Du prototypage à l'exploitation d'overlays FPGA

Théotime Bollengier

UNIVERSITE
BRETAGNE
LOIRE

THÈSE DE DOCTORAT / ENSTA BRETAGNE
sous le sceau de l'Université Bretagne Loire

pour obtenir le titre de
DOCTEUR DE L'ENSTA BRETAGNE
Mention : Informatique

École Doctorale 601, Mathématiques et Sciences et Technologies de l'Information et de la Comunication (MathSTIC)

présentée par

Théotime BOLLENGIER

préparée à
l'Institut de Recherche Technologique b<>com
et Lab-STICC UMR CNRS 6285,
École Nationale Supérieure de Techniques Avancées de Bretagne

Du Prototypage à l'Exploitation d'Overlays FPGA

Thèse soutenue le 15 janvier 2018
devant le jury composé de:

Bertrand Granado
Professeur, UPMC / Rapporteur et président du jury

Christophe Jego
Professeur, ENSEIRB-MATMECA / Rapporteur

Virginie Fresse
Maître de conférences, Université J. Monnet / Examinatrice

Olivier Muller
Maître de conférences, TIMA / Examinateur

Loïc Lagadec
Professeur de l'ENSTA Bretagne / Directeur

Jean-Christophe Le Lann
Maître de conférences de l'ENSTA Bretagne / Encadrant

Erwan Fabiani
Maître de conférences, UBO / Invité

b<>com

ENSTA Bretagne

Remerciements

En premier lieu, je tiens à remercier Bertrand Guilbaud et par lui l'IRT b<>com, pour m'avoir permis de réaliser cette thèse. Je veux aussi remercier l'IRT b<>com pour m'avoir offert des conditions de travail très appréciables durant ma thèse (en tout cas pour un doctorant) : grand bureaux, grands écrans, et matériel informatique très performant.

Je tiens à remercier les collègues bcommiens de Brest pour l'ambiance de travail agréable sur le site du Technopôle. Je veux aussi remercier les collègues de Rennes pour leur accueil lorsque nous venions dans les grands bâtiments de la maison mère. Un grand merci à l'équipe Cloud pour m'avoir permis d'utiliser les serveurs de test déployés à Brest pour réaliser des milliers de synthèses FPGAs, ce qui a permis de réduire à quelques jours des expériences qui auraient pu prendre plusieurs mois.

Je tiens à remercier tout particulièrement mon directeur de thèse, Loïc Lagadec, pour son soutien, sa patience, son aide et ses conseils qui m'ont permis de finaliser cette thèse avec succès. Merci également à mon co-encadrent, Jean-Christophe Le Lann, pour son aide, son enthousiasme communicatif pour toutes les bidouilles matérielles et informatiques, et pour m'avoir fait découvrir le Ruby.

Je veux aussi remercier les collègues de l'ENSTA, pour ces discutions qui des fois éclairent, donnent de nouvelles idées, et parfois embrouillent. Un merci particulier à Mohamad Najem avec qui j'ai eu le plaisir de travailler pendant ma thèse, car c'est en effet un plaisir de travailler avec lui.

Je voudrais aussi remercier tous les membres du jury pour avoir accepté de lire et rapporter ce manuscrit ainsi que d'assister à la présentation de ce travail.

Finalement, je voudrais remercier mes parents qui (outre le fais de m'avoir permis de réaliser confortablement mes études), m'ont fait naître quelques années après l'apparition du premier FPGA, à une époque où l'électronique et l'informatique sont accessibles à tous pour la bidouille.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Aujourd'hui, en 2017, la société est résolument numérique : des domaines traditionnels comme le militaire ou le médical restent consommateurs de calculs, mais plus généralement, le grand public est désormais friand de ressources de calculs de plus en plus conséquentes. Par exemple, avec la popularité croissante des services web et l'arrivée de l'Internet des objets [2], une masse de données chaque jour de plus en plus importante doit être traitée. Ces traitements ne sont pas réalisés sur des ordinateurs personnels, mais dans le Cloud [3], c'est-à-dire dans des datacenters. Ces datacenters sont des sites physiques regroupant des ressources informatiques de calcul et de stockage des données.

À notre sens, dans le contexte d'un datacenter, les ressources de calcul utilisées doivent répondre à certaines exigences. Au niveau de l'exploitation du datacenter :

- il faut être capable de supporter l'hétérogénéité des ressources matérielles, car du fait de la mise à jour graduelle de l'infrastructure, différentes générations de ressources peuvent être utilisées simultanément ;
- chaque ressource doit permettre une exploitation dynamique, de façon à pouvoir optimiser l'exploitation globale du datacenter.

De plus, les concepteurs d'applications pour le Cloud sont majoritairement issus du monde logiciel. Ainsi, pour ne pas restreindre le nombre de clients pouvant développer des applications et pour rendre "mainstream" la programmation des ressources de calcul :

- les outils de compilation doivent être rapides, augmentant le nombre de cycles débogage-édition-implémentation que peut réaliser le concepteur d'applications ;
- ces outils doivent être simples à utiliser et ne pas demander une expertise particulière ;
- les ressources de calcul doivent permettre le "design reuse", et notamment de mutualiser les efforts de développement dans des bibliothèques.

Dans les datacenters, les ressources de calcul sont majoritairement des serveurs informatiques, dont l'unité centrale de traitement est un processeur. Les processeurs répondent aux exigences citées ci-dessus. En effet, ils sont flexibles, re-

programmables dynamiquement, et sont généralistes au niveau des classes d'algorithmes qu'ils peuvent réaliser. La compilation sur processeur est un problème bien maitrisé et les outils de compilation sont matures. La compilation d'un programme varie de quelques secondes à quelques minutes, et de nombreuses bibliothèques logicielles facilitent le développement de nouvelles applications. Toutefois, l'exécution sur processeur est séquentielle, c'est-à-dire que les instructions machine sont exécutées les une à la suite des autres, ce qui limite la puissance de calcul.

Accélérateurs matériels

Une solution pour pallier l'exécution séquentielle des applications est d'utiliser des accélérateurs matériels, qui sont des ressources de calcul spécialisées pour les traitements qu'ils ciblent. Les GPUs (Graphic Processing Units) [4] en sont un exemple, dont le but original est d'accélérer les traitements graphiques. Des efforts ont été réalisés afin de les rendre plus généralistes (notamment via l'utilisation de langages tels qu'OpenCL [5]), mais aussi pour les rendre exploitables dans une infrastructure Cloud [6]. Cependant, les GPUs ne sont pas adaptés à tous les types de traitements, car leur architecture est spécialisée pour les calculs réguliers (c'est-à-dire présentant peu de contrôles tels que des branchements imbriqués) et massivement parallèles.

Les accélérateurs ASICs (Application Specific Integrated circuit) [7] sont des circuits logiques spécialisés dans le traitement de l'application pour lequel ils ont été conçus. Les décodeurs vidéos matériels sont un exemple d'ASIC. Le circuit matériel étant dédié à un traitement donné, les ASICs présentent les meilleures performances en termes de temps de traitement et de consommation d'énergie. En revanche, leur fonction est figée dans le silicium, ils ne sont pas reprogrammables, donc développer une nouvelle application demande de développer et produire un nouvel ASIC. Or, bien qu'une fois développé, un ASIC peut être produit à bas coût, le développement d'un nouvel ASIC présente un coût non récurent (NRC) extrêmement élevé. Ainsi, bien que généralistes et performants, les ASICs ne sont pas une solution pour l'accélération d'applications clientes dans le Cloud.

Les FPGAs (Field-Programmable Gate Array) [8] sont des circuits logiques reconfigurables. La granularité de la configuration des FPGAs est au niveau de la porte logique, ce qui exhibe un espace de conception extrêmement large et permet de concevoir des circuits matériels (et donc performants) dédiés à n'importe quel type de traitements. Par leur capacité de reconfiguration, les FPGAs se présentent comme un compromis entre la flexibilité des processeurs et la performance des ASICs. Notamment, la conception d'un design sur FPGA présente un coût non récurent bien inférieur à celui des ASICs. Les FPGAs sont donc des accélérateurs intéressant pour accélérer des applications clientes dans le Cloud.

Les FPGAs

Deux des raisons d'être des FPGAs sont le prototypage d'ASIC et le remplacement d'ASICs dans des produits vendus en trop faible quantité pour justifier le coût de production d'un ASIC. De ce fait, les FPGAs exhibent un modèle d'exécution très bas niveau, équivalent (la lookup table) au modèle d'exécution des ASICs (la cellule standard). Il s'ensuit que le développement d'applications sur FPGA suit les premières étapes du flot de développement sur ASIC. En particulier, le développement sur FPGA demande au concepteur des compétences et une expertise en micro-électronique, ainsi qu'une expertise propre aux outils de développement FPGAs, qui sont complexes. La finesse de la granularité du modèle d'exécution des FPGAs implique aussi que le temps d'exécution des outils de synthèse FPGA sont longs par rapport à une compilation logicielle, allant de quelques minutes à plusieurs heures. Ces deux points vont à l'encontre des exigences présentées en début de cette introduction, à savoir la rapidité et la simplicité des outils de programmation. Cependant, les constructeurs FPGA commencent à proposer dans leurs chaînes logicielles des outils de synthèse haut niveau (HLS) [9] qui permettent, entre autres, de réduire l'expertise électronique demandée au concepteur. Néanmoins, obtenir un circuit applicatif performant via la HLS demande tout de même une compréhension fine des processus de synthèse de l'outil, et donc des compétences propres.

Par ailleurs, dans le contexte d'un datacenter où plusieurs générations de matériel sont exploitées simultanément, l'hétérogénéité des FPGAs complique leur exploitation. En effet, les FPGAs ne sont pas compatibles entre eux : un binaire de configuration réalisé pour un FPGA ne peut pas être utilisé pour configurer un autre modèle de FPGA. La situation logicielle équivalente est moins problématique, car la majorité des processeurs utilisés dans le Cloud (de marque Intel et AMD) partage le même jeux d'instruction (x86), et les binaires logiciels peuvent être exécutés de manière transparente sur des processeurs Intel ou AMD. Cependant, dans le cas des FPGAs, il est nécessaire de synthétiser l'application à nouveau pour pouvoir l'exécuter sur un nouvel FPGA. L'incompatibilité binaire entre les FPGAs rend donc plus compliquée la gestion d'un ensemble hétérogène de FPGAs.

De plus, les FPGAs se prêtent mal au "design reuse", et donc à la productivité des développeurs applicatifs. En effet, même les sources applicatives ne sont pas nécessairement portables d'un modèle de FPGA à un autre : l'utilisation dans un design d'IPs spécifiques à un FPGA donné peut demander de reconcevoir entièrement le design pour pouvoir le porter sur un autre FPGA.

