our floorplanning algorithm (8 regions, 1000 slices, 8 BRAMs, 6 DSPs) on a Virtex7 690t

and slices L, which can lead to non-functional relocated modules, nor do they include the distinction as to whether a slice column can be placed on a left or right border of a region.

Many Xilinx FPGAs (for example the xc5vlx110t or the xc5vlx330) have a vertically homogeneous structure (if we only consider reconfigurable resources), which means that for these targets, only the description of one line (one clock region high) and the number of lines are needed (along of course with parameters that depend on the targeted series, such as the number of each resource in one column). Thus, for those targets, only the horizontal arrangement of resources has to be fully described, as we can consider all non-reconfigurable resources to belong to one type, which can not be included in a reconfigurable region. This presents the main advantage that a new layout can quickly be described and added to the supported targets.

However, some other Xilinx FPGAs (for example the xc7690t) present exceptions on their layouts (*i.e.* one line can be different from the others). For these targets, the layout must contain the structure of each row.

This layout description has already been included in the flow and tested on three targets, which are the Virtex5 xc5vlx110t and xc5vlx330, and the Virtex7 xc7vx690t.

3.5 Conclusion

In this chapter we presented a fully automated design flow dedicated to bitstream relocation on Xilinx FPGAs. This design flow is based on techniques already available in the literature, on the Xilinx ISE Design Suite, and new algorithms specially developed for answering issues that had not been addressed yet, which are a floorplanner and a timing constraining step both dedicated to relocation. This design flow aims at being user friendly, as no prior knowledge of the FPGA target nor of bitstream relocation is required from the designer. Also, no user intervention is required once all inputs have been provided. The additional time induced by the whole flow does not go above a few minutes, which is negligible compared to both the traditional design flow time (synthesis and implementation), which can usually take up to a few hours, and the time it would require to use relocation manually, which we claim can take up to a few days. This flow has been tested and validated on three FPGA targets, which are the xc5vlx110t, the xc5vlx330 from the Virtex5 series, and the xc7vx690t from the virtex7 series.

However, while our design flow provides functioning results, it is still in a early development stage, meaning that a lot of improvements can still

be made. First, a lot of parameters in our floorplanning are set empirically. Using complexity analysis on our designs could help adapt 1) the 10% LUTs overhead used in our pattern finder to the dynamic modules complexity and 2) the threshold used to estimate the ideal distance between relocatable regions in our region selection step to the complexity of the static part. Investigating other penalty functions in our distance computation function could also potentially provide better floorplanning results. Second, our design flow enforces several constraints that may not be acceptable depending on the application. Indeed, our design flow is not compliant with multi-clock designs. Adapting it so that it is not reserved for single clock designs could greatly increase its number of potential applications. Also, our flow can only output floorplans with rectangular regions, which can sometimes induce large resource wastes. Supporting non-rectangular relocatable regions could potentially result in designs that offer a better area efficiency. Long terms improvements include compliancy with newer Xilinx tools, such as the Vivado Design Suite, as well as with FPGAs from other manufacturers.

Finally, this design flow has been the subject of an accepted paper presented in the *19th Euromicro Conference on Digital System Design (DSD 2016)*, entitled "Autoreloc: An Automated Design Flow for Bitstream Relocation on Xilinx FPGAs".

Chapter 4

Using Network-on-Chips for network monitoring

As seen in section 2.4, Network-on-Chips seem to be an interesting communication architecture in network monitoring applications, as those applications need communication structures that can support both high throughputs and high flexibility. Other common architectures lack either flexibility or throughput capacity. Indeed, while being able to support really high data rates, point-to-point solutions are by nature inflexible, and therefore not suitable for network monitoring applications. Bus architectures, while being very flexible, can not provide an high enough throughput, as only one communication is allowed at any given time, which is likely to create a bottleneck in the case of network monitoring, especially when the number of components connected to it increases. For example, the AXI interconnect bus from Xilinx has a maximum theoretical throughput of 3.2Gbps [5].

Since both point-to-point and bus solutions are unable to provide both the flexibility or throughput required for flexible network monitoring architectures, Network-on-Chips seems like an ideal solutions for these architectures. However, this interconnection structure has not yet been used in the case of network monitoring, as existing network monitoring architectures did not focus on both flexibility and high performance at the same time. This means that the choice of a NoC topology, algorithms and dimensions will have to be investigated specifically for network monitoring applications. These NoC characteristics must be able to sustain several (depending on the number of links available on the targeted board) 10Gbps links.

Our NoC architecture must also be compliant with bitstream relocation. Particularly, bitstream relocation greatly benefits from having relocatable

modules as simple as possible. Also, using bitstreams relocation means that it can be difficult for a module to know where the other modules are located on the NoC. Adding mechanisms that decides where a module has to send its output packets in the relocatable units would then add significant complexity to them, decreasing the effectiveness of relocation, as more complex units lead to less compatible regions. This means that such mechanisms should not be included in the relocatable modules. As a result, using bitstream relocation with NoCs would be eased by having a mechanism that provide the sequence of what relocatable modules a packet has to go through inside that packet. Moreover, in order to achieve an high flexibility, it is likely that our functional units will have to be parameterable. While partial reconfiguration can be used to change the functionality of a given region, some easier changes (for example, changing the offset at which an operation is performed) does not require such an heavy mechanism. Furthermore, having a dedicated structure in parallel of the NoC for parameterization information would unnecessarily increase the complexity of the design. This means that both the sequence of modules a packet has to go through as well as the parameters of these modules should be provided in the packets.

Also, the fact the bitstream relocation benefits from using simple modules means that any operation that is common to all our functional units (for example, communication with routers or data buffering) should rather be placed outside the reconfigurable regions.

In this chapter we will investigate Network-on-Chips as a potential communication architecture for network monitoring applications. The choice of the topology, algorithms and NoC dimensions that we will use (related to the FPGA target of our project) will be presented in section 4.1. A protocol overlay that is able to both provide the whole sequence of treatments that a packet must go through as well as to parameterize those treatments will be presented in section 4.2. Finally, a generic router to functional unit interface that takes care of all operations that are common to all modules, thus easing their design, will be presented in section 4.3.

4.1 Choosing the NoC characteristics and dimensions

4.1.1 Overview of the project board

Dimensionning our NoC and choosing its characteristics can vastly depend on the targeted board. For this project, we have at our disposal a NetFPGA-

SUME board [6] [64]. This board is part of the NetFPGA project, which is a collaboration between industrial companies (most notably Xilinx and Digilent) and academics, which aims at providing an open platform for research in network applications on FPGAs. This board is equipped with a Xilinx Virtex7 690t (package 1761, speed grade -3) FPGA interfaced with 4 10Gbps SFP+ interfaces, with additional extensions able to support up to ten 10Gbps SFP+ interfaces. As our version of the board uses 4 10Gbps SFP+ transceivers, our goal is to provide an architecture able to sustain a 40Gbps throughput, while potentially being able to be scaled to 100Gbps. This FPGA is made up of:

- 693120 logic cells;
- 52920 Kbit of BRAM;
- 3600 DSPs;
- 30 GTH transceivers;
- Other features of this board are not relevant in the scope of this work.

The NetFPGA project also provides scripts to easily implement the interfaces with the transceivers, using the Xilinx *AXI 10G Ethernet IP* [1].

4.1.2 NoC characteristics

Topology

Due to the large variety in incoming traffic, it is likely that each pair of functional units can have to exchange data, which is further emphasized by bitstream relocation (as a specific functional unit can be placed anywhere on the NoC). As a result, an high interconnection density is required. This means that tree-based or irregular topologies are not suited for our case. Indeed, the tree-based topologies are by nature unable to efficiently manage a traffic distribution that can potentially go massively between leaves that are far away from each others [49]. In this case, it is likely that the root of the tree would create a bottleneck.

On the other hand, mesh topologies provide an higher density in terms of interconnections between routers. This makes that topology well suited for our case, which needs to be able to ease the routing between any pair of routers.

Switching type

The fact that it is likely that most packets incoming from one input of the board will not have to go through the same functional units (*i.e.* will not go to the same routers) implies that a packet switching approach is more suitable than a circuit switching one. Indeed, circuit switching approaches are advantageous when a lot of consecutive packets have to be routed to the same router. However, the time required to set the communication channel has a negative impact when that setting has to be done for every packet, which is likely to happen in network monitoring applications. On the other hand, while packet switching approaches are not optimal when routing a lot of consecutive packets to the same destination (as routing information has to be provided for each packet), it provides better results than circuit switching when consecutive packets have different destinations. As a result, Network-on-Chips for network monitoring should use packet switching.

4.1.3 Generation tool

In order to easily generate a NoC for our design, we decided to use the Atlas NoC generation environment [7] [43]. Atlas is an open-source tool that can generate mesh NoCs that has been validated on FPGAs. Among other things, the user can configure the dimensions of the NoC, the routing algorithm, and the width of the data channels.

The Atlas generation tool has been chosen mainly because it is an open-source tool, and supports both mesh topologies and packet switching. While doing a brand new NoC from scratch would have been possible, using a tool that has already been validated on FPGAs allowed us to quickly get a functioning Network-on-Chip.

The Hermes NoC [46] has been chosen because it supports both the topology and switching type required for our application, and is available in the Atlas generation tool. It is a 2D mesh topology that uses five bi-directional links per router (four communicating to the neighbor routers, and one local port that will be use to connect our functional units). Each port of the routers has a buffer that can temporarily store data in order to minimize the risks of congestion on a router. The Hermes NoC also claims to have a low area overhead, which can be advantageous in order to give the functional units enough resources to perform more complex tasks.

Routing algorithm

The Hermes NoC in Atlas supports two routing algorithms: XY and West-first. As stated in [44], the XY algorithm is faster than the West-first algorithm to route packets on an Hermes NoC, as well as being less complex (in terms of resources used). As a result, we decided to choose to use an XY routing algorithm on our Network-on-Chips.

4.1.4 Dimensionning the NoC

While choosing the number of routers in our Hermes NoC is dependant on the considered application, and on the parts of the design that are not part of the NoC (as the size of the NoC is limited by the resources available for the NoC), choosing a data width mostly depends on the board on which we want to implement our designs. The first thing we have to consider in order to choose our data width is that we should be able to saturate the optical links that are connected to the board. The 10Gbps transceivers IP is set to be driven by a clock at 156.25MHz. As stated in section 3.3.1, using our design flow for bitstream relocation makes it hard to use different clocks inside a same design. This means that we should use the same clock for our Network-on-Chip and for our transceivers IPs. Fortunately, synthesis estimations (from XST) indicate that using a 156.25MHz should not be an issue, as the estimated maximum operating frequency for the Hermes NoC is about 190MHz. These synthesis estimations are independent from the number of routers, provided that the NoC's dimensions are over 3 by 3, because then every type of router (middle (4 neighbours), side (3 neighbours) or corner (2 neighbours)) is already represented (see figure 4.1). It also appeared that these estimations are independent of the data width, which is to be expected as these are synthesis estimations.

This means that in order to saturate an interface from the output of a router (or vice-versa, that the input of a router can sustain the traffic incoming from an interface), we need a data width of $10Gbps/156.25MHz = 64$ bits. However, the Hermes NoC in Atlas uses credit based transactions (it also supports handshake transactions, but the Atlas documentation recommends the use of credit-based), which require 2 cycles to transmit a *flit* (packets are divided into flits, which are chunks of $data_width$ bits). This means that we actually need a data width of 128 bits instead of 64.

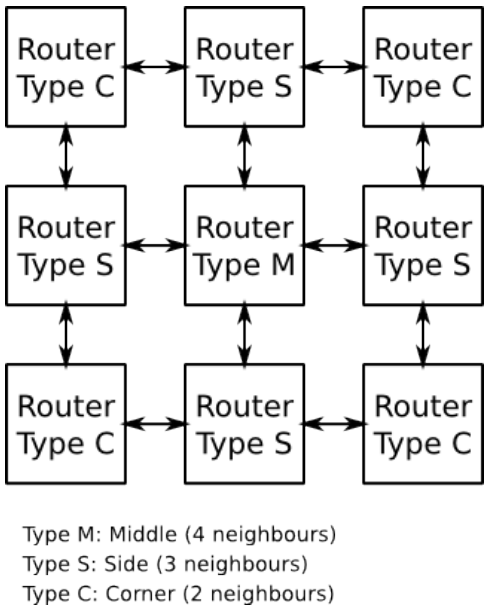


Figure 4.1 – Types of routers in a 2D-mesh Network-on-Chip

4.2 Protocol overlay

4.2.1 Motivation

As bitstream relocation benefits from having modules as small as possible (as it increases the number of compatible relocatable regions), any operation that is not directly related to the specific task of a module should be separated from that module. This means that we want our functional units to be as simple as possible, thus the decision concerning the next functional unit it has to send a particular packet should not be included in them. Moreover, the use of bitstream relocation can make it complicated for each module to know where the other ones are located on the NoC, whereas it can be more practical to provide that information to a single control unit outside of the NoC. As a result, our architectures would greatly benefit from having a mechanism where the list of treatments a packet has to go through is contained inside the packet.

Also, while partial reconfiguration can be useful in order to modify the whole functionality of a functional unit, minor modifications such as changing a parameter of a functional unit does not require such an heavy mechanism. Indeed, waiting a few milliseconds (which is the typical time of a

partial reconfiguration) to make a modification that would require only a few clock cycles such as for example modify a value stored in a register would be highly inefficient in terms of timings. Furthermore, having to reconfigure a functional unit in order to change one of its parameters would need a bitstream for each possible combination of its parameters, which would result in unnecessarily big storage space and long implementation times for that functional unit. Moreover, using the NoC structure to transmit parameter information instead of a dedicated structure means that no other hardware resource is required. Being able to provide the parameter of a functional unit in the packet it has to treat would then greatly reduce the reparameterization times as well as not require unnecessary mechanisms.

Finally, adding the sequence of functional unit a packet has to go through as well as their parameters inside a packet can drastically increase the header/data ratio. This means that an increasing part of the available bandwidth would then be used by the header transmission on the NoC. For example, in packets in which there are as many header flits as data flits, half the bandwidth would be used by the header. In these cases, we would only be able to use 5Gbps on our 10G optical links instead of 10Gbps.

In this section, we will describe a new protocol overlay for the Hermes NoC. This protocol is able to provide the sequence of functional units each packet has to go through. It can also include the parameterization information required by the functional unit. Finally, a solution to limit the header over data ratio will be presented.

4.2.2 Providing the sequence of treatments to packets

Initial header

On the Hermes NoC implementation from Atlas, the actual data packet is preceded by a two-flits header that contains the destination router as well as the size of the packet (see figure 4.2).

This header has to be recomputed after each functional unit.

Adding sequence information in the packets

In order to provide additional information without having to modify the NoC, the added information should be taken into account in the existing header. This means that the first flit should still indicate the next router to which a packet has to be sent, and the number of new information flits should be added to the packet size flit.

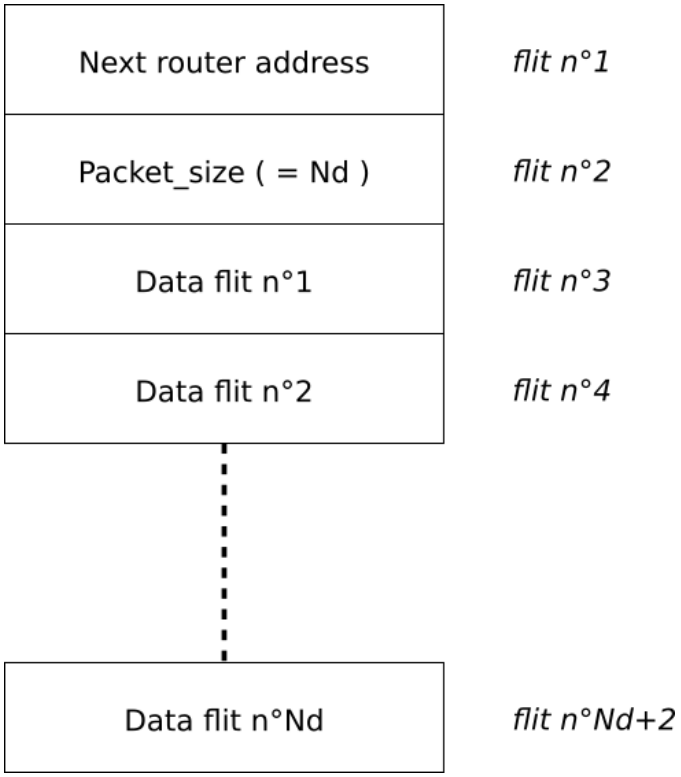


Figure 4.2 – Organization of a packet in the Hermes NoC (Nd = number of data flits)

In order to provide the sequence of functional units a packet has to be sent to, we obviously need the list of the addresses of routers connected to these functional units. However, we also need to provide the information about the number of flits that we added, as only the data flits must be sent to the functional units. For that reason, we add a flit containing the number of routers after the packet size one, so that we can know how many flits to skip before sending the data to the modules.

This new packet organization is shown in figure 4.3.

This sequence has to be updated after each functional unit (see an example in figure 4.4). In this example, the *source* module, which is connected to the router 0x0000, sends a packet consisting of 4 data flits (0xD100, 0xD200, 0xD300 and 0xD400) that must go through the functional units F (connected to the router 0x0100) and G (router 0x0101) before being sent to the sink connected to the router 0x0001. The source must thus forge the packet (packet A) as follows:

- The next router addresses:0x0100;
- The packet size: 7 (4 data flits plus 3 additional header flits);
- The number of remaining routers addresses: 3;
- The two additional routers addresses: 0x0101 and 0x0001;
- The four data flits.

When the functional unit F receives that packet, it removes the first flit (as it represents its own router), replace it by the first remaining router address in the sequence (0x0101) which is the next functional unit to send the packet to (G). It also decrements the packet size and the number of remaining routers by 1, and performs its treatment on the data flits before sending the packet (packet B) to the next functional unit.

Similarly, when G receives the new packet, it replaces the first flit by the next router address (0x0001) and decrements the packet size and the number of remaining router addresses. After having treated the data flits, it finally sends the new packet (packet C) to the sink module.

4.2.3 Parameterization information

While using partial reconfiguration to change the functionality of a region on the NoC, this technique requires mechanisms that are too heavy for it to be advantageous in the case of small modifications on a component. For

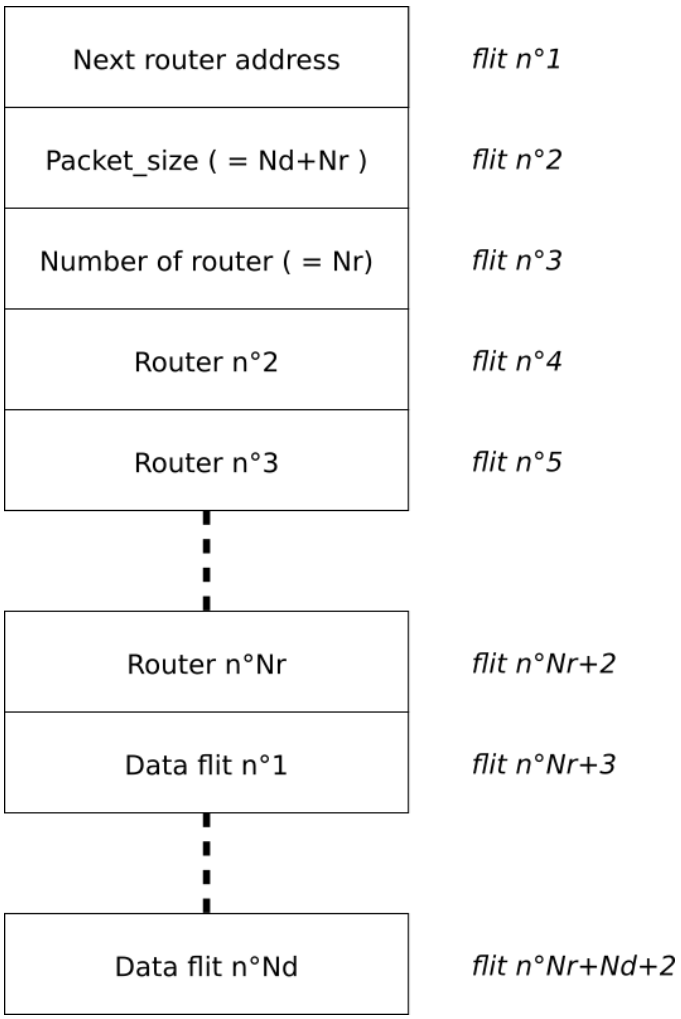
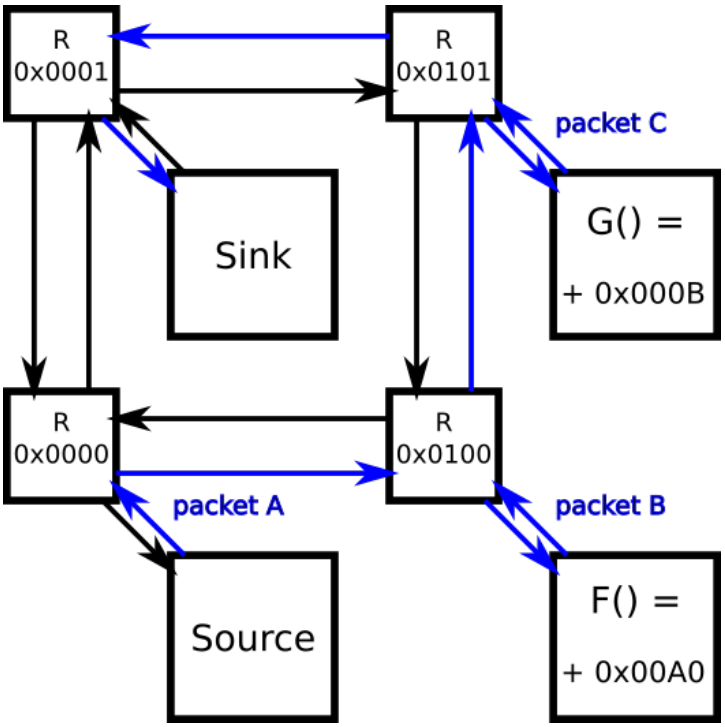


Figure 4.3 – Providing the sequence of treatments in packets



(a) Path of the example packet

0x0100
0x0007
0x0003
0x0101
0x0001
0xD100
0xD200
0xD300
0xD400

(b) Initial packet (A)

0x0101
0x0006
0x0002
0x0001
0xD1A0
0xD2A0
0xD3A0
0xD4A0

(c) Packet after first treatment (B)

0x0001
0x0005
0x0001
0xD1AB
0xD2AB
0xD3AB
0xD4AB

(d) Packet after second treatment (C)

Figure 4.4 – Example of the sequence update on a packet going through 2 functional units

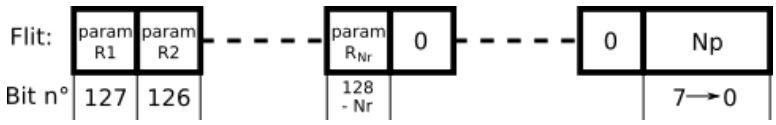


Figure 4.5 – Description of the parameterization flit

example, changing the value of a single parameter on a component would not only involve a reconfiguration time, which can typically take up to a few milliseconds, but also to generate a whole new partial bitstream with the new parameter value. On the other hand, using just a simple register to store and change the value of that parameter would only require a few clock cycles and no additional partial bitstream.

Providing parameter information in the packets would allow to use the same hardware resources for both data and parameters, meaning that no dedicated structure is required for transmitting parameters, thus not increasing the complexity of the design.

Similarly to the router information, each flit containing parameterization information has to be taken into account in the packet size flit. Also, as not every module will necessarily have to be reparameterized as each packet goes through it, we need a way to indicate to which module is a particular parameter flit supposed to be sent. For this purpose, an additional flit has been added in which individual bits are used to indicate which next routers in the packet have to be reparameterized, starting with the most significant bit. A logical '1' means that that component will have to be reparameterized, a '0' means the component keeps its current parameterization. The least significant bits are also used to indicate the number of parameter flits. The description of that parameterization fit is given in figure 4.5.

The new header with the parameterization part is described in figure 4.6.

4.2.4 Multipackets

As we increased the header size of our packets on the NoC, we also decreased the throughput of actual data that can be transferred on the NoC. Indeed, the protocol overlay that we added is seen by the NoC as actual data. For example, if the header is the same size as the data part, only half the theoretical throughput of the interfaces can be achieved, and even more so as the ratio header/data can be even higher. Indeed, the minimum size of an *Internet Protocol* (IP) packet is 64 bytes, which is equivalent to 4 data flits on the NoC as our data width is 128. Cases where the incoming

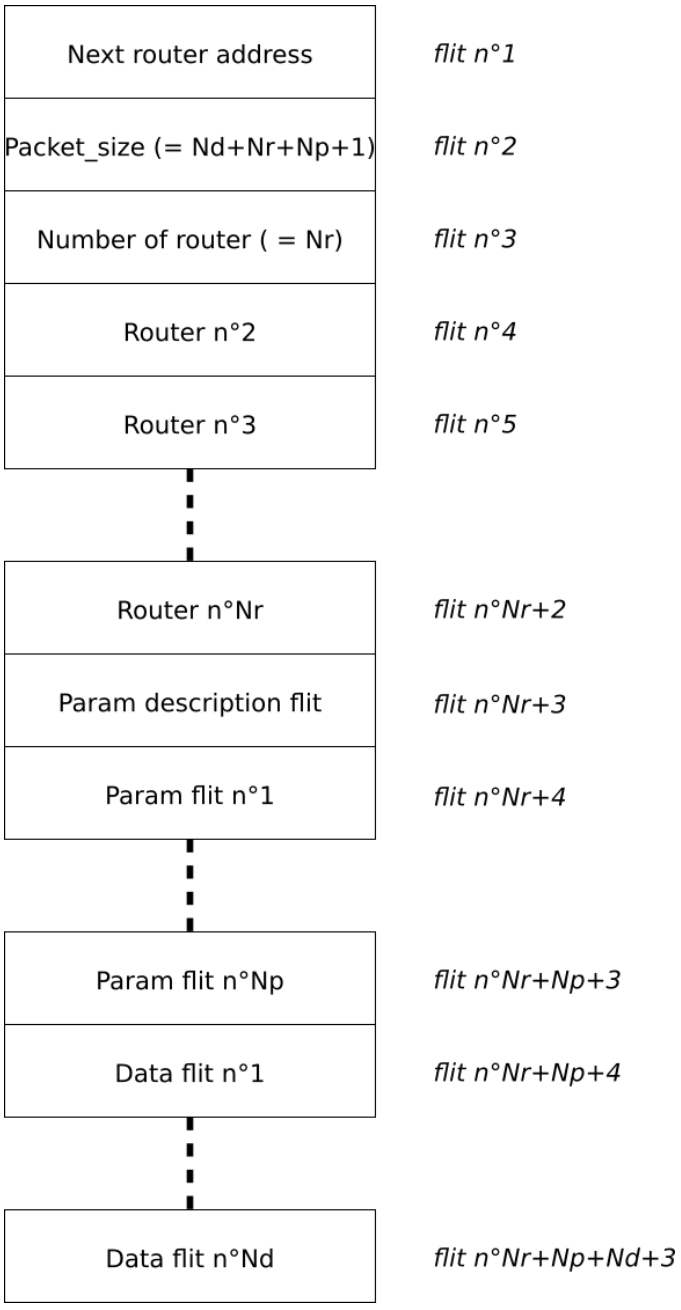


Figure 4.6 – Providing the parameterization information in packets

traffic consists in a lot of small packets could then results in unacceptable throughputs.