Finalement, les FPGAs ne sont pas conçus pour être multi-utilisateurs. Or, le partage dynamique des ressources de calcul entre différentes applications est nécessaire pour l'exploitation optimale des ressources dans le contexte d'un datacenter. Même si un FPGA peut être partagé spatialement entre différentes applications, cela demande au concepteur de synthétiser ensemble les différentes applications, et donc de préparer ce partage avant exécution. Il est tout de même possible

d'utiliser la capacité de reconfiguration partielle dynamique (DPR) [10] offerte par certains FPGAs pour reconfigurer certaines zones d'un FPGA indépendamment les unes des autres, ce qui permet de partager un FPGA spatialement et dans le temps entre différentes applications indépendantes. Cependant, la mise en place de telles zones dynamiquement reconfigurables ainsi que la synthèse d'applications ciblant ces zones sont lourdes à mettre en place, et demandent une expertise additionnelle propre à la DPR.

Ainsi, de par leur flexibilité et leurs performances, les FPGAs sont des ressources intéressantes pour le Cloud. Les FPGAs font d'ailleurs leur apparition en tant qu'accélérateurs de calcul reconfigurables par les clients dans Amazon Web Services [11] ; Microsoft utilise des FPGAs pour accélérer son moteur de recherche Bing [12] ; et l'achat d'Altera par Intel souligne l'importance des FPGAs dans les datacenters comme ressource de calcul [13]. Cependant, comme nous venons de le voir, les FPGAs ne répondent pas encore aux exigences qu'il faut à notre sens réaliser pour démocratiser leur adoption dans le Cloud et les rendre "mainstream". La solution que nous proposons pour utiliser les FPGAs comme accélérateurs reconfigurables dans le Cloud est d'utiliser des *overlays*.

Les overlays

Dans ce travail, nous défendons l'idée que le Cloud peut profiter avantageusement d'un type de circuit particulier appelé *overlay*. Le concept d'overlay est relativement récent, et il n'a pas encore de définition précise et arrêtée. La définition la plus générique d'un overlay que l'on puisse donner est la suivante : *Un overlay est un design reconfigurable implémenté sur FPGA (qui est lui-même reconfigurable)*. En d'autres termes, un overlay est une couche d'abstraction matérielle placée entre l'application le FPGA hôte, isolant celle-ci de ce dernier.

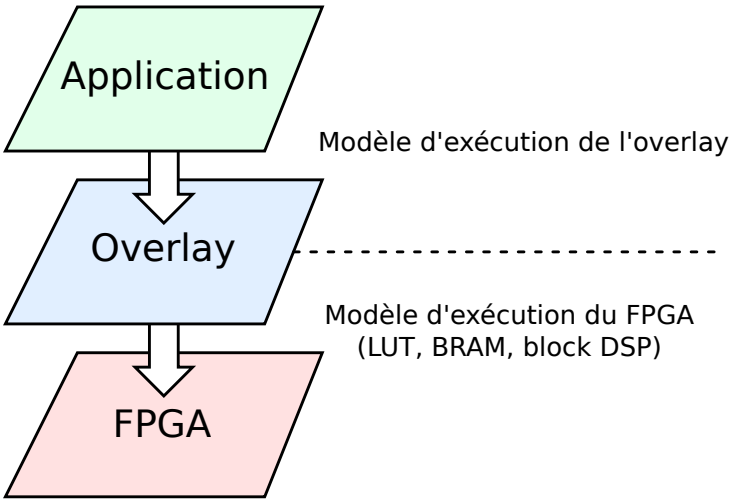


FIGURE 1.1 – L'overlay isole l'application du FPGA sous-jacent. Il apporte son propre modèle d'exécution.

Une analogie logicielle peut être faite avec la machine virtuelle Java : le processeur hôte utilise ses instructions pour implémenter les instructions de la machine virtuelle Java. L'application est compilée pour la machine virtuelle (en instructions Java), tandis que la machine virtuelle est compilée pour le processeur hôte (en instructions du processeur). Grâce à cette machine virtuelle, le processeur hôte est ainsi capable d'exécuter n'importe quel bytecode Java.

Une autre analogie, matérielle cette fois, peut être faite avec un processeur softcore (c'est-à-dire un processeur synthétisé sur FPGA) : le processeur softcore est un design programmable synthétisé sur FPGA, qui permet au FPGA d'exécuter les programmes ciblant le processeur. Cependant, à notre sens, le concept d'overlay vient avec une certaine notion de régularité, c'est-à-dire qu'une architecture overlay se présente sous forme d'une matrice d'éléments similaires. Ainsi, nous considérons un réseau de processeurs softcores comme étant un overlay, tandis qu'un seul processeur softcore ne sera pas considéré comme un overlay.

Comme illustré figure 1.1, l'overlay apporte son propre modèle d'exécution, qui peut être radicalement différent de celui du FPGA sous-jacent. L'application ne voit que le modèle d'exécution de l'overlay et est isolée du FPGA hôte. Le modèle d'exécution de l'overlay détermine sa granularité : on parle d'overlays grain fin lorsqu'il s'agit d'une granularité au niveau bit, tandis qu'on parle d'overlays gros grain lorsque la granularité est plus large, pour des opérateurs arithmétiques, ou encore un réseau de processeurs. Un overlay étant un design synthétisé sur FPGA, l'espace de conception des architectures overlay et de leurs modèles d'exécution est extrêmement large. Par l'étendue de cet espace, les overlays permettent d'explorer l'espace entre la facilité d'usage des processeurs et les performances des FPGAs. Les overlays sont donc de bons candidats pour être utilisés en tant qu'accélérateurs de calcul reconfigurables dans un contexte Cloud, tout en répondant aux exigences posées au début de cette introduction.

Proposition

Ce travail s'adresse aux opérateurs de datacenters, qui voudraient utiliser des overlays pour exploiter des FPGAs afin d'offrir des ressources matérielles reconfigurables à leurs clients. Cette thèse a été effectuée au sein de l'institut de recherche technologique b<>com, qui est issu d'un partenariat public/privé et est en partie financé par l'état via le programme d'investissement d'avenir. Ce travail s'inscrit dans le cadre du projet INDEED, qui vise à étudier différents aspects relatifs au déploiement et à l'exploitation d'un Cloud distribué et économe en énergie.

Contrairement aux FPGAs – qui sont des circuits directement disponibles dans le commerce – les overlays doivent être conçus et implémentés sur FPGA avant de pouvoir être utilisés. Aussi, dans un cadre Cloud, les overlays doivent fournir des mécanismes permettant à un système (de type système d'exploitation) de les contrôler pour partager des overlays entre différentes applications. Notamment,

les overlays doivent permettre la préemption des applications et offrir un temps de reconfiguration minimal.

Pour permettre l'exploration de l'espace de conception des overlays, le prototypage d'overlay doit pouvoir être automatisé, c'est-à-dire que la génération de la description RTL des overlays ne doit pas faire l'objet d'une écriture manuelle de la part du concepteur, et les sources RTL des overlays doivent pouvoir être synthétisées sur FPGA sans intervention manuelle telle qu'une étape de floorplaning.

Les overlays doivent ensuite pouvoir être déployés sur des plateformes FPGA physiques. L'intégration système des overlays doit supporter à la fois la variabilité des overlays intégrés, mais aussi la variabilité des plateformes physiques utilisées pour les héberger. Ces plateformes doivent pouvoir être connectées à un réseau informatique classique, et rendre homogène l'accès aux overlays qu'elles intègrent, quels que soient l'overlay et les spécificités physiques des plateformes.

Il faut ensuite être capable d'implémenter des applications sur les overlays déployés : à partir d'une description de l'application, produire le fichier binaire de configuration permettant de configurer l'overlay pour qu'il implémente l'application. L'outil de programmation doit pouvoir s'adapter aux architectures des overlays ciblés, et aussi permettre d'évaluer les performances des applications implémentées, en particulier leur fréquence maximale de fonctionnement sur les plateformes physiques d'exécution. Les outils de programmation doivent pouvoir accompagner le concepteur applicatif dans ses développements, et notamment permettre le débogage des applications à différents stades de leur implémentation.

Finalement, une fois les overlays déployés sur un réseau informatique et les binaires applicatifs prêts à être exécutés, il faut être capable d'orchestrer l'exécution des applications sur les différentes plateformes, et de contrôler dynamiquement l'infrastructure afin de satisfaire à la qualité de service souscrite par les clients tout en suivant la politique d'optimisation (par exemple performance ou économie d'énergie) imposée par l'administrateur du datacenter.

Dans ce manuscrit, nous proposons une approche verticale allant du prototypage à l'exploitation d'overlays, de la prise en main des fondamentaux, la conception des outils, jusqu'à la mise en œuvre d'overlays dans un contexte Cloud. Ce travail est complémentaire des travaux de référence présentés dans l'état de l'art, et sa complétude permet de présenter un environnement modulaire et extensible qui démontre l'ensemble des étapes nécessaires à la mise en œuvre d'overlays dans un cadre Cloud, de la conception à l'exécution sur carte. Ce travail a permis de dégager un ensemble de contributions qui seront listées à la fin de l'état de l'art.

Chapitre 2

État de l'art sur les overlays

Les overlays sont des architectures reconfigurables implémentées sur FPGA. Ils sont une couche intermédiaire entre l'application et le FPGA : l'application est synthétisée/compilée sur l'overlay, et l'overlay est synthétisé sur son FPGA hôte. Ce chapitre présente un état de l'art relatif aux overlays : quels bénéfices apportent-ils ? Quels types d'overlay trouve-t-on dans la littérature ? Quelles approches ont été mises en œuvre pour le prototypage et l'exploitation d'overlays ? Les contributions apportées par ces travaux sont listées à la fin de ce chapitre.

2.1 Bénéfices apportés par les overlays

En tant que couche d'abstraction placée entre les applications utilisateurs et le FPGA physique, les overlays apportent différents bénéfices par rapport à une exécution native sur FPGA. L'analogie logicielle peut être faite avec la machine virtuelle Java (JVM), qui apporte la portabilité des bytecodes Java sur n'importe quelle plateforme et système d'exploitation capable d'exécuter la JVM, et en exécutant du code objet, permet un gain de productivité de la part des concepteurs d'applications.

2.1.1 Virtualisation

En tant que couche d'abstraction placée entre l'application et le FPGA, l'overlay permet de virtualiser les ressources du FPGA, afin de faciliter leur exploitation dans un contexte Cloud.