Hence, we need a mechanism in order to limit the header/data ratio in our packets. On a network monitoring application, it is highly unlikely that a single packet would be the only one to have to go through a particular set of treatments. Indeed, in network monitoring applications, packets are usually grouped up in flows. Flows are a sets of packets that share common characteristics. For example, all packets that have the same source, destination, source port, destination port, and protocol could be grouped in a flow. As they share common characteristics, it is likely that packets from a same flow would have to go through the same functional units. For example, in the case of a DDOS attack detection system, the malicious packets are usually sent massively. All suspicious packets going through this system would then follow the same treatment sequence.

This means that the header of all these packets would be the same. Grouping data of all these packets would then be possible and would greatly decrease the header over data ratio, actually increasing the throughput of the NoC. This would however require a mechanism that stacks and delays packets in the case where packets from different flows can be interleaved.

However, when actually transmitted to the functional units, the whole data chunk has to be resegmented into the original packets. This means that the size of every packet has to be provided in the header. As the flit containing the number of routers is very unlikely to use more than one byte (as the maximum value is then 255, and 255 routers is unreasonable to expect on an FPGA), this leaves 15 bytes unused inside that flit. Also, IP packets have a maximum size of 1500 bytes (which is 96 flits), so one byte is enough to store the size of a single packet. As a result, we decided to store the size of each packet on a remaining byte of that flit.

This allows to group up to 15 IP sub-packets inside a single NoC packet. Experimental results (see chapter 5) have shown that, with packets of minimal size, grouping 15 packets together allows us to use 85% of the maximal theoretical bandwidth of the 10G optical links.

The final organization of a packet is given in figure 4.7. It is noteworthy that only the number of data of the first packet (Nd_1) must not be equal to 0, as it is not mandatory to group up packets that way.

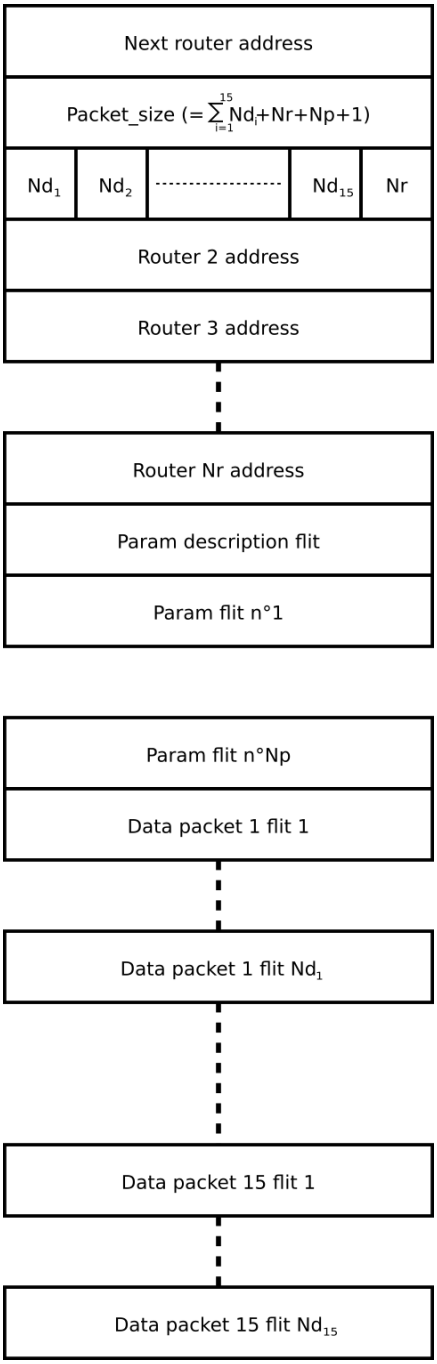


Figure 4.7 – Final organization of packets on the NoC

4.3 NoC/Functional units interface

Similarly to how bitstream relocation benefits from not having decision tasks about routing inside the relocatable modules, the communication mechanisms at the interface of the relocatable modules should be as simple as possible. Also, if any packet buffering has to be done, doing it inside the relocatable modules would require additional BRAMs and thus complexify the modules. Finally, the protocol overlay we described in section 4.2 involves implementing additional steps (as the header has to be reorganized after each functional unit) which do not have to be located inside relocatable modules.

For these reasons, a generic interface between the NoC and the relocatable regions has been developed. This interface provides all the mechanisms that are required in order to make our protocol overlay transparent to both the routers and the functional units. This means that the interface must be responsible for reforging the header of the packets so that it is ready to send to the next router address of each packet, and for transmitting data flits from the router to the functional unit and vice-versa. It is also in charge of reparameterizing the functional unit accordingly to the information provided in the header. It must also include buffering mechanisms so that a new packet can be accepted while the functional unit is still treating a previous one, in order to limit potential congestion issues on the NoC.

4.3.1 Interface overview

The general overview of our interface is given in figure 4.8.

This interface is made of buffers (see section 4.3.2) and 3 modules. The acquisition module (see 4.3.3) is responsible for managing packets incoming from the routers and storing them into the input buffers. It also modifies the headers according to the protocol described in section 4.2, as well as to reparameterize the functional unit if needed. The unit management module (see 4.3.4) is responsible for transmitting data flits from the input buffers to the output buffers. Finally, the release module (see 4.3.5) transmits the header (previously modified by the acquisition module) and the new data from the output buffers back to the router.

4.3.2 Buffers

In order to limit congestion on the NoC, we decided to add buffers between the routers and the functional units, both ways. This way, the router can

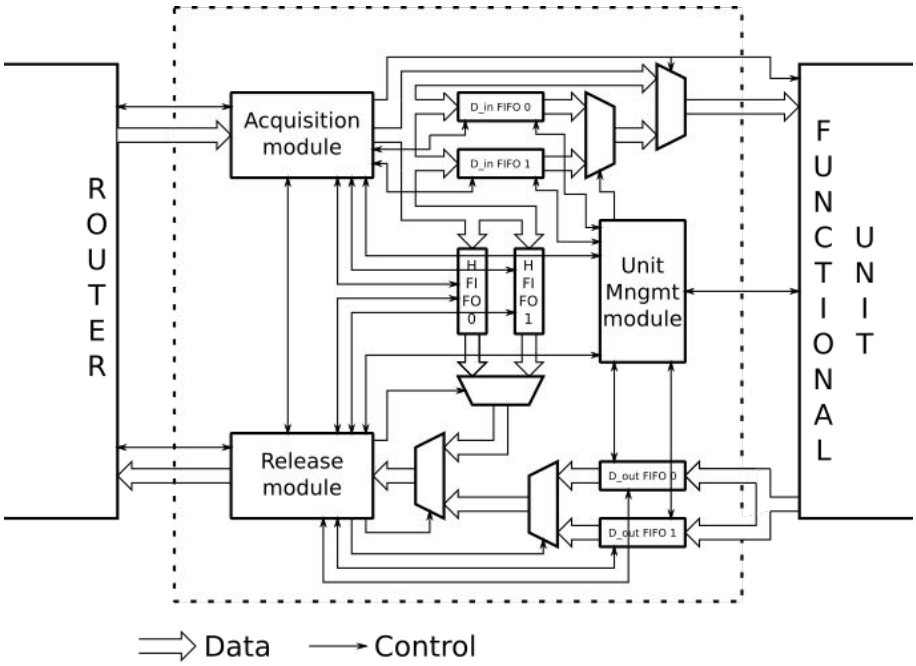


Figure 4.8 – Overview of our router to functional unit interface

still free its load towards the interface even when the functional unit is treating another one. Similarly, the functional unit can start working on a next packet even when the router is already busy. As only the data has to be transmitted to the functional unit, headers and data have different buffers. These buffers are implemented as *First-In-First-Out* (FIFOs). Each data buffer has been dimensionned so it is able to store the maximum number of data flits in a NoC packet (which is 15×96 flits), so that if a packet is being accepted by the interface, we make sure that it can be fully transmitted.

We also decided to double those buffers so that while one buffer channel (0 or 1) is currently being treated, another one can start being filled. As when the router first starts requesting to send a new packet, no information about the packet size is given, using only one buffer could potentially lead to situations where the router starts sending a packet while there is not enough space in the buffers. In that case, the router would be blocked until enough space has been freed in the buffers. During that time, all packets that must go through that router (but not to that interface) would be blocked as well. Using two buffers and denying the router from beginning to send a packet would then allow the router to route another packet while enough space is

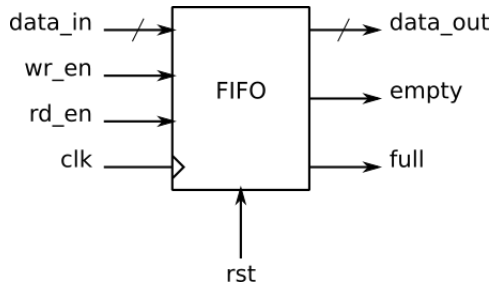


Figure 4.9 – Port description of our FIFOs

made, limiting congestion problems.

The port description of the FIFOs is provided in figure 4.9.

Data_in buffers connections

Both of our data_in buffers have their *data_in* inputs connected to the *data_out* of the acquisition module.

The selection of which buffer to write in is made thanks to the write enable signals (*wr_en*) also coming from the acquisition module. The *empty* signal is used both for the acquisition module to know when an input buffer is free to store a new packet, and for the unit management module to verify that all data of a packet has been sent to the functional unit.

Both *data_out* signals are connected to a 2:1 multiplexer. Which buffer output is selected is controlled by the unit management module, which also sends a read enable (*rd_en*) to the corresponding FIFO. The selected output is also connected to another multiplexer controlled by the acquisition module that can bypass the data flow to send parameterization flits to the functional unit.

The *full* signal is used as a security for the acquisition module not to write any more data if the FIFO is full (which should not happen as our buffer have been dimensionned so that they can store a packet of the maximum size).

Data_out buffers connections

The data_out buffers both have their *data_in* inputs connected to the output data from the functional unit. The unit management module uses write enable (*wr_en*) signals to select to which buffer the data will actually be sent. The *empty* signal communicates to the unit management module whether or not a buffer can store a new packet.

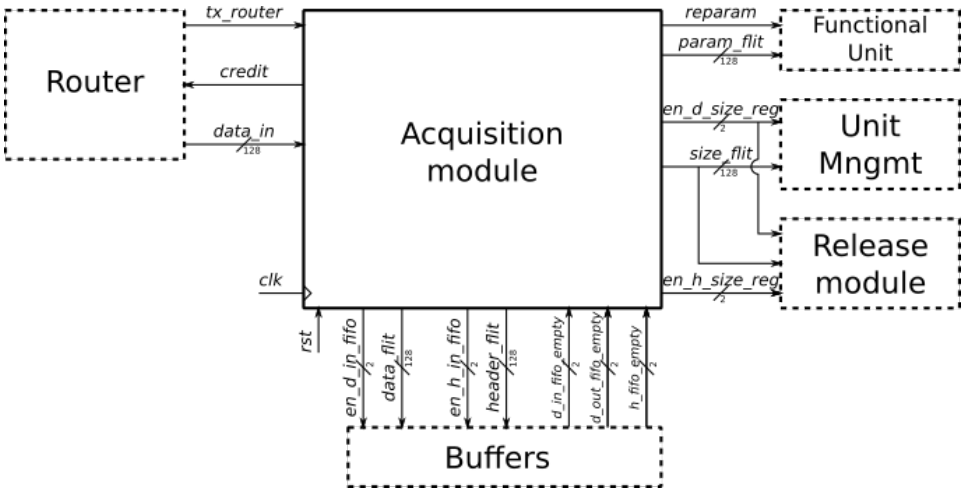


Figure 4.10 – Connections of the acquisition module

Data_out signals are sent to a multiplexer controlled by the release module, that can select in which buffer to read using the read enable *rd_en* signals. The output of that multiplexer is also multiplexed with the output of the header buffers in order for the release to choose which part (header or data) of a packet has to be sent to the router.

Similarly to the data_in buffers, the *full* signal is used as a security to not overwrite useful data.

Header buffers connections

The header buffers are used to store the headers, previously rearranged by the acquisition module, and release them once the packet has been treated by the functional unit. The write signals and input data are provided by the acquisition module, while the read signals and output data are controlled by the release module.

4.3.3 Acquisition module

The connections of the acquisition module are provided in figure 4.10.

The acquisition module has three main purposes. The first one is to manage incoming packets from the router and store them in the buffers. The second one is to prepare the header so that it is ready once the packet is done being treated by the functional unit. The last one is to reparameterize the functional unit if needed.

As information about the actual content of packets is given flits by flits, the implementation of this module is based on an *Finite State Machine* (FSM). This FSM will not be fully described here, as it has 18 states, however its main steps will be covered.

Incoming flits management

Flits are transmitted using a credit-based protocol, which means that the router activates its *tx_router* signal and waits for the *credit* signal from the interface to be activated to transmit the current flit. This is done for each flit until the packet is fully transmitted. As our buffers are dimensionned so that they can store a full packet, the credit value is set to 1 until the whole packet is received. When the router requests sending a new packet, the acquisition module checks if one of the two buffer channels is free (meaning that all three buffers of that channel, data_in, data_out and header, are empty). If no channel is free, the credit value is set to 0 and reactivated once a channel has been emptied. Once a channel has been selected, it will be used until the whole packet has been received.

Header management

The first flit to enter the interface is the router address. This flit is discarded, as it will not be part of the header of the packet that will be released on the NoC.

The second flit is the packet size flit (the packet size will be noted Ps). This one will also not be sent to the header buffers, as the size will have to be recomputed after it has been treated by the functional unit. However, this flit has to be stored in a register, as it is used in order to know when the packet has been fully received.

The third flit contains both the number of routers (Nr) and the size of each sub-packet (Nd_i), which are to be stored as well in registers. Nr should in this case not be equal to 1, as this would mean that we are in the last router of sequence, in that case the packet should be sent to an output of the NoC and not to a functional unit.

The next router address is then received. As this will be the first flit of the output packet, we can now start filling the header buffer. If $Nr = 2$, no other router will be next in the header, which means that we can skip to the parameterization description flit step.

Else we go to a state that gets all next routers. This state uses a down-counter that is first set to $Nr - 2$, and that is decremented every time a

new flit is received. During the first iteration, $Nr - 1$ is sent to the header buffer, as it will be the number of router addresses in the output packet. After that first iteration, the router address received in the previous iteration is sent to the header buffer. Once that downcounter reaches 0, the FSM then transitions into the parameterization description flit step.

Parameterization management

When receiving the parameterization description flit, regardless of if the previous state has been reached by the FSM, there is still one previously obtained flit that has not been stored if the header buffer yet. If $Nr = 2$ (meaning that the previous state was getting the next router), the flit that has to be sent is the new number of routers $(Nr - 1)$. Else the last router of the sequence has to be sent. During this the last byte of the incoming flit is stored, as it represents the number of parameterization flits (Np). If it is null, the FSM transitions directly into the data flits reception. At this point, we can also compute the number of data flits (Nd) in our packets:

$$Nd = Ps - (Nr - 1) - Nd$$

While this could have been computed earlier with the information contained in the third flit (by adding all the Nd_i), this way only requires 3 subtractions instead of up to 15 adders. The way our parameterization description flit is build, we have to check its most significant bit. If it is equal to 1, that means that the functional unit has to be reparameterized, in which case we go to a reparameterization state. Also, if it is equal to 1, this means that one parameter flit will not be in the output header. In that case, the number of parameterization flits Np is decremented by one. If the most significant bit is equal to 0, but Np is not null, this means that some of the following functional units will have to be reparameterized. We thus go to a state that stores all the parameter flits. In any case, all the bits except the ones from the last byte have to be shifted to the left to update the parameterization description flit for the output packet. During that step, a flit containing all the sizes of the sub-packets is sent to the unit management module. The number of header flits is also sent to the release module.

During the reparameterization state, the module uses the *reparam* signal both to force the functional unit to reparameterize itself, and to switch the input of the functional to the current incoming flit. An important thing to note is that reparameterizing a functional unit requires that all previous packets that had to go through that unit with the former parameterization must be done being treated. As these packets can come from different paths,

it is possible that a packet that has been sent before the one containing the new parameters can reach the router after the new one. As we have not come up with a mechanism that can control this yet, the solution we have temporarily adopted is to prevent the sources from sending packets to that interface during an idle time (for now set to 1000 clock cycles) before sending a new parameterization. This way it is very likely that all packets that had to use the former parameterization would be done being treated. The FSM then transitions to a state that manages all other parameter flits (if $Np > 1$) or start managing the data flits.

Similarly to the state that manages the remaining routers, the one that manages the remaining parameter flits uses a downcounter set to $Np - 1$. Every time a new flit is received, the downcounter is decremented by 1, until it reaches 0, at which point the FSM transitions to managing data flits. At the first iteration, the previously updated parameterization description flit is sent to the header buffers. After that iteration, the parameter flit received at the previous iteration is sent.

Data flits management

During that step a downcounter is first set to Nd. Each time a new flit is received, the downcounter is decreased by 1. At the first iteration, one header flit still remains to be sent. If Np was greater than 1, that flit is the last parameter flit, else it is the parameterization description flit. For the other iterations, the flit previously received is sent to the data buffer, until the downcounter reaches 0, at which point the FSM transitions back to the initial state (first router reception) while sending the last remaining data flit.

4.3.4 Unit management module

The unit management module is pretty straightforward. It uses the flit containing the sizes of the sub-packets in order to send each packet from the data_in buffers to the functional unit, using *Start* and *End* flags to outline each sub-packet. Every flit is then modified and sent to the data_out buffers by the functional unit, that also transmits *Start* and *End* flags to the unit management module. Using these flags, the unit management module can then compute the new size of each sub-packet and send it to the release module so that it can insert them in the new header.

Once all sub-packets have been treated, it also triggers the release module that it can start sending the buffer channel to the router.

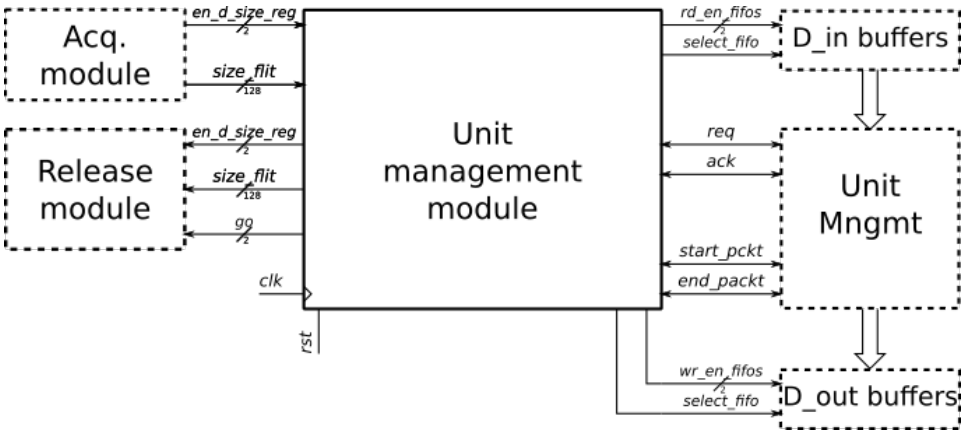


Figure 4.11 – Connections of the unit management module

The connections of the unit management module are provided in figure 4.11.

4.3.5 Release module

Once the unit management module triggers the release module that all data flits are available in the buffers, it first sends the first flit in the header buffer (the first router flit) to the router. It then uses the sizes it received from both the acquisition module and the unit management module to compute the new packet size and send it as the second flit. It then combines the next flit in the header buffer and the sizes of the sub-packets from the unit management module to form the third flit of the packet. The rest is straightforward as it only needs to empty the header buffer then the data buffer to send the whole packet.

The connections of the release module are provided in figure 4.12.

4.4 Conclusion

In this chapter we investigated the use of Network-on-Chips for network monitoring applications. We chose to use the Atlas tool for Hermes NoCs generation, as it an open source tool that can generate NoCs that have already been used and validated on FPGAs, as well as having characteristics that are well-suited for our design choices for network monitoring applications, such as mesh topologies and packet switching. We gave an example on

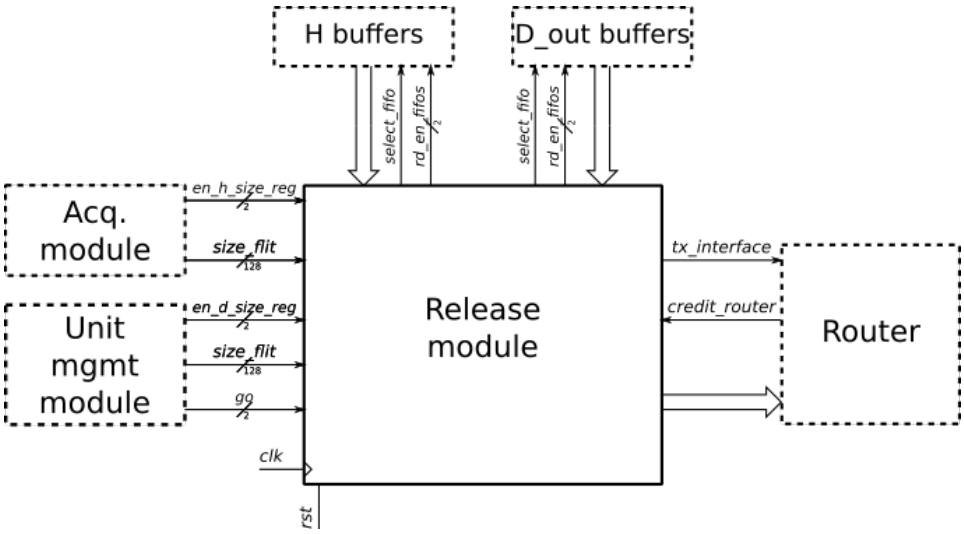


Figure 4.12 – Connections of the unit management module

how to dimension the NoC for a specific board (in this case the NetFPGA-SUME).

As bitstream relocation is still in its infancy, existing NoC architectures and protocols involve mechanisms that are not suited for the use of relocatable regions. We thus extended the original Hermes NoC protocol in order for it to solve issues that are raised by bitstream relocation constraints. The protocol overlay that has been developed is able to provide the whole sequence of functional units that a specific packet has to go through on the NoC, by giving the whole sequence of router addresses to which those functional units are connected. This protocol also allows to provide parameter flits in the packet, so that small modifications of functional units do not require a whole reconfiguration. Also, as the increase of the header size induced by that overlay can be detrimental in terms of achievable throughput on the NoC, we add a mechanism that allows to provide multiple data packets in a single NoC packet, provided that all those packets must be applied the same sequence of treatments. However, this overlay has several limits. It requires a control mechanism that is able to provide the whole sequence of functional units each packet has to go through. Also, the use of the multipackets functionality is conditioned by whether packets can be grouped up together, which depends on the application.

Finally, we provided a generic interface between routers and functional units. This interface provides buffers in order to decrease the local conges-

tion of the router to which it is connected. It also handles all the new mechanism involved in our protocol overlay. This interface is particularly useful when combining NoCs and bitstream relocation, as it moves the communications from the relocatable units to the interface and makes all the mechanisms introduced by the protocol overlay transparent to both the router and the functional unit. Both these characteristics allow for much smaller relocatable units, which increases the possibilities of relocations on the design.

Chapter 5

Test case: traffic generator

In order to test, calibrate or validate new equipment for network monitoring, it is mandatory to provide some realistic traffic to it in order to verify its behavior. However, it is obviously unacceptable to insert some untested equipment into a real network, as it can jeopardize that network if the equipment is faulty. One easy solution to test an equipment would be to record some traces of real traffic and then reinject it in the equipment under validation. However, this method has two main drawbacks. The first is that if a specific scenario has to be tested, it needs to be present in the trace that is provided to the application under test. For example, testing an application that mitigates a specific type of network attack would require the trace to contain that type of attack. This method would thus require to store a lot of different traces, which can lead to huge storage requirements (for example, storing traces on a 10Gbps for one hour can require up to 36Tb of storage). The traces would then need to be analyzed, which leads to the second drawback of this method: ethics. With net neutrality being a central topic in today's networks practices, storing and analyzing traces of real internet traffic can become problematic. Obtaining real traces can thus be difficult for legal purposes, and network providers are unlikely to transmit traces to third-parties. In order to be able to provide realistic test conditions for newly developed network monitoring applications, traffic generators have been introduced.

Traffic generators usually rely on a model that aims at being realistic, or specifically tailored to test a specific feature (synthetic traffic). These generators can be used for example to validate the functionality of an application, test the maximum throughput that it can sustain or calibrate it if parameterizations are needed. This allows developers to test their network

monitoring applications without requiring real traffic traces. Also, not using traces gives the advantage to easily test the application's reaction to more specific traffic scenarios, depending on the degree to which the generator can be parameterized.

However, commercial traffic generators tend to be highly expensive, and while they usually provide the ability to parameterize the traffic they generate, the fact that they are not open-source make that they can not be adapted to more recent traffic trends, generate a model for newer attacks, or be expended to generate a new kind of traffic they were not intended for.

Software generators usually focus on providing realistic traffic models, as the flexibility offered by software solutions allows them to handle various statistical features to generate models. For example, [57] provides a framework to generate models based on probabilistic features. Swing [61] extracts probabilistic distributions from existing traffic by observation construct generation models. However, both do not support traffic generation for throughputs that are greater than 1Gbps. [19] provides realistic models that can go up to 10Gbps, however that throughput is only achieved if packets are big enough (the throughput for minimum size packets (64 bytes) drops to 6Gbps), meaning that this solution is not suitable for testing scenarios involving a lot of small packets.

On the other hand, FPGA-based generators usually focus on providing higher data rates. FPGEN [51] can achieve 5Gbps, however it has some really poor configuration capabilities, as primary features such as IP address or transport layer ports can not be specified. Another interesting approach is presented in [30]. It is an open-source traffic generator implemented on a previous board of the NetFPGA project. The model implemented in this project is based on a skeleton/modifiers structure. Packet skeletons, which are base IP packets, are sent to the generator which then sends them to a sequence of parameterizable modifiers that modify the skeleton to achieve a specific synthetic traffic. For example, if a designer wants to test the reaction of its design to a port scan (which is a technique used by pirates to find security flaws that consists in sending request to every port to identify the ones that are open), he can set the skeleton to a request packet with the port number set to 0, and then use a modifier that increments the port number field each time the skeleton is resent. Similarly to FPGEN, this traffic generator can achieve 10Gbps throughputs for large packets, but observe a drop when packets get too small.