Historiquement, la virtualisation de ressources matérielles reconfigurables de type FPGA a été considérée dans la recherche depuis une vingtaine d'années, mais les overlays n'ont pas tout de suite été considérés comme solution de virtualisation. Les premiers concepts de virtualisation adressent la limitation en ressources

reconfigurables et en entrées/sorties des FPGAs [14, 15]. Ils proposent d'exploiter la capacité de reconfiguration des FPGAs afin de charger et extraire des modules matériels dans/depuis un FPGA de manière similaire à la pagination de la mémoire par un système d'exploitation (SE), où des segments mémoires sont échangés entre la RAM et le disque dur. Le but est de donner aux applications une vue virtuelle du FPGA, comme ayant un nombre illimité de ressources et étant entièrement dédié à chaque application.

D'autres travaux [16, 17, 18] vont dans ce sens de la gestion des ressources FPGA par un SE, en partageant le FPGA spatialement et dans le temps. Les FPGAs récents proposent la fonctionnalité de reconfiguration partielle dynamique (DPR), permettant de reconfigurer dynamiquement différentes zones du FPGA indépendamment les unes des autres, c'est-à-dire que la reconfiguration d'une zone ne perturbe pas l'exécution des autres zones. Ainsi, la DPR peut être utilisée pour partager spatialement le FPGA, et configurer différents modules au cours du temps. Cependant, un point clef pour un partage efficace de ressources est la capacité de préemption, c'est-à-dire de suspendre l'exécution d'un module afin de libérer la ressource au profit d'un autre, pour la reprendre plus tard. En plus de la capacité de reconfiguration, le mécanisme de préemption nécessite donc aussi de pouvoir sauvegarder et restaurer l'état d'exécution des modules. Cependant, les FPGAs ne possèdent pas nativement un tel mécanisme de sauvegarde et restauration. Différentes approches existent [19] : relire le bitstream pour extraire les informations d'état, ou instrumenter le circuit applicatif pour permettre le chargement et l'extraction d'état [20].

Les travaux cités ci-dessus traitent la virtualisation des FPGAs par leur partage temporel et spatial, ce qui permet à un système d'exploitation de donner aux applications une vue virtuelle du FPGA physique. Une approche différente pour la virtualisation de ressources FPGA est le concept de *FPGAs virtuels*. Les FPGAs virtuels ont été introduits par Lagadec et al. [21] comme des architectures FPGA synthétisées sur des FPGAs physiques, c'est-à-dire des overlays. Lagadec et al. ont exposé les avantages d'une telle approche comme étant la portabilité des circuits applicatifs et la possibilité d'expérimenter avec de nouvelles architectures FPGA. Ils ont aussi souligné des désavantages tels que le surcoût en ressources physiques, la diminution de la fréquence d'horloge, et le besoin d'outils de synthèse ciblant l'architecture du FPGA virtuel.

Notons que le concept de d'overlay n'est pas incompatible avec l'approche de virtualisation par un système d'exploitation (partage temporel et spatial). Au contraire, du fait que les architectures overlays ne sont pas contraintes par les architectures des FPGAs hôtes, il est possible de concevoir des overlays qui implémentent des mécanismes de sauvegarde et de restauration de contextes permettant la préemption des applications, et ainsi faciliter la gestion des overlays par un système d'exploitation. Ainsi, les overlays se prêtent bien à une exploitation dans un cadre Cloud, qui nécessite une exploitation dynamique des ressources.

2.1.2 Portabilité

Dans le contexte d'un datacenter, différentes générations de ressources sont exploitées simultanément. Il est donc nécessaire de pouvoir supporter l'hétérogénéité des ressources de calcul. Pour l'exécution logicielle, l'hétérogénéité des ressources informatiques est bien supportée : la majorité des processeurs utilisés dans les datacenters partagent le même jeu d'instruction (le x86), et des mécanismes tels que les machines virtuelles permettent d'exécuter un système d'exploitation depuis un autre. Cependant, dans le cas des FPGAs, l'hétérogénéité des FPGAs fait obstacle d'une part à la gestion homogène des ressources, et d'autre part à la productivité des concepteurs applicatif (les clients du Cloud). En apportant la portabilité applicative sur FPGA, les overlays sont à même de faciliter l'exploitation dans le Cloud de FPGAs différents.

Il n'y a pas de compatibilité binaire entre différents FPGAs. Le bitstream d'une application est le fichier binaire de configuration permettant de configurer un FPGA pour implémenter une application donnée. Chaque bit du bitstream participe à la configuration d'un élément configurable du FPGA. Ainsi, même si deux FPGAs d'une même famille partagent le même format de bitstream, un bitstream produit pour un modèle de FPGA ne peut pas configurer un FPGA d'un modèle différent. Or, dans le contexte d'un datacenter, où différentes générations de matériel son exploitées simultanément et présentent donc un ensemble hétérogène de ressources, la non-compatibilité des binaires applicatifs avec certaines ressources de l'infrastructure fait obstacle à une gestion efficace de cette infrastructure.

Les FPGAs n'offrent pas non plus de compatibilité au niveau des sources applicatives. Suivant leur modèle et leur constructeur, les FPGA présentent différentes spécificités, telles que le fonctionnement, l'interface et les possibilités offertes par les blocks DSP et les blocks mémoires qu'ils intègrent. D'une part, ces différences entre FPGAs rendent le portage d'une application d'un FPGA à un autre non trivial, et peut demander jusqu'à re-concevoir entièrement l'application. D'autre part, ces incompatibilités entre FPGAs nuisent au "design reuse" sur FPGA, et donc à la productivité des concepteurs d'applications.

Kirchgessner et al. adressent le problème de la portabilité des sources en introduisant VirtualRC [22]. VirtualRC est une couche d'abstraction qui agit comme un "wrapper" : le concepteur écrit son circuit dans une entité "top level", l'interface de laquelle fournit des connexions à des mémoires, des multiplieurs et des canaux de communication. VirtualRC est portée sur différents FPGAs, implémentant selon les spécificités de chaque FPGA les mémoires, multiplieurs et canaux de communication, tout en gardant une interface fixe pour le "top level" offert au concepteur. Le code applicatif écrit dans le top level de VirtualRC n'instancie pas directement de composants spécifiques au FPGA et est donc portable tel quel sur tous les FPGAs pour lesquels le framework VirtualRC a été porté, quel que soit l'outil de synthèse constructeur utilisé. Cette approche résout le problème de la portabilité des sources : l'effort de portage est mutualisé sur le portage de du framework et non sur chaque application. Cependant, cette approche ne répond pas au problème de

l'incompatibilité binaire entre différents FPGAs.

L'overlay étant une architecture reconfigurable placée entre l'application et le FPGA hôte, il apporte son propre modèle d'exécution (cf figure 1.1) et isole l'application du FPGA sous-jacent ; l'application ne voit que l'overlay. L'application doit donc se conformer au modèle d'exécution de l'overlay, et non à celui du FPGA hôte. Par conséquent, l'overlay apporte par nature la portabilité au niveau des sources sur tous les FPGAs pour lesquels il a été synthétisé.

Au-delà de la portabilité des sources applicatives, l'overlay apporte aussi la compatibilité binaire entre différents FPGAs hôtes. En effet, le mécanisme de configuration de l'overlay est implémenté dans l'overlay lui-même, donc le rôle de chaque bit de configuration du bitstream ciblant l'overlay est propre à l'overlay et est indépendant du FPGA sous-jacent. Ainsi, si une application est synthétisée pour un overlay, alors elle peut être exécutée de manière transparente sur tout FPGA supportant l'overlay, sans nécessiter de re-synthétiser l'application. L'effort de portage et donc mutualisé sur le portage de l'overlay d'un FPGA à un autre, et non des applications. Par exemple, dans [23], les auteurs ont démontrés la compatibilité binaire apportée par leur overlay grain fin Zuma entre un FPGA Xilinx et un FPGA Altera.

Finalement, l'overlay apporte l'indépendance par rapport aux outils de synthèse constructeur. Effectivement, l'architecture de l'overlay étant indépendante de celle du FPGA hôte, l'outil de synthèse ciblant ce dernier ne peut pas être utilisé pour synthétiser des applications sur l'overlay. Il est donc nécessaire d'obtenir un outil de synthèse adapté à l'architecture de l'overlay ciblé [21]. En revanche, cet outil est indépendant des outils constructeurs, et le même outil peut être utilisé pour synthétiser sur overlay quel que soit le constructeur et le modèle du FPGA hôte. Dans le contexte du Cloud, les overlays permettent donc de présenter un environnement de développement applicatif uniforme, quels que soient les FPGAs hôtes physiques présents dans l'infrastructure des datacenters.

2.1.3 Abstraction des ressources physiques

Les overlays permettent de changer le modèle d'exécution par rapport à celui du FPGA. Comme nous l'avons vu, l'overlay apporte son modèle d'exécution. Les applications voient ce modèle d'exécution et non celui du FPGA sous-jacent. L'overlay peut présenter un modèle d'exécution au même niveau que celui du FPGA hôte, comme c'est le cas des overlays grain fin tels que les FPGA virtuels, qui présentent à l'application des LUTs et des pistes de routage virtuelles. L'architecture de l'overlay étant indépendante de celle de son hôte, l'overlay peut aussi présenter un modèle d'exécution de plus haut niveau, comme un réseau d'opérateurs interconnectés par un réseau de routage au niveau mot (et non bit).

Grossir la granularité du modèle d'exécution de l'overlay d'une part de diminuer le temps de synthèse/compilation des applications, et d'autre part d'augmen-

ter la productivité des concepteurs d'applications. En effet, le concepteur n'a plus besoin d'une expertise bas niveau, et la diminution du temps de compilation permet d'augmenter le nombre d'itérations débogage-édition-implémentation. En utilisant un overlay gros grain comme cible de compilation intermédiaire, le développement applicatif se rapproche d'une expérience de développement logiciel.

Ces avantages ont motivé les travaux de Stitt et al. sur les "intermediate fabrics" (IF) [24, 25]. Les IFs sont des overlays composés d'opérateurs tels que des cœurs en virgule flottante, en virgule fixe, des FFTs (Fast Fourier Transform) ou encore des filtres numériques. Par le choix des opérateurs quelles intègrent, les IFs peuvent être spécialisées par application ou par domaine applicatif. Ainsi, pour son application, l'utilisateur peut choisir une IF à partir d'une bibliothèque d'IFs pré-synthétisées, ou générer une IF personnalisée. Générer une IF demande de passer par les outils de synthèse constructeur, et implique donc le temps de synthèse d'un design FPGA. Cependant, une fois l'IF synthétisée sur FPGA, l'IF n'a plus à être modifiée pour chaque itération du développement applicatif. Dans [24], les auteurs rapportent un gain de $554\times$ en temps de compilation ciblant les IFs par rapport à une synthèse native de l'application sur FPGA.