One particular flaw of this generator is that each time the number or the sequence of modifiers needs to be changed, a whole reimplementation is needed, which is very time consuming. This means that if a new equipment

is under test, the tester either has to plan for all test cases in advance, or wait for a few hours between each test. Adding flexibility to this approach could thus improve the time required to start a new test.

In this chapter, we will present an extension to the work done in [30]. The goal is to apply our architecture combining bitstream relocation and Network-on-Chips to improve both the flexibility and the throughput of that approach. Using our architectures, if a new traffic has to be generated, we would then only need to reconfigure the modifiers according to new model instead of requiring a full reimplementation, which would reduce the time between each test from a few hours to a few milliseconds. Also, no reimplementation would then be required if the connectivity between modifiers has to be changed, as it is natively supported by our NoC architecture. This is also an interesting use case to provide a proof of concept for our architectures. Indeed, this test case does not involve complex problematics such as reconfiguration timings, allowing us to validate the functionality of our mechanisms on a rather simple test case.

5.1 Model presentation

The model that we base our test case upon is made of a set of streams. Each stream consists in a skeleton, which is the base raw packet that includes both the Ethernet and IP header, which goes through a list of modifiers. The modifiers provided in the open-source project [8] are:

- a signed incrementer (the increment value, start and end values, offset (which byte to increment in the packet), and a skip n packets field can be parameterized);
- an IP checksum computer;
- an Ethernet checksum computer;
- a rate modifier (the rate value is parameterizable).

This model allows for very specific scenarios to be generated. Figure 5.1 shows an example of the generation of a stream that performs a port scan on two machines (which IP addresses are 192.168.0.3 and 192.168.0.1) using this model. In this stream, we need a skeleton whose base destination IP address is 192.168.0.3 (the address of the first machine on which want to execute the port scan). The base destination port number is set to 0. IP

and Ethernet checksum modifiers are included. In order to execute the port scan, we need two increment modifiers.

The first one manages the least significant byte of the destination port field (which is coded on two bytes).

- start value = 0;
- end value = 255;
- increment value = 1;
- offset = 40 (least significant byte of the destination port field);
- skip packets = 0;

The second one manages the most significant byte of the destination port field.

- start value = 0;
- end value = 255;
- increment value = 1;
- offset = 39 (most significant byte of the destination port field);
- skip packets = 256 (one increment every time the first increment's loop is done);

A third increment is also required to modify the IP address once the first machine has been scanned to set the IP address of the second machine. This increment is parameterized as follows:

- start value = 0;
- end value = -2;
- increment value = -2;
- offset = 33 (least significant byte of the destination IP address);
- skip packets = 65536 (one increment every time the second increment's loop is done);

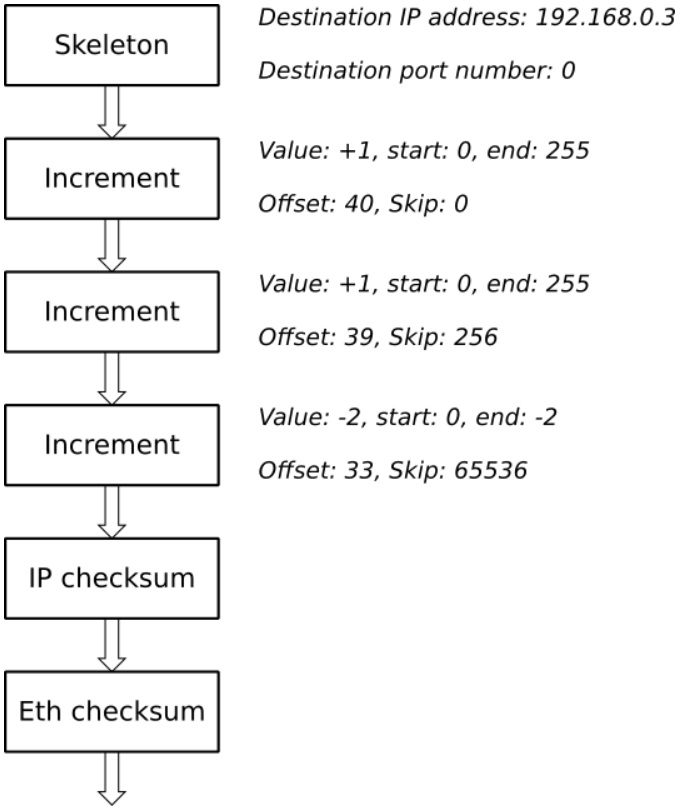


Figure 5.1 – Port scan stream generation example

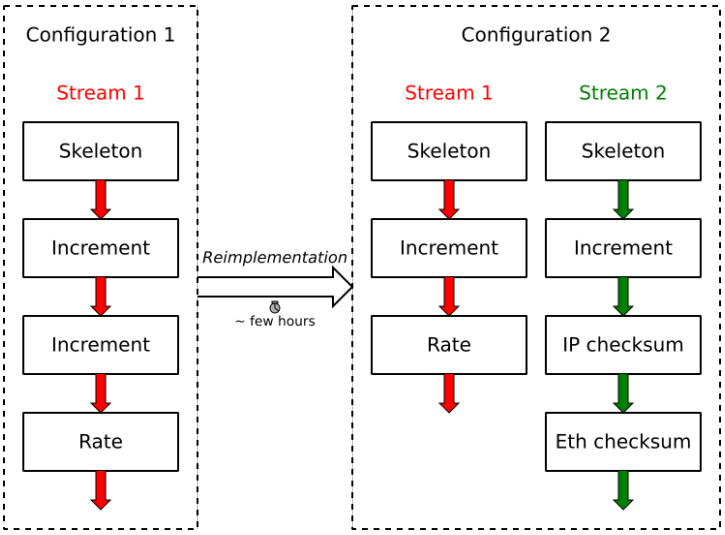
5.2 Design presentation

The original implementation of this model does include any mechanism that allows to modify the streams once it has been implemented, except for the parameterization of the modifiers. This means that every time any modification has to be made on the streams, may it be adding a new stream, changing a modifier or inserting a modifier in an existing stream, the whole design has to be reimplemented. This is very impractical, as waiting a few hours to be able to test a new scenario on a network monitoring applications can drastically increase the development and validation time of that application.

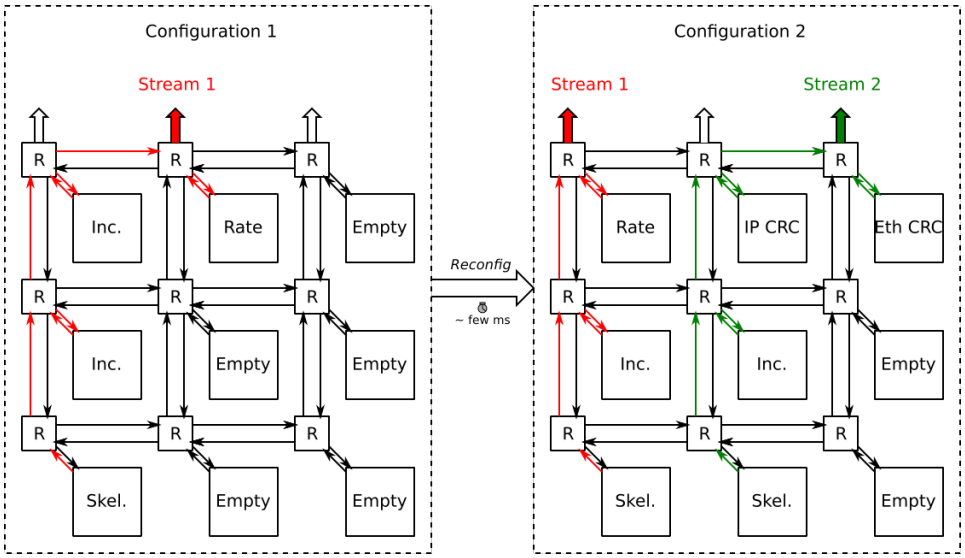
Using our architecture that uses both NoCs and bitstream relocation for this traffic generation model can greatly enhance the flexibility of the design. Indeed, using reconfigurable regions to host the modifiers would allow a designer to change modifiers on the fly. The Network-on-Chip architecture also improves the point-to-point connections in the original implementation, as inserting a modifier in a stream then does not require to change the interconnections, as it is already supported by the NoC. Only adding the router address to which the new modifier is connected in the packet header is now required, as well as configuring the new modifier in a previously empty reconfigurable region (this means that there must be regions that are still unused to add a modifier). This means that modifying the streams in the traffic generator would then only require a few reconfigurations and reparameterizations, which typically takes up to a few milliseconds.

Figure 5.2 shows an example of modifying the same streams using the original implementation and using our architecture. This figure only focuses on the modifications on the streams' elements, as such no parameterization is shown.

While using bitstream relocation instead of traditional reconfiguration is not crucial in this test case, as storing the bitstreams should not be an issue since this generator does not aims at being embedded, and no critical reconfiguration times have to be met, this is a good way to validate the mechanisms introduced in our design flow on a real test case, before implementing them on more complex designs. The fact that there is no constraint on the reconfiguration times also allows us to only focus on the mechanisms. For these reasons, we decided to use bitstream relocations instead of simple reconfigurations in this design.



(a) Using the original implementation



(b) Using our architecture

Figure 5.2 – Example of modifying the streams in the traffic generator using a) the original implementation and b) our architecture

5.3 Implementation

5.3.1 Overview

In order to implement our traffic generator on the NetFPGA SUME board [6], we need to integrate it with the NetFPGA design flow, and to make some adaptations because of our NoC architecture overlay. Figure 5.3 shows the global architecture of our generator.

As 4 output 10G optical links are provided on the board, we decided to use 4 columns of routers on our NoC. Each 10G optical output is connected to the local port of one router of the top row. More details about our outputs are provided in section 5.3.3. The number of rows will depend on synthesis and implementation results, that will indicate how many relocatable regions our design can afford.

As our NoC architecture requires the sequence of all modifiers a packet has to go through to be in the header of the packet, the skeleton sender provided in the original implementation is not suited for our architecture. Instead, we developed a new sender that also manages the parameterization of the modifiers. As the data width of our links on the NoC has been dimensionned to reach 10Gbps (which is the throughput of the 10G optical links), 4 senders are enough to saturate the 4 outputs. More details about our senders are provided in section 5.3.2.

5.3.2 Senders

The senders in our test case are responsible for transmitting both the base packets and the parameterizations to the modifiers. As parameterization does not require a high throughput, it is unnecessary to parallelize the sending of parameterization. As a result, only one sender is responsible for the parameterization of all modifiers. Each sender is connected to two data RAMs (and 2 parameter RAMs for the first sender). This is done so that a future traffic model can be preloaded onto the FPGA while the previous one is still generated.

When the first traffic model has to be generated, all data (and parameters) are sent to the first ram of each sender, preceded by the number of data to be sent. Once all data has been sent to the RAMs, a signal is sent to the first sender (the one that also handles the parameterization). Once that signal has been received, the sender releases the content of the parameter RAM on the NoC. A timer (which is for now set to 1000 clock cycles) is then used to let enough time for all modifiers to receive their parameter packets. At the end of that timer, the first sender sends a signal to all other senders,

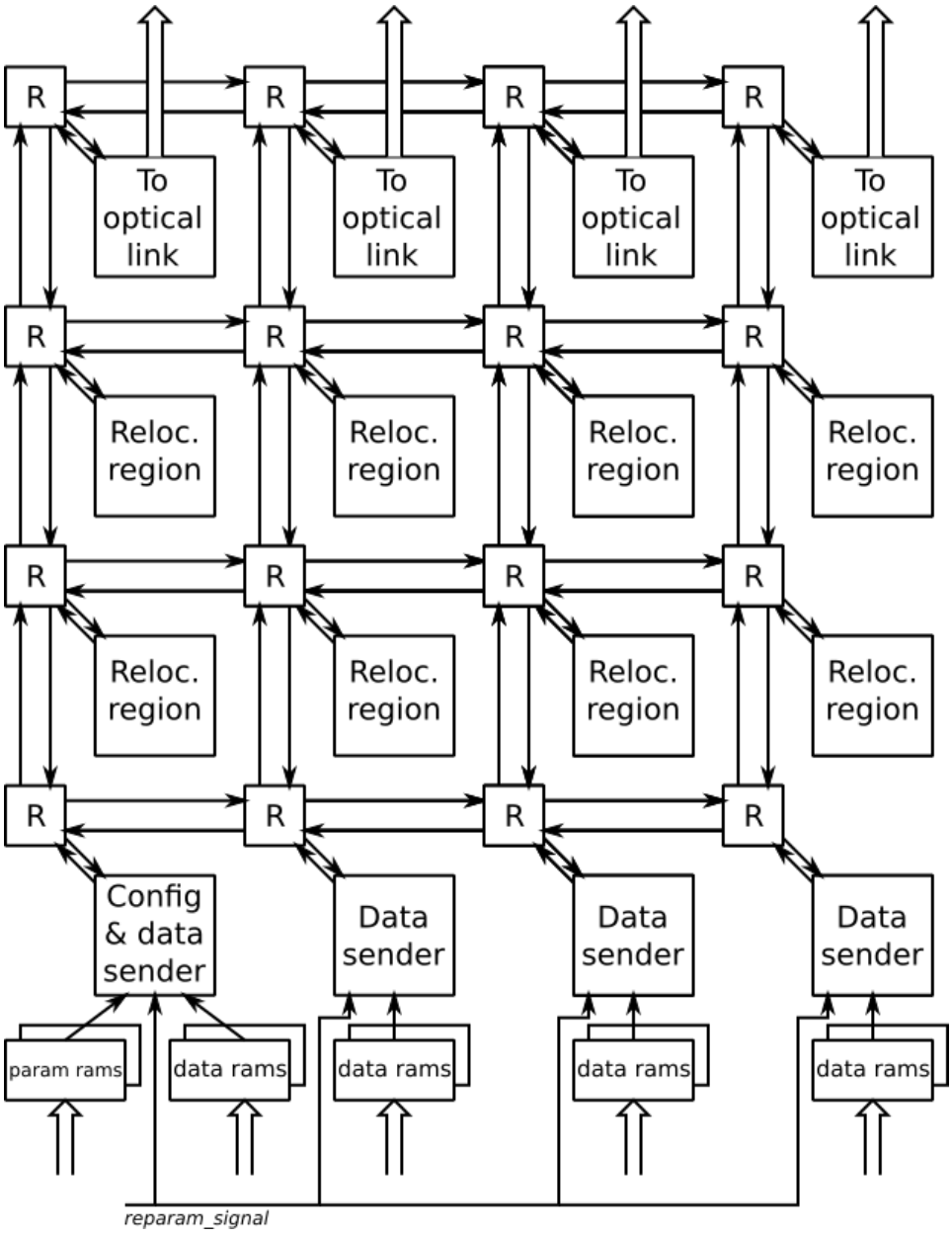


Figure 5.3 – Overview of our traffic generator

which let all senders start releasing the content of its ram to the NoC. When a sender reaches the end of its data RAM, it loops back to the beginning of the RAM. This enables the same behavior as the skeleton generator of the original implementation, which sends the same packet over and over.