Un autre avantage de la montée en grain du modèle d'exécution et de faciliter le débogage. Si les applications ciblant l'overlay sont décrites dans un langage généraliste qui peut aussi être compilé sur un processeur, le débogage applicatif peut se faire sans outils de simulation. C'est le cas par exemple de MARC [26], dont l'architecture présente plusieurs cœurs de processeurs et chemins de données reliés par un réseau d'interconnexion. Les applications ciblant MARC sont écrite dans le langage OpenCL [5], ce qui permet de compiler et d'exécuter la même application de manière transparente sur MARC et sur processeur, sans avoir à modifier l'application. Le débogage de l'application peut donc se faire directement de manière logicielle sur processeur, sans outils spécifiques de simulation ciblant MARC.

2.1.4 Nouvelles fonctionnalités et usages

Les overlays permettent d'ajouter de nouvelles fonctionnalités aux FPGAs. L'overlay est une architecture reconfigurable, mais du point de vue du FPGA, l'overlay est un design classique. L'espace de conception offert par un FPGA étant très large, il est possible d'implémenter tout type d'overlay sur FPGA. Par exemple, les unités fonctionnelles (UF) de l'overlay peuvent très bien être des opérateurs arithmétiques classiques, ou encore des LUTs. Pour des applications cryptographiques, on peut aussi imaginer des unités fonctionnelles réalisant des multiplications en corps de Galois.

L'apport en fonctionnalités par l'overlay ne se limite pas aux capacités de ses UFs, mais s'étend aux fonctionnalités d'usage de l'overlay. Par nature, un overlay apporte la fonctionnalité de reconfiguration dynamique, même si de FPGA sous-jacent ne le permet pas. En effet, l'overlay intègre son propre mécanisme de configuration, indépendant de celui du FPGA hôte. L'overlay peut donc être reconfiguré

sans qu'il n'y ait besoin de reconfigurer l'hôte par un nouveau bitstream FPGA.

Le plan de configuration de l'overlay peut être segmenté de façon à permettre la reconfiguration partielle dynamique [27], et le mécanisme de configuration peut être conçu de façon à accélérer le chargement de la configuration dans l'overlay. Par exemple, dans [28] Sekanina et al. étudient les circuits matériels évolutifs, dans lesquels les connexions entre les composants et les composants eux mêmes évoluent. Pour se faire, les auteurs utilisent des architectures reconfigurables, et font évoluer les circuits en effectuant des "mutations" sur les binaires de configuration. Ils utilisent des architectures overlay pour bénéficier de temps de reconfiguration supérieurs à ceux possibles nativement sur le FPGA, de façon à augmenter la vitesse d'évolution des circuits.

Il est aussi possible avec les overlays de concevoir un plan de configuration multi-contextes, permettant de commuter en un cycle d'horloge la configuration active avec différentes configurations pré-chargées, et ainsi augmenter le taux d'utilisation des ressources de calcul de l'overlay de manière similaire aux DPGA (Dynamically Programmable Gate Arrays) d'André DeHon [29].

Nous avons aussi évoqué en 2.1.1 la possibilité d'intégrer dans l'overlay un mécanisme de sauvegarde et de restauration du contexte d'exécution de l'overlay afin de permettre à un système d'exploitation de gérer celui-ci au même titre que le processeur ou la mémoire RAM, pour qu'il présente aux applications une vue virtuelle de l'overlay.

2.2 Types d'overlays et optimisations

On peut classer les overlays suivant la granularité de leur modèle d'exécution, c'est-à-dire la granularité de leurs unités fonctionnelles et du réseau d'interconnexions reconfigurables.

2.2.1 Overlays grain fin

On parle d'overlays grain fin lorsque les ressources de calcul et de routage de l'overlay opèrent sur une largeur de mot de 1 bit. Un exemple d'overlay grain fin sont les FPGAs virtuels [21] : les LUTs ne produisent qu'un bit de sortie et le routage des signaux est indépendant pour chaque bit. De par leur granularité, ces overlays présentent un espace de conception équivalent à celui des FPGAs, c'est-à-dire qu'ils sont généralistes et permettent d'implémenter tout type d'application, du contrôle au flot de données.

En revanche, les overlays grain fin sont ceux qui présentent un coût le plus important. En effet, plusieurs ressources FPGA sont mises en œuvres pour implémenter une ressource de l'overlay ; mais dans le cas d'un overlay grain fin, chaque

ressource fonctionnelle de l'overlay (qui consomme plusieurs ressources natives) a une capacité de calcul équivalente à une seule ressource native. Dans le cas d'un FPGA virtuel par exemple, plusieurs LUTs et flip-flops physiques sont utilisées pour implémenter une LUT virtuelle, mais cette LUT virtuelle n'a pas une capacité de calcul supérieure à celle d'une seule des LUTs physiques qu'elle consomme. Aussi, la finesse de la granularité de configuration de l'overlay implique un nombre important de bit dans le binaire de configuration virtuel. Là où un seul mot de configuration suffit pour configurer une unité fonctionnelle d'un overlay gros grain à effectuer une addition par exemple, il faudra une configuration bien plus importante pour configurer toutes les LUTs et les interconnexions nécessaires pour implémenter cette même addition sur un overlay grain fin.

Dans [30, 31], Lysecky et al. réalisent un FPGA virtuel avec un nombre surdimensionné de ressource de routage afin d'alléger la tâche de routage lors de la synthèse applicative, dans le but de permettre la synthèse "just-in-time" de designs matériels pour un usage des FPGAs similaire à la compilation just-in-time de programmes logiciels. Les auteurs rapportent un surcoût en performances de $6\times$ et un surcoût en surface (rapport LUTs physiques occupées / LUTs virtuelle réalisées) de $100\times$.

Zuma [23, 32] est un autre exemple d'overlay grain fin, aussi de type FPGA virtuel. Brant et Lemieux ont diminué le surcoût en surface en cherchant à tirer parti des spécificités architecturales du FPGA hôte. Ainsi, ils utilisent la capacité de reconfiguration des LUTs physiques – appelées LUTRAM – pour implémenter les LUTs virtuelles directement dans les LUTs physiques et remplacer les multiplexeurs des ressources de routage virtuelles par des LUTs physique en mode "route through". Cependant, ces optimisations spécifiques à l'hôte contraignent l'architecture de l'overlay et font perdre à l'overlay son indépendance par rapport à son hôte, le rendant aussi moins portable. Aussi, le mécanisme de reconfiguration des LUTRAMS apporte son propre coût. Les auteurs arrivent à diminuer le surcoût en surface de ZUMA jusqu'à un facteur de $40\times$, soit trois fois moins qu'une implémentation générique de Zuma.

Dans [33], Koch et al. cherchent à créer un processeur softcore comprenant une extension de jeux d'instructions reconfigurable et portable. Pour satisfaire à la portabilité de l'extension sur différents FPGAs, les auteurs implémentent cette extension par un petit overlay grain fin intégré dans le chemin de donnée du processeur. Dans son implémentation générique, l'overlay a un coût en ressources plus élevé que celui du processeur softcore lui-même. Partant du constat que les ressources virtuelles de routage d'un overlay sont en nombre supérieur aux ressources virtuelles de calcul, mais sont aussi celles qui consomment le plus de ressources du FPGA hôte, les auteurs implémentent les ressources de routage de l'overlay directement dans les ressources de routage de l'hôte (sans utiliser de LUTs de l'hôte). En d'autres termes, les multiplexeurs implémentant les ressources de routage de l'overlay ne sont pas implémentés par des LUTs de l'hôte, mais sont implémentées directement via les éléments de routage du FPGA. L'implémentation d'un tel overlay est plus compliquée que la synthèse d'un design classique sur FPGA : l'utilisateur FPGA n'a pas la maitrise des ressources de routage du FPGA depuis le

code RTL applicatif. Il faut pour cela des techniques similaires à l'utilisation de la DPR pour réserver différents chemins dans le routage, allant de différentes sources à une même destination (plusieurs pilotes pour un même signal). Cette méthode d'implémentation d'overlays demande une certaine expertise des outils constructeurs, mais permet de diminuer le surcoût de l'overlay significativement : les auteurs rapportent un surcoût $21\times$ inférieur à celui d'une implémentation générique, et $3.7\times$ inférieur à celui d'une implémentation via les LUTRAM (méthode de Zuma).

Bien que cette méthode soit intéressante par rapport au surcoût, elle a cependant le désavantage de lier la configuration de l'overlay à celle de l'hôte. Notamment, les binaires de configuration de l'overlay sont générés par l'outil de synthèse constructeur, et la reconfiguration de l'overlay demande la reconfiguration partielle du FPGA hôte. L'overlay perd alors son bénéfice de portabilité et ne permet plus d'instrumenter son mécanisme de reconfiguration indépendamment de l'hôte.

2.2.2 Overlays gros grain

Dans la section précédente, nous avons vu que les overlays grain fin sont généralistes au niveau des applications qu'ils permettent d'implémenter, mais présentent un surcoût important. Les travaux [23, 33] diminuent ce surcoût en optimisant l'implémentation de l'overlay sur son hôte. Une autre approche est d'optimiser l'architecture de l'overlay par rapport au domaine applicatif ciblé, c'est-à-dire de grossir le grain jusqu'à ce que le modèle d'exécution de l'overlay corresponde à celui du domaine applicatif : plutôt que de proposer des unités fonctionnelles généralistes de type LUT, l'overlay propose alors des unités fonctionnelles qui correspondent aux opérateurs propres au domaine applicatif ciblé. L'overlay perd en flexibilité, mais il gagne en performance et sa surface diminue. En effet, en augmentant la capacité de calcul des unités fonctionnelles, le rapport entre le coût en ressources physiques occupées et la capacité de calcul totale de l'overlay diminue. Par exemple, une LUT virtuelle implémentée de manière générique et un additionneur occupent une surface de même ordre de grandeur, alors que la capacité de calcul de l'additionneur et supérieure à celle de la LUT.

Les overlays gros grain sont très variés, proposant des unités fonctionnelles allant de simples opérateurs [34] à des cœurs de processeur [26, 35]. Le réseau d'interconnexion entre les unités fonctionnelles peut permettre les connexions de point à point [24], seulement aux plus proches voisins [34], ou encore être de type bus [26, 35].

Dans [36], Jain et al. proposent une classification des overlays gros grain en deux catégories : les overlays configurés spatialement, et les overlay à temps partagé. Dans un overlay à temps partagé [37, 38], la configuration du réseau d'interconnexion est des unités fonctionnelles change à chaque cycle au cours de l'exécution d'une même application. Les unités fonctionnelles se comportent comme des cœurs de processeur (ou sont effectivement des processeurs [26, 35]), c'est-à-dire

que leur ALU est contrôlée à chaque cycle par une mémoire d'instruction qui leur est propre. Ce partage dans le temps des ressources de l'overlay entre différentes opérations atomiques permet de diminuer le nombre de ressources physique nécessaire pour implémenter l'application, mais augmente aussi le temps d'exécution nécessaire ainsi que la complexité des outils de compilation.