When a new traffic model has to be generated, a signal is sent to all senders to make them stop releasing data once they reach the end of their data RAM content. Meanwhile, the new data and parameters can be sent to the RAMs that were previously free. Once all the remaining data have gone through the NoC, the modifiers can be reconfigured. In the current version of the generator, the reconfiguration still has to be done manually by the use, which means that waiting for all packets to be done going through the NoC should not be an issue. However, mechanisms that allows to automatically know when a modifier do not longer have data to treat (and thus can be reconfigured) should be implemented is the reconfiguration has to be automated. Once all modifiers have been configured according to the new model, a signal is sent to the senders, and the generation be start using the same steps as for the first generation.

5.3.3 Outputs

Once the packets have gone through all the functional units they had to, they are sent to one of the output *to_10G_optical_link* modules (which is indicate by the last router address in the packet header). The *to_10G_optical_link* are responsible for removing the headers of the packets so that the NoC information is not sent to the 10G optical links. It also has to resegment the packets according the third flit of the header, which contains the size of each sub-packet (see 4.2.4).

The optical links are managed using the *AXI 10G Ethernet IP* [1] provided by Xilinx, with a wrapper provided by the NetFPGA flow. The packets have to be sent to the wrapper using the AXI4-stream protocol, using a 256 bits data width. This means that besides removing the header, the output component also has to group flits 2 by 2 (unless the end of a sub-packet is reached) as our NoC uses a 128 bits data width.

5.4 Results

To the day of writing this version of this document, we still have not managed to make our bitstream relocation design flow work with designs that also use the AXI 10G Ethernet IP. This means that our architecture still has not been fully validated with the traffic generator. For this reason, a version

#Resource type	Used	Available	% Util.
LUTs	141186	433200	32.59
Registers	74708	866400	8.62
BRAM36k	340	1470	23.12
BRAM18k	44	2940	1.49

Table 5.1 – Resources used by the 5 by 4 traffic generator using increments only

using static functional units has been developed in order to validate the NoC part of our architecture. This implementation results of this version will be presented in section 5.4.1. Section 5.4.2 will present the issues that we encountered while using our bitstream relocation design flow in a design that uses the AXI 10G Ethernet IPs. Since we were not able to make our design flow work in this particular case, we decided to run tests on this design flow using simpler test cases. These tests will be presented in section 5.4.3.

5.4.1 Static version

The static version of our traffic generator has been developed in order to validate the NoC part of our architecture. This static version uses increment modules as all of the functional units. It also implements the reconfiguration mechanism which, while we don't use reconfiguration on this design, allows to reparameterize the modules on the generator.

The first test we ran was to identify the number of routers that we could afford to use on the Virtex7 690t (which is the FPGA included in the NetFPGA SUME board). Considering our constraints of a design clock at 156.25MHz, a data width of 128 bits, we obtained successful implementations for NoC sizes up to 5 by 4 with the Xilinx Vivado design suite. The resources utilization after placement and routing is provided in table 5.1. This design meets all timing requirements and output packets accordingly to the way the increment modules are parameterized. However, we only achieved successful implementation for NoC sizes up to 3 by 4 using the ISE design suite. This further emphasizes the fact that our design flow would greatly benefit from being compliant with the Vivado design suite if we want to use it for more complex designs, as the Place & Route algorithm from the ISE design suite has troubles meeting timing requirements that Vivado does not.

We also tested the maximum achievable throughputs of our NoC when only considering packets whose sizes are the minimal size of an IP packet (64 bytes). The results provided here are for a NoC which size is 5 by 4. For this test, 4 streams are generated, all going through each router that is located between the sender and the output module in the same router column. This means that each packet goes through 3 increment modules before being released on the 10G optical links. When only providing one 64 bytes packet in each NoC packets (*i.e.* without using the multipacket functionality of our NoC protocol overlay), we reach a throughput of only about 3 Gbps, which is expected considering that in this case the ratio $useful_data/NoC_packet_size$ is one third (4 useful data flits for 8 header flits). Using our multipackets functionality, providing 4 IP packets in each NoC packet is enough to obtain throughput similar to the ones of the original implementation (about 6 Gbps for minimum size packets). Using this functionality to its maximal potential (grouping up to 15 IP packets in each NoC packet) we obtained a throughput of 8.5Gbps, which is close to maximal theoretical throughput, which is equal to 8.82Gbps (the number of useful data flits in each packet is 15×4 while the header size is still 8 flits, which gives a ratio of $60/68 = 0.882$). This means that, besides improving the flexibility of the original model implementation, our NoC architecture can also increase its throughput in the worst case scenario.

Of course, these are only maximum throughputs that have been obtained using a favorable scenario (only one stream goes through each modifier). Using a modifier for several streams (*i.e.* having a router routing several streams) heavily decreases the throughput of the design, for example, having two streams going through one router would cut their throughputs by half. This means that placement of modifiers on the NoC can have a big impact on its overall performances.

5.4.2 Issues with relocation

While using our bitstream relocation design flow on our traffic generator, we encounter some issues that seem to come from the way the AXI Ethernet 10G IP from Xilinx is implemented. As stated in section 5.4.1, using the ISE design suite (upon which our bitstream relocation design flow is based) only allows for a 3 by 4 NoC to be implemented. As a result, only 4 relocatable regions will be used in the current relocation version of the generator.

Prohibiting clock regions for relocation

As stated in chapter 3, our bitstream relocation design flow does not support designs that use several different clocks. This is because our floorplanning algorithm does not know where the different clock domains will be located on the final design. As a result, it can place relocatable regions, which are in one clock domain, in clock regions that are allocated to another clock domain. However, our traffic generator has two distinct clock domains, one at 322.5MHz that is used by the interfaces with the 10G optical links, and one at 156.25MHz for the rest of the design. This means that we have to find a way to prevent our floorplanner from placing regions in clock regions used by the interfaces. Fortunately, the 322.5MHz clock domain only uses one clock region, which has been identified by using the implementation results of the static version. In order to be sure that our floorplanner does not use that clock region, we decided to declare that region as non reconfigurable in the layout description of the FPGA target. As our floorplanner can not place any region at places where non reconfigurable resources are present, we thus make sure that we can use both clock domains in our design.

However, the layout modification is dependant on that particular design, needs a recompilation of the layout libraries, and has to be done manually. This means that while it is possible to prevent some clock regions from hosting relocatable regions, this process has not been automated, and as such, is not natively supported by our design flow.

Placement conflicts with the Xilinx IP

When trying to implement our traffic generator, errors were occurring during the MAP process. These errors seem to indicate a conflict between the placement constraints of the interfaces and our relocatable regions. Indeed, the Xilinx IP uses *Relatively Placed Macros* (RPMs) [9]. RPMs are used by the Xilinx tools to define relative placement of elements, in order to ensure placements and timings of critical parts of the design. While trying to implement our design, the RPMs are unable to be respected, which led us to think that some of our relocatable regions placements were conflicting with the placement induced by these RPMs. As a result, we identified the locations of all the parts of the IP by using the implementation results of the static version (see figure 5.4). On this figure, all resources of the AXI Ethernet 10G IPs are highlighted in green.

As the IPs use the same RPMs in the static version and in the relocation one, we manually placed our relocatable regions in locations where no part

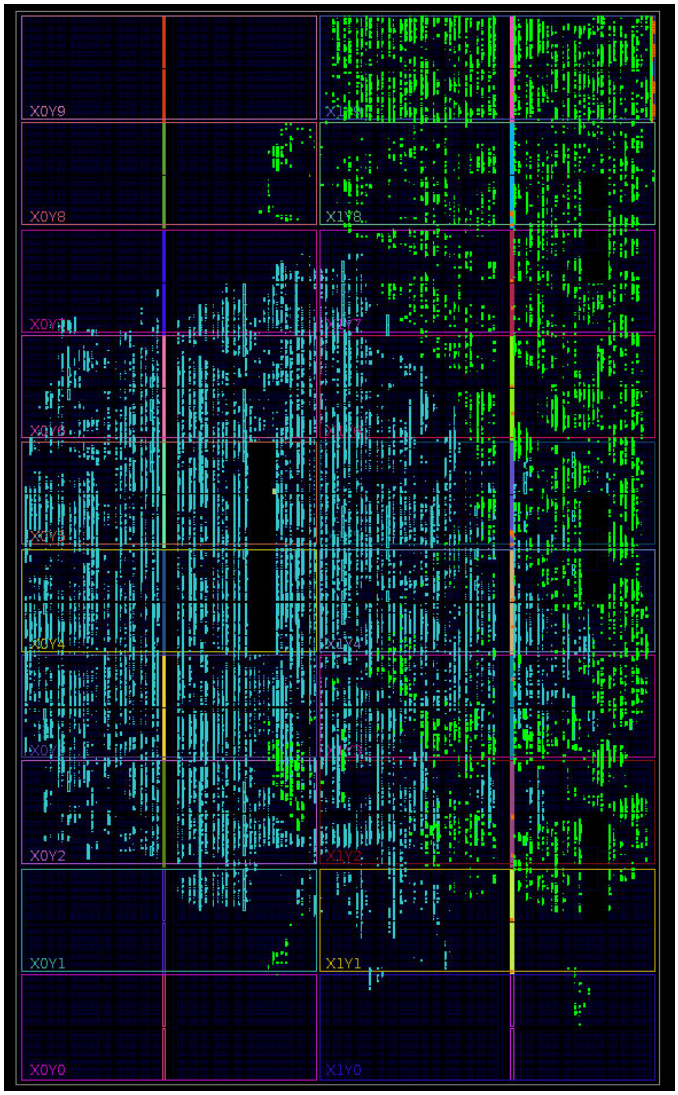


Figure 5.4 – Screenshot of the placement of the AXI Ethernet 10G IPs (highlighted in green)

IP	#Slices	#BRAMs	#DSPs	Max frequency (Mhz)
DFT_8	1048	4	8	542.505
DFT_16	1470	5	12	542.505
Cordic_r_8_8_8	272	0	0	775.964
Cordic_v_8_8_8	305	0	0	475.884
Uniform_Generator	129	0	0	1402.328

Table 5.2 – Resources requirements and maximum achievable frequency of the tested reconfigurable modules

of the IP are placed. The resulting floorplan is presented in figure 5.5. On this figure, our relocatable are the rectangles encircled in purple.

This led to the same RPM placement errors. We then tried to remove the PRIVATE constraint from our reconfigurable regions, as this could potentially have prevented some long wires to be connected in the interface, however, this did not solve the issue. This means that even reconfigurable designs that do not use relocation would not be successfully implemented when using that IP.

While no solution to this issue has been found yet, it is still under heavy investigation as it prevents us from validating the relocation version of our traffic generator.

5.4.3 Validation of relocation on other designs

As we did not manage to make our bitstream relocation design flow work with our traffic generation application, we ran tests on simpler designs in order to validate our approach. For these tests, we use the same FPGA as for the traffic generator (the Xilinx Virtex7 690T). On this design, we use 4 relocatable regions. For each tested module, we use its partial bitstream to configure it in the first region, and relocate it in the other 3. The tested modules are: two Spiral DFTs (8 and 16 bits) [10], two cordic operators from OpenCores [11] and a 128-bits uniform operator described in [58]. Their resource footprint, as well as the maximum frequency estimated by the synthesis tool are summarized in table 5.2. For each module tested, a static version is also implemented, and the same inputs are provided from a RAM to all relocatable and static modules. The output of all modules are then compared using XOR gates, and an output signal is activated once a single difference has been observed.