Les overlays configurés spatialement sont les plus fréquents dans la littérature [34, 24, 39, 37]. Pour ces overlays, la configuration est statique durant l'exécution entière de l'application. C'est-à-dire que si une unité fonctionnelle est configurée pour réaliser une multiplication par exemple, alors cette unité fonctionnelle ne réalisera que des multiplications durant l'exécution de l'application. Ce modèle de fonctionnement permet de paralléliser spatialement l'exécution des applications en flux de données sous forme de pipelines, et ainsi de tirer parti de l'abondance des ressources d'un FPGA. Par contre, ces architectures gèrent difficilement le flot de contrôle. Les travaux [40, 41] proposent des architectures overlays permettant de réaliser des machines d'état, et ainsi élargir le spectre applicatif des overlays gros grain aux applications nécessitant du contrôle.

Les overlay gros grain présentent un surcoût moindre que leur homologues grain fin. Pour les IFs, [25] rapporte un surcoût de 7% en performance et de 34% à 44% en surface. Tout comme pour les overlays grain fin, il est possible d'optimiser l'architecture et l'implémentation d'un overlay gros grain par rapport au FPGA sous-jacent. Ainsi, dans [39], Jain et al. construisent les unités fonctionnelles de leur overlay autour des blocks DSP DSP48E1 des FPGAs Xilinx. L'overlay est capable de fonctionner à des fréquences proches de la fréquence maximale théorique du FPGA, et les auteurs rapportent même une amélioration de 11 à 52% du débit des applications comparé à une implémentation native de ces applications sur le FPGA en utilisant l'outil de synthèse haut niveau Vivado HLS (de Xilinx).

Dans [42], Coole et Stitt présentent une famille d'architectures overlay qu'ils appellent *supernets* (pour *superset* de *netlists*) dont le but est de minimiser le surcoût en surface, au détriment de la configurabilité de l'overlay. Pour ce faire, l'architecture est spécialisée suivant un ensemble de netlists en fusionnant les ressources partagées par les différentes netlist. Les auteurs rapportent un surcoût en surface $8.9\times$ inférieur à celui de l'IF [25] minimale permettant d'implémenter les netlists de l'ensemble de départ. Ensuite, un réseau d'interconnexion secondaire est intégré à l'overlay pour lui apporter de la flexibilité et permettre d'implémenter des netlists non présentes dans l'ensemble de départ. Ce réseau vient se connecter aux unités fonctionnelles du supernets. Lors de la compilation applicative, l'outil cherche les parties de la netlist applicative qu'il peut implémenter directement avec les ressources supernets, puis implémente le reste de la netlist via le réseau secondaire. Ajuster la flexibilité du réseau d'interconnexion secondaire permet de jouer sur le compromis entre la flexibilité et le surcoût de l'overlay.

Dans un contexte Cloud, les overlays sont donc une solution intéressante pour faciliter l'adoption des FPGAs en tant qu'accélérateurs de calculs reconfigurables :

— ils sont plus simples à programmer que les FPGAs, notamment pour les clients du Cloud qui sont majoritairement des développeurs logiciel : l'ex-

périence de développement applicatif sur overlay se rapproche d'une expérience logicielle, le développement ne se fait pas à bas niveau, l'utilisateur n'a pas besoin d'expertise particulière, les itérations de développement sont raccourcie grâce à une compilation rapide, et le débogage et la simulation sont simplifiés. De plus, les outils de compilation sont indépendants des outils de synthèse constructeur, ce qui permet de présenter aux développeurs applicatifs un environnement uniforme quels que soient les FPGAs physique utilisés dans les datacenter. Finalement, la portabilité apportée par les overlays permet le design-reuse, et donc un gain de productivité de la part des développeurs.

— les overlays facilitent l'exploitation dynamique et le partage des FPGAs physiques sous-jacents : Les overlays offrent potentiellement des mécanismes facilitant leur contrôle par un système d'exploitation en permettant le partage dans le temps de ses ressources.

Finalement, le surcoût des overlays en surface et en fréquence peut être minoré en spécialisant l'architecture par domaine applicatif et en optimisant l'implémentation de l'overlay par rapport aux FPGAs hôtes.

2.3 Du prototypage à l'exploitation d'overlays

Pour pouvoir explorer l'espace offert par le concept d'overlays et jouer sur différents compromis, il est nécessaire de pouvoir les évaluer suivant différents aspects, comme leur occupation en ressources, leur performance en fréquence, et la flexibilité des architectures par rapport aux applications. Pour se faire, il faut être capable de concevoir, générer et synthétiser des overlay sur FPGAs, de compiler des applications sur overlay, et pour pouvoir l'exploiter, d'accéder a l'overlay synthétisé ainsi que de le contrôler.

2.3.1 Conception, implémentation et synthèse d'overlays

Concevoir une architecture overlay consiste à concevoir et dimensionner les ressources reconfigurables visibles par l'outil synthèse/compilation pour implémenter des applications sur l'overlay, et aussi à concevoir les ressources de support tel que le mécanisme de reconfiguration, afin d'aboutir à une description RTL synthétisable de l'overlay. Il s'agit donc de concevoir une architecture reconfigurable, que celle-ci soit ensuite implémentée sur FPGA pour former un overlay, ou qu'elle soit directement implémentée sur silicium pour former un circuit reconfigurable ASIC. Les outils existants de conception d'architectures reconfigurables peuvent donc être utilisés dans le cas des overlays, même si ceux-ci n'ont pas été fait spécifiquement pour les overlays. Parmi les outils les plus utilisés dans la littérature pour la conception d'overlay, VPR et Madeo sont les plus fréquemment utilisés.

VPR [43] est un outil de placement et routage sur architectures reconfigurables

de type FPGA [44]. Pour réaliser le placement et routage d'une netlist sur une architecture, une description haut niveau de l'architecture doit être fournie à VPR [45]. À partir de cette description, l'outil génère le graphe des ressources logiques et de routage. Il est donc possible d'utiliser VPR pour générer le graphe de routage d'une architecture à partir de paramètres hauts niveaux, ce qui est le cas par exemple de Zuma [23]. Cependant, VPR ne prend en compte que les éléments de l'architecture vus par les applications, et ne s'intéresse pas à leur implémentation. Par exemple, VPR permet d'intégrer des macro-blocks tels que des block DSP dans les architectures qu'il cible, VPR prend alors en compte les connexions offertes par ces macro-blocks, mais n'a aucune information sur l'implémentation de ces blocks ni même la fonction logique qu'ils réalisent. Aussi, VPR ne prend en compte aucune information relative aux mécanismes de configuration de l'architecture. La seule information physique qui est prise en compte par VPR est la surface occupée par les ressources reconfigurables, permettant ainsi d'évaluer la surface totale occupée par l'architecture. VPR est donc un outil efficace pour l'exploration fonctionnelle d'architecture reconfigurables de type FPGA, il permet d'élaborer le graphe de routage "à plat" des ressources reconfigurable de l'architecture à partir de paramètres hauts niveaux, mais ne peut pas être utilisé tel quel pour générer la description RTL d'une architecture.

Madeo est un framework logiciel pour la modélisation et la programmation d'architectures reconfigurables, principalement développé par Loïc Lagadec [46]. Madeo permet d'adresser un spectre d'architectures plus large que VPR, et n'est pas restreint aux architectures grain fin. La généricité de Madeo est assurée par une conception fondée sur le paradigme logiciel orienté objet [47] : à partir d'une description structurelle de l'architecture exprimée dans un DSL (Design Specific Language), Madeo modélise l'architecture et génère les outils de floorplaning, placement et routage la ciblant. Le framework Madeo a ensuite été augmenté [27, 48] pour permettre de modéliser non seulement le graphe des ressources vues par les applications, mais aussi les mécanismes de configurations de l'architecture, et ainsi permettre de générer une description RTL (en VHDL) simulable et synthétisable de l'architecture modélisée. Le prototype matériel d'une architecture gros grain a été généré avec Madeo- et implémenté sur FPGA avec succès. Cependant, le code HDL généré par le framework cible des cibles ASIC, et non particulièrement des FPGAs, et la description RTL générée présente des structures logiques qui se prêtent mal à une implémentation sur FPGA. Par exemple, beaucoup de signaux (tels que les pistes de routage) sont décrits par de la logique à trois états, qui, dans les FPGAs, sont implémentés par plusieurs signaux unidirectionnels (en dehors des entrées/sorties).

QuickDough [37] permet d'accélérer sur FPGA les boucles de calcul intensif de code logiciel. Pour ce faire, après avoir analysé la boucle à accélérer, l'outil compile le graphe de flot de donnée (DFG) de la boucle sur un overlay gros grain de type CGRA (Coarse Grained Reconfigurable Array) [49]. L'overlay minimal nécessaire pour implémenter le DFG est sélectionné depuis une bibliothèque d'overlays pré-synthétisés, ou est généré s'il n'est pas déjà disponible. Les overlays générés par QuickDough sont des overlays à temps partagé, et non configurés spatialement (cf 2.2.2). Ainsi, les mémoires de configuration des unités fonctionnelles peuvent

être implémentées dans des blocks mémoires du FPGA hôte (chaque adresse mémoire contient la configuration d'un seul cycle), ce qui n'est pas possible dans le cas d'un overlay configuré spatialement (où la configuration de toutes les ressources doit être disponible en même temps, ce qui présenterait une largeur de données des mémoires qui n'est pas offerte par les block-RAM des FPGAs commerciaux). La configuration de l'overlay se fait en écrivant le contenu des mémoires de configuration (issu de compilation du DFG) directement dans le bitstream de l'overlay (pré-)synthétisé, ce qui est possible via les outils Xilinx. QuickDough permet donc de créer en quelques secondes un accélérateur sur FPGA pour une application donnée, car il ne passe pas nécessairement par les étapes de placement et routage de l'outil de synthèse constructeur. Cependant, le modèle de l'architecture overlay utilisé est figée, et tout changement architectural demande une modification de l'outil. De plus, l'approche de QuickDough ne fonctionne pas dans le cas d'un overlay configuré spatialement.

La méthodologie Rapid Overlay Builder (ROB) [50] vise à réduire le temps de synthèse des overlays sur leur FPGA hôte. Le concepteur fournit la description RTL des différentes tuiles formant la matrice de son overlay. ROB inclu un ensemble de scripts et d'outils interagissant avec la suite d'outils ISE de Xilinx qui permettent de ne synthétiser, placer et router qu'une seule instance de chaque tuile sur le FPGA, puis d'utiliser la relocation de module pour répliquer les tuiles suivant la matrice lors du floorplaning, qui est effectué manuellement. Ainsi, ROB facilite la synthèse d'overlay, mais seulement sur les FPGA du constructeur Xilinx, et demande une certaine expertise des outils de synthèse constructeur. De plus, ROB nécessite une intervention manuelle de floorplaning, ce qui ne permet pas d'automatiser la synthèse lors du prototypage afin d'explorer le surcoût d'un modèle d'overlay.

La plupart des overlays que l'on trouve dans la littérature sont cependant réalisés au cas par cas (à la main) [28, 30, 31, 33, 34, 39], ou sont générés par des outils qui ne supportent qu'une variabilité limitée, comme les IFs [24, 25].

2.3.2 Programmation des overlays

Par "programmation" des overlays, nous entendons le fait d'obtenir le binaire de configuration ciblant un overlay donné à partir de la description d'une application. La programmation est dépendante de l'architecture de l'overlay ciblé. Suivant l'architecture et le modèle d'exécution que présente l'overlay, la programmation correspond soit à une *synthèse* (comprenant entre autres des étapes de placement et de routage), soit à une *compilation*.

Outils de programmation

Comme nous l'avons vu dans la sous-section précédente, de part sa capacité à modéliser différentes architectures, VPR [43] permet le placement et routage de netlists sur des architectures grain fin de type FPGA. VPR est utilisé pour le placement et routage sur Zuma [23, 51], et aussi sur [1]. Cependant, VPR n'ayant connaissance que des ressources de calcul (LUTs, pistes de routage) et non des ressources de configuration de l'architecture, il ne peut pas générer les binaires de configuration des applications placées et routées. Dans le cas du Zuma, scripts additionnels sont nécessaires pour assembler les binaires de configuration à partir des sorties de VPR.

VPR peut aussi être utilisé pour réaliser le placement et routage sur des overlays gros grain qui, hormis la granularité des ressources de routage et des unités fonctionnelles, présentent une architecture similaire aux FPGAs, c'est-à-dire des overlays configurés spatialement et qui ont un réseau de routage commuté qui permet de connecter entre elles n'importe quelles unités fonctionnelles de la matrice. C'est le cas par exemple des IFs [24] : VPR route les signaux de plusieurs bits comme s'il ne s'agissait de signaux que d'un seul bit. Les opérateurs des unités fonctionnelles (tel qu'un opérateur arithmétique ayant deux entrées et une sortie) sont alors vus par VPR comme une simple LUT à deux entrées.

Madeo [48] permet de générer les outils de placement et routage spécifiques aux architectures qu'il modélise. Madeo permet de modéliser un spectre d'architectures plus large que VPR, et donc de compiler des applications sur d'autre cibles que des architectures de type FPGA classique. Aussi, comme les ressources de configuration sont modélisées en même temps que les ressources de calcul, Madeo est capable de générer les binaires de configuration des applications compilées/placées-routées pour l'architecture ciblée. Cependant, le framework Madeo doit être vu comme une bibliothèque pour la modélisation et la programmation d'architectures reconfigurable, qui aide à la conception d'outils, mais non comme un outil utilisable tel quel. Ainsi, utiliser Madeo pour cibler une nouvelle architecture peut demander des développements additionnels et une connaissance approfondie du framework.

Outre VPR et Madeo, la plupart des overlays utilisés dans la recherche [39, 37, 34, 40, 41, 30, 31] sont programmés par des outils réalisés spécifiquement pour leur cible, et ne supportent qu'une faible variabilité des architectures ciblées. Dans [33], les auteurs indiquent même qu'ils n'ont réalisé que trois circuits applicatifs pour leur overlay, car ils effectuent le placement et le routage à la main.

Analyse de timing sur overlay grain fin

Un problème qui se pose lors de la programmation des overlays grain fin et l'obtention d'une fréquence maximale de fonctionnement maximale fiable de l'application. Lors de la synthèse classique d'un design sur FPGA, l'outil constructeur a connaissance des délais des ressources physiques empruntées par les chemins

applicatifs, et peut donc déterminer le plus long délai combinatoire de l'application. Lors de la synthèse sur overlay, l'outil de synthèse applicative ciblant l'overlay doit connaitre les délais des ressources virtuelles pour déterminer le chemin critique de l'application, son délai, et en déduire la fréquence maximale de fonctionnement. Cependant, les délais des ressources virtuelles dépendent du FPGA hôte ainsi que de placement et routage de l'overlay sur son hôte par l'outil de synthèse constructeur. À cause de l'absence de registres dans le réseau de routage de l'architecture overlay, il est difficile d'extraire les délais des ressources virtuelles (telles que les pistes de routage virtuelles) depuis la synthèse physique de l'overlay. De plus, deux ressources virtuelles équivalentes (comme deux pistes de routage adjacentes) peuvent avoir des délais physiques différents une fois implémentées sur FPGA.

Dans les quelques overlays grain fin présents dans la littérature, seulement deux travaux s'intéressent au problème de l'analyse de timing sur overlay. Dans [51], les auteurs utilisent l'overlay Zuma [23]. Ils utilisent comme fréquence de fonctionnement applicative la fréquence de fonctionnement maximale retournée par l'outil de synthèse constructeur. Or, cette fréquence est extrêmement pessimiste puisqu'elle prend en compte le plus long chemin combinatoire possible dans l'overlay, qui peut être extrêmement long selon la flexibilité du réseau d'interconnexion de l'overlay. Les auteurs proposent en travaux futurs d'extraire les délais des ressources virtuelles de la synthèse physique de l'overlay pour annoter la description de l'architecturale fournie à VPR pour la synthèse virtuelle, mais à notre connaissance, cette méthode n'a jamais été mise en œuvre.

Dans [30], les auteurs évaluent le surcoût en fréquence de leur overlay. En utilisant plusieurs configurations virtuelles différentes, les auteurs mesurent physiquement les délais à travers les différents éléments de l'architecture virtuelle. À cause du nombre élevé de ressources virtuelles d'un overlay grain fin, cette méthode peut être extrêmement longue à mettre en œuvre. De plus, les outils de synthèse constructeur utilisent des heuristiques lors du placement et routage, donc pour plusieurs synthèses d'un même overlay sur un même FPGA, l'overlay peut avoir un placement et routage différent suivant les synthèses. Ainsi, les mesures des délais doivent être effectuées à chaque nouvelle synthèse de l'overlay, même si l'overlay reste inchangé et que le FPGA hôte reste le même. Dans se travail, nous proposons une méthode fiable et rapide permettant l'analyse de timing sur overlay.

Le problème de l'analyse de timing applicative ne se pose cependant pas pour les overlays gros grain, qui, étant orientés pour le calcul en flux de données, ont des architectures fortement pipelinées, y compris dans les ressources de routage [34]. Dans ce cas, le délai de chaque ressource virtuelle est un multiple de l'inverse de la fréquence maximale de l'overlay, retournée par l'outil de synthèse constructeur lors de la synthèse physique de l'overlay sur le FPGA hôte.

2.3.3 Accès à l'overlay, contrôle et exploitation

Une fois un overlay conçu, sa description RTL synthétisable générée, et que les outils de programmation permettent de le cibler, il est nécessaire de pouvoir accéder à l'overlay dans son FPGA hôte pour le configurer, l'alimenter en données, et le contrôler. Pour une gestion automatisée de l'overlay – par exemple pour ordonnancer différentes applications – un processeur peut être nécessaire pour orchestrer le contrôle de l'overlay. De plus, en tant qu'accélérateur, l'overlay peut occuper différentes places par rapport à l'exécution d'une application :

— accélérer seulement les parties ponctuelles de calcul intensif (des opérations spécifiques sur une donnée) tout en laissant le reste de l'exécution à un processeur ;
— traiter des parties plus larges (traitement d'un tampon de données) ;
— ou bien exécuter l'intégralité de l'application, auquel cas le processeur n'a pas de rôle calculatoire dans l'exécution applicative, mais seulement un rôle de contrôle.

Dans [33], Koch et al. intègrent leur overlay grain fin dans le chemin de donnée d'un processeur MIPS softcore (ie. un processeur implémenté sur FPGA). Le but et de pouvoir étendre le jeu d'instruction du processeur par des instructions reconfigurables, et que cette extension reste portable quel que soit le FPGA hôte. L'overlay permet d'implémenter des instructions spécifiques à une application, qui auraient demandé un nombre important d'instructions fixes du processeur pour réaliser une fonction équivalente. Il s'agit d'un couplage fort entre l'overlay et le processeur : des instructions fixes ont été ajoutées au processeur pour reconfigurer l'overlay, et celui-ci est alimenté en données depuis la file de registres du processeur. Le contrôle de l'overlay est donc entièrement intégré dans l'application logicielle accélérée.

Cependant, un couplage fort entre l'overlay et le processeur ne permet pas une exécution concurrente : lorsque l'overlay exécute une instruction, les ressources du processeur sont en attente du résultat, et quand le processeur exécute ses instructions fixes, l'overlay est inutilisé. La solution est alors d'utiliser un couplage plus faible entre le processeur est l'overlay : ce dernier est alors vu comme un coprocesseur ; le processeur fournit à l'overlay une tâche à exécuter sur un tampon de données, et continue son exécution pendant que l'overlay s'exécute. Dans [52, 1], les auteurs intègrent un overlay grain fin dans un SoC (System on Chip) softcore basé sur un processeur ARM Cortex-M1. L'overlay est couplé à un contrôleur de configuration, l'overlay et le contrôleur étant accessibles via le bus système du SoC. L'overlay a été conçu avec différents segments de configuration, et permet de sauvegarder et restaurer le contenu des éléments séquentiels (i.e. le contexte applicatif). Le processeur est chargé du contrôle de l'overlay, il peut y placer des applications occupant une ou plusieurs zones reconfigurables séparément. Lorsque la fragmentation des zones empêche de placer une nouvelle application, le processeur peut reloger dynamiquement les applications (grâce au mécanisme d'extraction de contexte) sur l'overlay pour libérer des zones adjacentes, afin de placer la nouvelle application.

Dans [51], Wiersema et al. intègrent l'overlay Zuma [23] dans l'architecture et système d'exploitation ReconOS [53, 54] sur une plateforme Zynq de Xilinx (qui comporte un processeur ARM intégrée dans une matrice FPGA). ReconOS est une architecture matérielle et un environnement d'exécution pour les systèmes hybrides matériels/logiciels. ReconOS permet la création de threads (fils d'exécution) matériels. Ces threads matériels sont gérés par le système d'exploitation au même titre que les threads logiciels, bénéficient des même services de la part de l'OS, peuvent accéder à la mémoire virtualisée de l'OS, et communiquer avec les autres threads (matériels ou logiciels) via des files de messages. L'intégration d'un overlay dans ReconOS délègue le contrôle de l'overlay au système d'exploitation et facilite la conception applicative en mettant au même niveau les threads matériels et logiciels, permettant au concepteur applicatif d'accélérer des portions de son application logicielle en utilisant des threads logiciels ou matériels de façon transparente. Cependant, Wiersema et al. n'intègrent pas de mécanismes de sauvegarde et de restauration du contexte matériel applicatif dans Zuma, ce qui limite les possibilités de gestion de l'overlay par l'OS : un thread matériel doit se terminer avant de pouvoir libérer l'overlay, l'OS ne peut pas réaliser de préemption sur les threads matériels pour mieux gérer le partage de l'overlay entre différents threads.