Using our design flow, all these modules have been successfully relocated

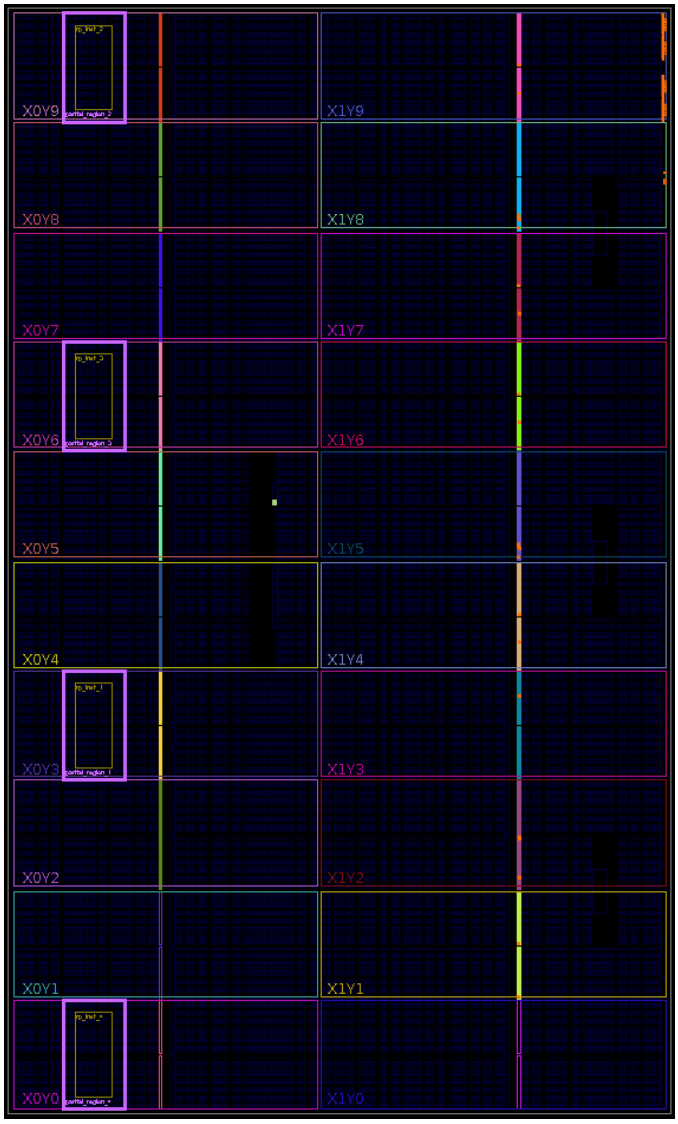


Figure 5.5 – Placement of our relocatable regions in regions not occupied by the AXI Ethernet 10G IPs

and validated using our test method. The clock frequency of the system is set to the board's 156.25MHz clock, however, timing analysis indicate that this design would be able to run successfully with clock frequencies up to 450MHz, which is really close to the maximum estimated frequency of our slowest module (the vectorial mode cordic (cordic_v_8_8_8)).

5.5 Conclusion

In this chapter we investigated the use of our architecture, that combines Network-on-Chips and bitstream relocation, in the case of a traffic generator. This generator is based on a traffic generation model that uses static base IP packets (skeletons) that go through a sequence of parameterizable modifiers to generate a precise test case. The reconfiguration part of our architecture allows to change the modifiers that are in that sequence on the fly, while the NoC natively supports the resequencing of the modifiers. While the initial implementation of this model lacked flexibility, our architecture would allows the generated model to be modified within a few milliseconds.

However, despite having validated a static version of this generator, the use of our bitstream relocation design flow with the AXI Ethernet IP from Xilinx, which is used by the 10G Ethernet optical interfaces of the board on which our generator is implemented, rose issues during the implementation process. These issues are still under heavy investigation.

Additional tests have thus been provided in order to validate our bitstream relocation design flow on simpler designs This means that while our traffic generator has not been yet successfully implemented, all mechanisms introduced in this thesis have been validated.

Chapter 6

Conclusion

In this thesis we investigated flexible architectures for high performance data processing. As up-to-date data processing applications require always more computation power, fitting all functionalities of a specific application into a single chip is becoming challenging. As a result, architectures that have to support high throughputs must also be flexible enough so that their resource footprints stay reasonable. Network monitoring applications are a perfect example of this kind of application, as they need to be able to sustain high data rates while their functionalities highly depend on the incoming traffic. Studying the use of FPGAs has been done as these platforms are ideal to address both the flexibility and throughput requirements of network monitoring applications. However, existing solutions were not able to sustain both these requirements. The architecture presented in this thesis is based on bitstream relocation, which highly increases the flexibility of FPGA designs, and Network-on-Chips, which are an interconnection structure that can provide flexibility while being able to sustain high data rates.

6.1 Automated design flow for bitstream relocation

Bitstream relocation is a technique that allows to use a single partial bitstream to configure a functional unit in several regions on an FPGA. This means that designs that use relocation instead of simple partial reconfiguration see their memory usage and generation times for partial bitstreams greatly reduced, particularly in designs that use both many reconfigurable modules and reconfigurable regions.

However, while experimenting with bitstream relocation, we realized that

using that technique is fairly challenging. Indeed, it requires a deep knowledge of the targeted FPGA, as it involves low level constraining of the relocatable regions. Identifying compatible regions for relocation and constraining them is also a long and error-prone process. For these reasons, bitstream relocation is still a very scarcely used technique. Automating it would then largely improve its accessibility. However, no automated design flow dedicated to bitstream relocation was available yet.

As a result, we developed a new fully automated design flow for bitstream relocation. This design flow is based on the Xilinx ISE Design Suite, methods already described in the state-of-the-art, and new algorithms for steps that had not been addressed yet. Indeed, in order to automate bitstream relocation, a floorplanner that is specifically tailored for it has been developed, as well as a technique to ensure that timings at the interfaces of all regions will be met when a relocation is performed. This design flow aims at being user friendly, as no prior knowledge of the specific target or relocation specific constraints is required from the designer. All the steps are scripted, which means that once all the inputs and parameters of a design have been provided, no intervention is needed for our design flow to generate all files required to run that design and perform the relocations. This makes any relocation specific step entirely transparent to the user. The time required by our design flow to perform its relocation specific tasks is also negligible compared to the time needed by the necessary steps of traditional FPGA design flow.

This design flow has been the subject of a publication in the *19th Euromicro Conference on Digital System Design (DSD 2016)*, entitled "Autoreloc: An Automated Design Flow for Bitstream Relocation on Xilinx FPGAs".

6.2 Network-on-Chips architectures for network monitoring

In order to support network monitoring applications, a communication architecture that can fulfill both the flexibility and throughput requirements of those applications is needed. However most common communication architectures are usually lacking in one of these requirements. Indeed, while offering high throughputs, point-to-point structures do not offer any flexibility regarding interconnections. On the other hand, bus architectures provide high levels of flexibility, however, as only one pair of modules can communicate at any given time.

We thus decided to investigate the use of Network-on-Chips as a commu-

nication architecture for network monitoring applications in order to provide flexibility while still being able to offer high data rates. Based on network monitoring requirements, the choice of a particular NoC has been made. The chosen Network-on-Chip is a 2D-mesh Hermes NoC generated using the Atlas generation tool, which uses a packet switching strategy and an XY routing algorithm. The data width has been dimensionned so that the throughput of each router can saturate a 10Gbps Ethernet link, which is used on the NetFPGA-SUME board available for this project.

A protocol overlay has been added to the Hermes NoC protocol so that the functional units on the NoC that will be subject to bitstream relocation can be as simple as possible. This overlay is able to provide information about the whole sequence a packet has to go through inside that packet. It is also able to add parameterization flits to bring small modifications to the functional units. Since both these additions can be detrimental to the achievable throughput on the NoC, as more bandwidth is then used to transmit the packet headers, the possibility to group up packets that have to go through the same functional units has been added. This results in smaller header over data ratio and thus reduces the throughput drop introduced by our overlay.

Finally, as bitstream relocation benefits from having relocatable units as simple as possible, a generic interface between routers and functional units has been developed. This interface handles all mechanisms introduced by our protocol overlay, as well as buffering packets so that a router can still transmit a new packet while its functional unit is still treating a previous one. This interface thus makes all these tasks entirely transparent to both the routers and the functional units.

6.3 Traffic generator test case

A test case was required in order to validate our architecture on a real case application. The choice of a traffic generator for this test case has been made, as implementing a flexible high throughput traffic generator is required to test and validate future applications that will use this architecture. It would also allow us to provide a proof of concept of the mechanisms we have introduced in this thesis without having to focus on more critical points of reconfigurable designs, especially reconfiguration times.

This traffic generator is based on an existing open-source project that generates traffic models based on a skeleton/modifiers template. It sends a skeleton, which is a base IP packet, to a sequence of modifiers in or-

der to generate test scenarios. However, the original implementation uses static modifiers with point-to-point connections, meaning that any time a modifier or an interconnection between modifiers must be changed, a whole reimplementation has to done, which can take up to a few hours. Using our architecture would then allow to make these types of changes in only a few milliseconds, as changing a modifier would only require a relocation, and changing an interconnection is natively supported by our NoC communication structure.

However, while implementing this test case, we encountered some issues related to using relocation in a design that uses the *AXI 10G Ethernet* IP from Xilinx, which is required to manage the 10G Ethernet optical links of the NetFPGA-SUME board we use for this project. This means that this test case has not been successfully implemented yet.

While these issues are still under investigation, we used simpler test cases in order to validate both our bitstream relocation design flow and our NoC architecture separately. Relocations have been performed on designs that do not use the troublesome IP from Xilinx. On the other hand, the NoC architecture has been validated on a static version of the traffic generator. This means that while we still have issues with the implementation of our test case, all mechanisms introduced in this thesis have been validated.

6.4 Perspectives and future work

This thesis presented a flexible architecture based on bitstream relocation and Network-on-Chip that is able to sustain multiple 10Gbps links. However, this architecture has still not been successfully implemented due to conflicts happening when using both bitstream relocation and the *AXI 10G Ethernet* IP used to handle the 10G optical links. In that regard, short term improvements must involve the solving of these conflicts in order to fully validate our architecture on our traffic generator test case.

Once the traffic generator is validated, this architecture will be able to be used to host various network monitoring applications. This means that more use cases could be validated, and providing a framework that can handle most network monitoring basic functionalities would be highly beneficial for the community. While our NoC architecture has been validated using 4 10G interfaces, it is likely that future applications will require more than 40Gbps throughputs. This means that increasing the parallelism in our architecture will be required to achieve those increased throughputs. Also, while this architecture has been dimensionned accordingly to network mon-

itoring requirements, we believe that combining NoCs and bitstream relocation can be used in other fields where flexible computing architectures that can sustain high data rates are required, prime example being *Software Defined Radio* or *Cloud-RAN*, as they both have strong requirements in terms of data rates and flexibility. Investigating the adaptation of our architecture to those fields is thus another possibility for mid-term improvement.

Finally, as our bitstream relocation is based on a design flow that is not supported anymore, working towards the compliancy with newer tools for FPGA designs (especially the Vivado design suite from Xilinx) should be part of the long-term perspectives of this work, as should be the compliancy with tools from other FPGA vendors.

Glossary

ASIC Application Specific Integrated Circuit, integrated circuit specifically optimized for one specific application. 9, 10

AXI Advanced eXtensible Interface, interconnect bus for ARM based systems. 59, 61, 93–96, 100, 104

BRAM Block Random Access Memory, basic memory element on FPGAs. 10, 18, 47, 55, 61, 74

DDoS Distributed Denial of Service, attack consisting of using multiple sources to saturate its victim, making it unable to respond. 17

DPR Dynamic Partial Reconfiguration, mechanism that allows to modify a portion of an FPGA during run-time. 10–12, 17, 36–38

DSP Digital Signal Processing, arithmetic processor. 10, 18, 47, 55, 61

EDA Electronic Design Automation. 19, 20, 22, 25

FAR Frame Address Register, starting position of a partial bitstream in the FPGA SRAM. 25

FIFO First In First Out, buffering mechanism where data is released in the same order it was received. 75, 76

FPGA Field Programmable Gate Array, integrated circuit that can be configured at a very low level, using configurable hardware resources. 3, 10–14, 17–26, 30, 34, 35, 38, 45, 47, 48, 50, 54, 55, 57, 58, 60–62, 72, 81, 85, 91, 94, 96, 98, 101, 102, 105

FSM Finite State Machine, behavioral description of an automaton using states and transitions. 19

GPP Global Purpose Processor, highly fexible, sequencement based processor. 9

ICAP Internal Configuration Access Port, FPGA reconfiguration port located on the fabric itself, allowing for self-reconfiguration. 44

IO Input Output. 19, 20, 23

IP Intellectual Property, hardware component provided by a third party. 7, 15, 61, 63, 93–96, 98–100, 104

IP Internet Protocol. 70, 72, 85–87, 95, 100, 103

LUT Look Up Table, configurable logic element able to implement any n-entry logic equation. 10, 18, 21, 40, 42, 45, 47, 58

NGD Native Generic Database, Xilinx specific file describing a design using elementary cells. 20

NIDS Network Intrusion Detection System. 17

NoC Network-on-Chip, communication infrastructure made of interconnected routers. 4, 6, 12–15, 17, 28, 30–34, 59–65, 67, 70, 72, 74, 75, 78, 81–83, 86, 89, 91, 93–95, 100, 103–105

NPU Network Processing Unit. 9

PaR Place and Route, process that places and routes the elements of a design on an FPGA. 20, 53

QoS Quality of Service, measurement of the overall performance of a service. 9

RAM Random Access Memory. 91, 93, 98

RPM Relatively Placed Macro, constraint allowing for placed elements based on the location of others on an FPGA. 96, 98

RTL Register Transfer Level, behavioral, register and cycle accurate description of an hardware design. 19, 21, 22

SFP Small Form-factor Pluggable. 61

SRAM Static Random Access Memory. 18, 20, 21, 23, 25

UCF User Constraint File, file provided by a designer to constrain signals and entities on FPGA designs. 20, 37, 40, 54

XST Xilinx Synthesis Tool. 38

Bibliography

[1] Xilinx AXI 10G Ethernet IP, https://www.xilinx.com/products/intellectual-property/do-di-10gemac.html.