Cependant, dans la plupart des travaux sur les overlays, la gestion de l'overlay n'est pas adressée et est laissée à la charge de l'utilisateur. Les overlays générés par QuickDough [37] intègrent des tampons d'entrée sorties permettant d'alimenter la matrice de l'overlay en donnée depuis le bus système d'un SoC. De même, Rapid Overlay Builder [50] permet d'intégrer un block pré-synthétisé dans la matrice de l'overlay pour connecter la matrice à des mémoires RAM externes au FPGA, à une connexion Ethernet ou à un ordinateur via une connexion PCIe. Les IFs [25] sont générées avec des générateurs d'adresses et des contrôleurs mémoire pour alimenter la matrice en données directement depuis une mémoire externe. Pour ces trois cas, les overlay fournissent des interfaces pour faciliter leur intégration à un système capable de les contrôler. Cependant, dans d'autres travaux [23, 30, 31, 34, 39], l'intégration système de l'overlay n'est pas adressée non plus : l'overlay est fourni "nu", son intégration et sont contrôle sont à la charge de l'utilisateur.

Exploitation de FPGAs dans un cadre Cloud

Certains travaux adressent l'intégration de FPGAs dans un cadre Cloud ; ils n'utilisent pas d'overlay, mais s'intéressent à la manière de rendre des ressources FPGA disponibles dans le Cloud et comment les intégrer dans le modèle de gestion d'un système Cloud. Dans [55], Chen et al. relèvent quatre exigences fondamentales à adresser :

l'abstraction : les FPGAs doivent être exposés à la pile logicielle de gestion du système Cloud comme un pool de ressources qui peuvent être gérées dynamiquement, allouées, programmées et libérées par les clients ;

le partage : les ressources FPGA doivent permettre leur partage entre différents clients, tout en assurant l'isolation des applications les unes des autres ;

la compatibilité : les applications doivent utiliser une interface prédéfinie pour pouvoir être chargée dans le FPGA et se connecter au système (comme une interface DMA) ;

la sécurité : des systèmes doivent être mis en place pour empêcher une application d'accéder aux données d'une autre, et aussi empêcher que le dysfonctionnement d'une application n'entraine le dysfonctionnement du système qui l'héberge.

Les auteurs réalisent une architecture sur FPGA comprenant quatre régions reconfigurables par DPR et une partie statique gérant la reconfiguration de ces régions ainsi que l'alimentation en données des applications configurées sur ces régions. Les FPGA sont intégrés dans des serveurs, et l'hyperviseur (le logiciel exécutant les machines virtuelles logicielles) a été adapté pour permettre aux machines virtuelles d'allouer une région DPR, la programmer, échanger des données avec l'accélérateur configurer, et le libérer. Cette architecture permet le partage spatial du FPGA en différentes régions, et son partage temporel via la reconfiguration dynamique de ces régions. La partie statique de l'architecture et l'hyperviseur assurent la sécurité du système. Les accélérateurs se connectent à la partie statique du FPGA via une interface DMA.

Pour permettre l'abstraction des ressources FPGA, les auteurs ont ajouté des modules à Openstack pour gérer les régions DPR de leur architecture en plus des ressources processeur et mémoire des serveurs. Openstack [56] et une pile logicielle pour le déploiement d'infrastructures Clouds, gérant différentes plateformes de calcul, de stockage et réseau. Dans une infrastructure basée sur Openstack, un nœud de contrôle (unique) gère les requêtes des clients, déploie des machines virtuelles sur les nœuds de calcul, puis retourne au client une adresse réseau lui permettant d'accéder à la machine virtuelle qui lui a été allouée. Dans [55], lorsqu'un client demande une machine virtuelle avec un accélérateur FPGA, le nœud de contrôle choisi un serveur ayant un FPGA pour déployer la nouvelle machine virtuelle. Le client peut ensuite allouer une région DPR et la programmer depuis sa machine virtuelle.

Dans [57], Byma et al. réalisent une architecture similaire, comprenant des régions DPRs et une partie statique pour les gérer. Ils ont aussi intégré leur système à Openstack, mais via une approche différente : plutôt que d'intégrer les FPGAs dans des serveurs et rendre les régions DPRs accessibles depuis des machines virtuelles via l'hyperviseur, dans [57] les plateformes FPGAs sont directement connectées au réseau, et sont considérées par Openstack comme des nœuds de calcul à part entière. À la requête d'un client, le nœud de contrôle configure une région DPR libre d'un FPGA avec le bitstream client de la même manière qu'il démarrerait une machine virtuelle avec une image (de VM) sur un serveur. Les FPGAs ne sont alors plus utilisés pour accélérer matériellement des portions d'applications logicielles, mais réalisent entièrement les applications. En plus d'un accès à une mémoire RAM, les applications hébergées dans les régions DPRs communiquent avec l'extérieur via une interface Ethernet.

Cependant, l'utilisation de régions DPRs pose encore quelques obstacles à une

intégration transparente des FPGAs dans le Cloud. Notamment, l'hétérogénéité des FPGAs n'est pas supportée : un bitstream client ne peut être exécuté que par sur modèle de FPGA. Supporter un autre modèle de FPGA de l'infrastructure demande de synthétiser à nouveau les applications clientes, mais la non-portabilité des sources d'un FPGA à un autre pose alors problème. Aussi, le client doit utiliser des outils de synthèse différents propres à chaque constructeur de FPGA. Par ailleurs, le partage temporel des FPGAs est limité par le manque de mécanisme de préemption : les régions DPRs doivent être libérées par les clients avant de pouvoir être utilisées par d'autres clients, elles ne sont pas partagées aussi finement que la ressource processeur des serveurs. De plus, les applications clientes exécutées sur FPGA ne peuvent pas être migrées dynamiquement afin d'optimiser l'infrastructure, comme c'est le cas des machines virtuelles qui peuvent être migrées à chaud d'un serveur physique à un autre.

Dans ce travail, nous proposons d'utiliser les overlays afin de lever ces obstacles, tout en reprenant les principes de [57, 55] pour l'intégration Cloud. De manière similaire aux travaux [57], un ou plusieurs overlays sont synthétisés sur FPGA pour accueillir les applications clientes (à la place des régions DPRs), avec une partie fixe permettant le contrôle local du ou des overlays ainsi que leur alimentation en données. Les applications clientes exécutées sur l'overlay se connectent à la partie fixe via une interface mémoire et une interface DMA. Les plateformes FPGA sont connectées directement au réseau, et sont vues comme des nœuds de calcul par le nœud de contrôle. De manière analogue à [52, 51] la partie fixe implémente un SoC auquel sont greffés le ou les overlays. Le contrôle des overlays est réalisé au niveau local (ordonnancement des applications clientes) par un système d'exploitation exécuté sur le SoC, et exécute les requêtes du nœud de contrôle.

2.4 Synthèse et contributions

Les overlays sont selon nous une solution pour permettre l'utilisation des FPGAs dans un contexte Cloud en tant qu'accélérateurs reconfigurables par les clients. Premièrement, les overlays sont plus faciles à adopter que les FPGA pour les utilisateurs du Cloud : ils sont plus simples à programmer, et l'expérience de développement sur overlay se rapproche d'une expérience de développement logicielle. Deuxièmement, les overlays permettent une exploitation plus souple des FPGAs, en assurant la portabilité des applications sur différents FPGAs et en permettant la mise en place de mécanismes facilitant leur contrôle et leur partage. Les overlays permettent aussi d'optimiser l'exploitation d'une infrastructure en rendant possible la migration à chaud d'application d'une plateforme FPGA hôte à une autre.

Les overlays apportent un surcoût en surface et en fréquence, mais il est possible de minimiser ce surcoût, et même d'obtenir de meilleurs performances avec un overlay que via une synthèse native avec un outil de synthèse haut niveau du constructeur [39].

Les overlays sont des objets complexes, et leur mise en œuvre dans un contexte Cloud présente différents aspects :

— de la conception d'une architecture,
— son implémentation RTL,
— sa synthèse physique sur FPGA,
— l'évaluation de son coût en ressources,
— son intégration sur des plateformes hétérogènes,
— l'automatisation de son contrôle,
— la mise en place de l'outillage de programmation applicative,
— jusqu'au déploiement d'overlays dans une infrastructure et leur à gestion selon un modèle Cloud.

Certains de ces aspects sont adressés séparément dans la littérature, mais jamais ensemble, de manière verticale.

Ce travail présente les différents aspects de la mise en œuvre d'overlays dans un cadre Cloud. Notre approche pour la conception d'architectures overlays consiste à explorer l'espace de conceptions suivant deux axes : leur coût physique une fois synthétisées sur FPGA, et leur adéquation avec les applications ciblées. Pour faciliter cette exploration, le processus d'implémentation des overlays est automatisé : l'architecture est modélisée, puis sa description VHDL est générée. Un overlay étant un design atypique pour un FPGA, l'outil de synthèse peut avoir du mal à l'implémenter. Pour permettre la synthèse d'overlays, quelles que soient leur tailles, sans demander d'intervention manuelles (telle qu'une étape de floorplaning), des registres additionnels sont injectés dans les architectures. Finalement, le coût en surface de l'overlay est modélisé suivant les paramètres de l'overlay.

L'implémentation de l'overlay est décomposée en trois plans conceptuels : le plan de calcul, qui correspond aux ressources reconfigurables réalisant les applications, le plan de configuration, et un plan de snapshot. Le plan de snapshot permet de sauvegarder et restaurer l'ensemble des éléments séquentiels du plan de calcul, c'est-à-dire l'état d'exécution de l'application. Il est séparé du plan de configuration afin de rendre plus simple et plus rapide le mécanisme de préemption. Le plan de configuration est réalisé de façon à diminuer le coût de reconfiguration lors de l'ordonnancement d'application sur l'overlay, en permettant le pré-chargement d'une configuration sans perturber l'exécution de l'application en cours.