[2] DDoS attacks number growth, http://www.silicon.fr/augmentation-de-140-des-attaques-ddos-a-plus-de-100-gbits-en-2016-168892.html.

[3] DDoS attacks volume growth, https://www.infosecurity-magazine.com/news/ddos-attack-volumes-spike/.

[4] Xilinx Synthesis Tool, https://www.xilinx.com/products/design-tools/xst.html.

[5] Xilinx AXI DMA documentation, https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[6] NetFPGA website, http://netfpga.org/site/systems/1netfpga-sume/details/.

[7] Atlas redmine, https://corfu.pucrs.br/redmine/projects/atlas.

[8] T. Groléat. Open-source hardware traffic generator. https://github.com/tristan-TB/hardware-traffic-generator, 2013.

[9] Xilinx RPM documentation, http://application-notes.digchip.com/077/77-43452.pdf.

[10] Spiral website, http://www.spiral.net/hardware/dftgen.html.

[11] https://opencores.org/project,cf_cordic.

[12] A. B. Ahmed, A. B. Abdallah, and K. Kuroda. Architecture and design of efficient 3d network-on-chip (3d noc) for custom multicore soc. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 67–73, Nov 2010.

[13] R. Backasch, G. Hempel, S. Werner, S. Groppe, and T. Pionteck. Identifying homogenous reconfigurable regions in heterogeneous fpgas for module relocation. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pages 1–6, Dec 2014.

[14] T. Becker, M. Koester, and W. Luk. Automated placement of reconfigurable regions for relocatable modules. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3341–3344, May 2010.

[15] T. Becker, W. Luk, and P.Y.K. Cheung. Enhancing relocatability of partial bitstreams for run-time reconfiguration. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 35–44, April 2007.

[16] C. Beckhoff, D. Koch, and J. Torresen. Go ahead: A partial reconfiguration framework. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 37–44, April 2012.

[17] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, Jan 2002.

[18] C. Bolchini, A. Miele, and C. Sandionigi. Automated resource-aware floorplanning of reconfigurable areas in partially-reconfigurable fpga systems. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 532–538, Sept 2011.

[19] A. Botta, A. Dainotti, and A. Pescapé. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, 56(15):3531 – 3547, 2012.

[20] S.D. Chawade, M.A. Gaikwad, and R.M. Patrikar. Review of xy routing algorithm for network-on-chip architecture. *International Journal of Computer Applications*, 43(21):975–8887, 2012.

[21] K. Compton, J. Cooley, S. Knol, and S. Hauck. Configuration relocation and defragmentation for fpgas. Technical report, Citeseer, 2000.

[22] N. Dahir, R. Al-Dujaily, A. Yakovlev, P. Missailidis, and T. Mak. Deadlock-free and plane-balanced adaptive routing for 3d networks-on-chip. In *Proceedings of the Fifth International Workshop on Network on Chip Architectures*, NoCArc '12, pages 31–36, New York, NY, USA, 2012. ACM.

[23] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 684–689, 2001.

[24] A. Das, D. Nguyen, J. Zambreno, G. Memik, and A. Choudhary. An fpga-based network intrusion detection architecture. *Information Forensics and Security, IEEE Transactions on*, 3(1):118–132, March 2008.

[25] T. Drahonovsky, M. Rozkovec, and O. Novak. Relocation of reconfigurable modules on xilinx fpga. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2013 IEEE 16th International Symposium on*, pages 175–180, April 2013.

[26] T. Drahonovsky, M. Rozkovec, and O. Novak. A highly flexible reconfigurable system on a xilinx fpga. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pages 1–6, Dec 2014.

[27] J. Fong, Xiang Wang, Yaxuan Qi, Jun Li, and Weirong Jiang. Parasplit: A scalable architecture on fpga for terabit packet classification. In *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*, pages 1–8, Aug 2012.

[28] F. Fusco, F. Huici, L. Deri, S. Niccolini, and T. Ewald. Enabling high-speed and extensible real-time communications monitoring. In *2009 IFIP/IEEE International Symposium on Integrated Network Management*, pages 343–350, June 2009.

[29] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *IEEE Design Test of Computers*, 22(5):414–421, Sept 2005.

[30] T. Groléat. *High performance traffic monitoring for network security and management*. PhD thesis, Telecom Bretagne, 2014.

[31] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '00, pages 250–256, New York, NY, USA, 2000. ACM.

[32] L. Guo, W. Hou, and P. Guo. Designs of 3d mesh and torus optical network-on-chips: Topology, optical router and routing module. *China Communications*, 14(5):17–29, May 2017.

[33] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist. Network on chip: An architecture for billion transistor era. In *Proceeding of the IEEE NorChip Conference*, volume 31, page 11, 2000.

[34] X. Jiang and T. Watanabe. A novel fully adaptive fault-tolerant routing algorithm for 3d network-on-chip. In *2013 IEEE International Conference of IEEE Region 10 (TENCON 2013)*, pages 1–4, Oct 2013.

[35] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert. Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 151b–151b, April 2005.

[36] H. Kalte and M. Porrmann. Replica2pro: Task relocation by bitstream manipulation in virtex-ii/pro fpgas. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 403–412, New York, NY, USA, 2006. ACM.

[37] Y. B. Kim and Y. B. Kim. Fault tolerant source routing for network-on-chip. In *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*, pages 12–20, Sept 2007.

[38] S. Kumar, A. Jantsch, J. P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*, pages 105–112, 2002.

[39] j. Li, Y. Chen, C. Ho, and Z. Lu. Binary-tree-based high speed packet classification system on fpga. In *Information Networking (ICOIN), 2013 International Conference on*, pages 517–522, Jan 2013.

[40] M. Li, Q. Zeng, and W. Jone. Dyxy: A proximity congestion-aware deadlock-free dynamic routing method for network on chip. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 849–852, New York, NY, USA, 2006. ACM.

[41] Z. Li, K. Compton, and S. Hauck. Configuration caching management techniques for reconfigurable computing. In *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871)*, pages 22–36, 2000.

[42] I. Loi, S. Mitra, T.H. Lee, S. Fujita, and L. Benini. A low-overhead fault tolerance scheme for tsv-based 3d network on chip links. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '08, pages 598–602, Piscataway, NJ, USA, 2008. IEEE Press.

[43] A. Mello, N. Calazans, and F. Moraes. Atlas-an environment for noc generation and evaluation.

[44] A. Mello, L. Ost, F. Moraes, and N. Calazans. Evaluation of routing algorithms on mesh based nocs. Technical report.

[45] Alessio Montone, Marco D. Santambrogio, Donatella Sciuto, and Seda Ogrenci Memik. Placement and floorplanning in dynamically reconfigurable fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 3(4):24:1–24:34, November 2010.

[46] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69 – 93, 2004.

[47] T.D.A. Nguyen and A. Kumar. Prfloor: An automatic floorplanner for partially reconfigurable fpga systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 149–158, New York, NY, USA, 2016. ACM.

[48] P. P. Pande, C. Grecu, A. Ivanov, and R. Saleh. Design of a switch for network on chip applications. In *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, volume 5, pages V–217–V–220 vol.5, May 2003.

[49] P.P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Transactions on Computers*, 54(8):1025–1040, Aug 2005.

[50] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[51] M. Sanlı, E. Güran Schmidt, and H. Cengiz Güran. Fpgen: A fast, scalable and programmable traffic generator for the performance evaluation of high-speed computer networks. *Performance Evaluation*, 68(12):1276 – 1290, 2011.

[52] T. Schonwald, J. Zimmermann, O. Bringmann, and W. Rosenstiel. Fully adaptive fault-tolerant routing algorithm for network-on-chip architectures. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 527–534, Aug 2007.

[53] L. Singhal and E. Bozorgzadeh. Multi-layer floorplanning on a sequence of reconfigurable designs. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–8, Aug 2006.

[54] L. Singhal and E. Bozorgzadeh. Physically-aware exploitation of component reuse in a partially reconfigurable architecture. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006.

[55] A.M. Smith, G.A. Constantinides, and P.Y.K. Cheung. Integrated floorplanning, module-selection, and architecture generation for reconfigurable devices. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(6):733–744, June 2008.

[56] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French. Torc: Towards an open-source tool flow. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 41–44, New York, NY, USA, 2011. ACM.

[57] S. Stoev, G. Michailidis, and J. Vaughan. On global modeling of backbone network traffic. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5, March 2010.

[58] David B Thomas and Wayne Luk. High quality uniform random number generation using lut optimised state-transition matrices. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 47(1):77–92, 2007.

[59] K. Vipin and S.A. Fahmy. Efficient region allocation for adaptive partial reconfiguration. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–6, Dec 2011.

[60] K. Vipin and S.A. Fahmy. Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. In OliverC.S. Choy, RayC.C. Cheung, Peter Athanas, and Kentaro Sano, editors, *Reconfigurable*

Computing: Architectures, Tools and Applications, volume 7199 of *Lecture Notes in Computer Science*, pages 13–25. Springer Berlin Heidelberg, 2012.

[61] K.V. Vishwanath and A. Vahdat. Swing: Realistic and responsive network traffic generation. *Networking, IEEE/ACM Transactions on*, 17(3):712–725, June 2009.

[62] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107, Apr 1995.

[63] L. Yuan, C. N. Chuah, and P. Mohapatra. Progme: Towards programmable network measurement. *IEEE/ACM Transactions on Networking*, 19(1):115–128, Feb 2011.

[64] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41, Sept 2014.

Résumé

L'augmentation de la taille des réseaux actuels ainsi que de la diversité des applications qui les utilisent font que les architectures de calcul traditionnelles deviennent limitées. En effet, les architectures purement logicielles ne permettent pas de tenir les débits en jeu, tandis que celles purement matérielles n'offrent pas assez de flexibilité pour répondre à la diversité des applications.

Ainsi, l'utilisation de solutions de type matériel programmable, en particulier les *Field Programmable Gate Arrays* (FPGAs), a été envisagée. En effet, ces architectures sont souvent considérées comme un bon compromis entre performances et flexibilité, notamment grâce à la technique de *Reconfiguration Dynamique Partielle* (RDP), qui permet de modifier le comportement d'une partie du circuit pendant l'exécution.

Cependant, cette technique peut présenter des inconvénients lorsqu'elle est utilisée de manière intensive, en particulier au niveau du stockage des fichiers de configuration, appelés *bitstreams*. Pour palier ce problème, il est possible d'utiliser la relocation de bitstreams, permettant de réduire le nombre de fichiers de configuration. Cependant cette technique est fastidieuse et exige des connaissances pointues dans les FPGAs. Un flot de conception entièrement automatisé a donc été développé dans le but de simplifier son utilisation.

Pour permettre une flexibilité sur l'enchaînement des traitements effectués, une architecture de communication flexible supportant des hauts débits est également nécessaire. Ainsi, l'étude de *Network-on-Chips* dédiés aux circuits reconfigurables et au traitements réseaux à haut débit.

Enfin, un cas d'étude a été mené pour valider notre approche.

Mots-clés : FPGAs, Reconfiguration Dynamique Partielle, Calcul reconfigurable, Relocation de bitstreams, Traitement de données haut-débit, Réseaux sur puce

Abstract

The increase in both size and diversity of applications regarding modern networks is making traditional computing architectures limited. Indeed, purely software architectures can not sustain typical throughputs, while purely hardware ones severely lack the flexibility needed to adapt to the diversity of applications.

Thus, the investigation of programmable hardware, such as *Field Programmable Gate Arrays* (FPGAs), has been done. These architectures are indeed usually considered as a good tradeoff between performance and flexibility, mainly thanks to the *Dynamic Partial Reconfiguration* (DPR), which allows to reconfigure a part of the design during run-time.

However, this technique can have several drawbacks, especially regarding the storing of the configuration files, called *bitstreams*. To solve this issue, bitstream relocation can be deployed, which allows to decrease the number of configuration files required. However, this technique is long, error-prone, and requires specific knowledge in FPGAs. A fully automated design flow has been developped to ease the use of this technique.

In order to provide flexibility regarding the sequence of treatments to be done on our architecture, a flexible and high-throughput communication structure is required. Thus, a *Network-on-Chips* study and characterization has been done accordingly to network processing and bitstream relocation properties.

Finally, a case study has been developed in order to validate our approach.

Keywords: FPGAs, Dynamic Partial Reconfiguration, Reconfiguable computing, Bitstream relocation, High-speed data processing, Network-on-Chips

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

N° d'ordre : 2017IMTA0054

IMT Atlantique Bretagne-Pays de la Loire - www.imt-atlantique.fr

Campus de Brest
Technopôle Brest-Iroise
CS 83818
29238 Brest Cedex 03
T +33 (0)2 29 00 11 11
F +33 (0)2 29 00 10 00

Campus de Nantes
4, rue Alfred Kastler - La Chantrerie
CS 20722
44307 Nantes Cedex 3
T +33 (0)2 51 85 81 00
F +33 (0)2 51 85 81 99

Campus de Rennes
2, rue de la Châtaigneraie
CS 17607
35576 Cesson Sévigné Cedex
T +33 (0)2 99 12 70 00
F +33 (0)2 99 12 70 08