La synthèse applicative – c'est-à-dire la synthèse d'applications sur un overlay – repose sur un flot modulaire, permettant d'utiliser différents outils existants, tels que VPR et Madeo. Il s'adapte à l'overlay ciblé en utilisant le modèle et les paramètres du plan de calcul de l'architecture. Le flot de synthèse permet d'aboutir à un binaire de configuration propre à l'overlay ciblé à partir d'une description de l'application. Une méthode et des outils sont présentés afin de permettre le débogage applicatif à différents niveaux d'implémentation, de la description RTL de l'application jusqu'à son exécution sur carte. De plus, le problème de l'analyse de timing sur overlay est analysé, une solution est proposée et intégrée au flot, qui permet d'abstraire les performances applicatives des performances du FPGA hôte. En découplant l'analyse de timing sur overlay de l'analyse de timing de l'overlay sur son hôte, cette méthode permet d'une part de séparer complètement le flot de

synthèse applicative du flot de synthèse constructeur (FPGA), et d'autre part de n'avoir à réaliser l'analyse de timing de l'application qu'une seule fois quel que soit le FPGA hôte utilisé, tout en obtenant une fréquence fiable de fonctionnement maximale sur carte.

Pour rendre les overlays utilisables dans une infrastructure de type Cloud, ceux-ci sont intégrés sur des plateformes FPGA avec un système permettant de les contrôler et de les alimenter en données depuis une interface réseau. Selon les composants offerts par les plateformes, ce système peut être entièrement implémenté sur le FPGA avec l'overlay, ou tirer parti de la présence de composants externes au FPGA, tels qu'un processeur. Afin de supporter la variabilité des overlays lors de leur intégration, les overlays sont intégrés en une IP avec différents contrôleurs, tel qu'un contrôleur de configuration et un contrôleur DMA. Cette IP présente la même interface quelles que soient les caractéristiques de l'overlay qu'elle intègre, et peut facilement venir se greffer sur le bus d'un système de type SoC.

Le déploiement des overlays dans une infrastructure de type Cloud est réalisée en connectant les plateformes à un réseau informatique classique. Au niveau de chaque plateforme, des applications sont ordonnancées sur l'overlay. Les temps de sauvegarde et de restauration d'une application sur l'overlay sont modélisés afin de guider la mise en place d'une politique d'ordonnancement suivant les spécificités de chaque plateforme. Suivant un schéma similaire au framework Openstack, un nœud de contrôle unique communique avec chacune des plateformes et orchestre l'ensemble du cluster ainsi formé, répartissant les applications sur les différentes plateformes. Suivant le taux d'utilisation et les capacités de chaque plateforme, le nœud de contrôle peut initier la migration d'applications d'une plateforme à une autre afin d'optimiser l'exploitation totale du cluster. Finalement, l'ensemble de l'environnement réalisé est démontré par la mise en œuvre d'un même overlay sur deux plateformes comprenant des FPGAs différents. Des applications réalisées via le flot de synthèse applicative sont exécutées de manière transparente sur chacune des plateformes, ordonnancées et migrée en cours d'exécution d'une plateforme à l'autre.

Pour démontrer la faisabilité de tous les aspects de la mise en œuvre des overlays, de leur conception à leur exploitation dans un contexte Cloud, nous nous somme placés dans le cas qui, à notre sens, est le plus compliqué, c'est-à-dire en utilisant des overlays grain fin.

2.5 Plan du manuscrit

Le chapitre 3 est centré sur la réalisation d'un overlay, de la conception de son plan de calcul, son implémentation à sa synthèse sur FPGA. Le plan de calcul de deux architectures réalisées est présenté, ainsi que l'implémentation de leur ressources. Les mécanismes de reconfiguration et de sauvegarde/restauration de contexte sont détaillés. Ensuite, la faisabilité de la synthèse d'overlays sur FPGA

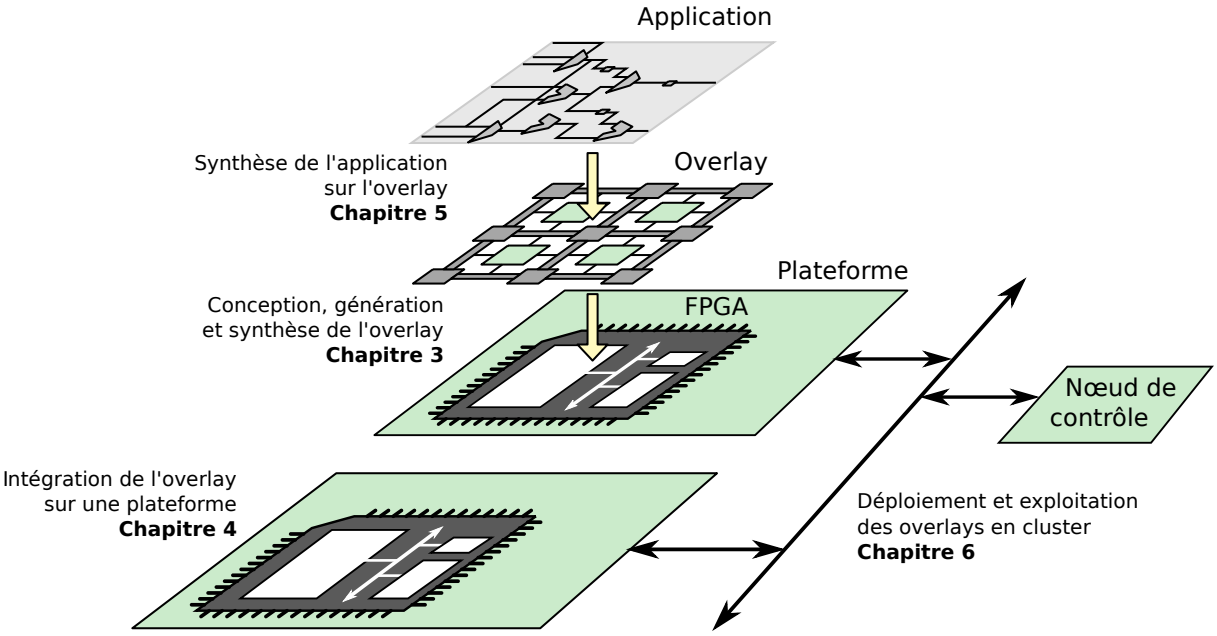


FIGURE 2.1 – Du prototypage à l'exploitation des overlays : vue des chapitres

est abordée. Deux méthodes sont proposées pour la génération de la description RTL des overlays, chacune ayant ses avantages. Finalement, le coût d'un des overlays réalisés est évalué puis modélisé suivant ses paramètres architecturaux.

Le chapitre 4 traite de l'intégration des overlays sur des plateformes FPGA commerciales. Afin de supporter la variabilité des overlays, ceux-ci sont d'abord intégrés dans une IP. En plus de l'overlay, cette IP contient aussi les contrôleurs nécessaires à la gestion bas niveau de l'overlay. Cette IP est ensuite intégrée à un système capable de le contrôler localement. Cette gestion locale étant réalisée de manière logicielle, un processeur est nécessaire sur la plateforme. Suivant les composants offerts par celles-ci, différentes méthodes d'implémentation sont présentées afin de faire des plateformes des nœuds de calcul intégrables à un réseau informatique. Différents systèmes d'intégration sont réalisés sur des plateformes FPGA commerciales présentant différentes caractéristiques, et le coût des composants de ces systèmes sont évalués.

Le chapitre 5 aborde la mise en place d'un flot permettant la synthèse d'applications sur overlay. Différents outils existent qui permettent la réalisation des différentes étapes de synthèse. Un flot modulaire est assemblé, qui permet d'aboutir à un binaire de configuration ciblant un overlay donné. Le problème de l'analyse de timing sur overlay est analysé, et une solution proposée. Finalement, le flot est mis en œuvre pour évaluer sur un ensemble d'applications le surcoût de l'overlay par rapport à une implémentation native de ces applications sur le FPGA.

Le chapitre 6 s'intéresse aux mécanismes de gestion des overlays dans un contexte Cloud. Au niveau de chaque nœud de calcul, un ensemble d'applications est ordonnancé localement sur l'overlay en exploitant les mécanismes de changement de contextes implémentés par les overlays. Au niveau du cluster, un nœud de contrôle

communique avec les différents nœuds de calcul afin de répartir les applications sur le cluster de façon à optimiser sont utilisation. Les mécanismes de migration à chaud d'applications d'un nœud à un autre sont présentés, et les temps impliqués dans l'ordonnancement sur overlays sont modélisés. La figure 2.1 illustre schématiquement la place de chaque chapitre dans la mise en œuvre des overlays.

Le dernier chapitre effectue la synthèse des points principaux de notre contribution et dresse les perspectives de recherche ouvertes par ces travaux.

Chapitre 3

Architecture virtuelle : conception, faisabilité et modèle de coût

Un overlay est une architecture reconfigurable, qui peut donc être configuré pour implémenter des circuits applicatifs. Cependant, un overlay a la particularité d'être implémenté sur une autre architecture reconfigurable (physique) telle qu'un FPGA, et non pas directement sur silicium. Ainsi, contrairement à un FPGA qui est un circuit directement disponible dans le commerce, un overlay doit être conçu et implémenté sur FPGA avant de pouvoir être utilisé. Ce chapitre traite de la conception d'architectures overlay. Pour répondre à la problématique principale de ce travail, qui est de permettre l'exploitation de FPGA en tant qu'accélérateurs reconfigurables dans un cadre Cloud via l'utilisation d'overlays, les overlays implémenter des mécanismes permettant à un système d'exploitation de partager leur ressources entre différentes applications, et notamment permettre la préemption des circuits applicatifs et une reconfiguration rapide ; Aussi, pour permettre l'exploration de l'espace de conception des overlays, ceux-ci doivent être synthétisables sur FPGA, quelle que soit la taille de l'overlay, et la génération et la synthèse de leurs descriptions RTL doit pouvoir être automatisée, sans intervention manuelle telles qu'une étape de floorplaning.

Du point de vue de l'hôte (le FPGA physique), l'overlay est lui-même un circuit applicatif. Un overlay présente donc deux facettes suivant de quel angle on le considère : soit comme une architecture reconfigurable (facette *architecture*), soit comme un circuit classique, i.e. une IP (facette *implémentation*). La figure 3.1 illustre ces deux facettes de l'overlay.

La facette architecture correspond à l'aspect fonctionnel de l'overlay, et considère l'ensemble des ressources reconfigurables de l'overlay qui sont manipulables par l'outil de synthèse virtuelle et qui permettent d'implémenter les circuits applicatifs. Il s'agit des canaux de routage, des matrices d'interconnexion et des unités fonctionnelles. Nous appelons *plan de calcul* de l'overlay l'ensemble de ces ressources et leur agencement, c'est-à-dire l'architecture de l'overlay visible par l'outil de synthèse virtuelle et les applications. La facette implémentation correspond à

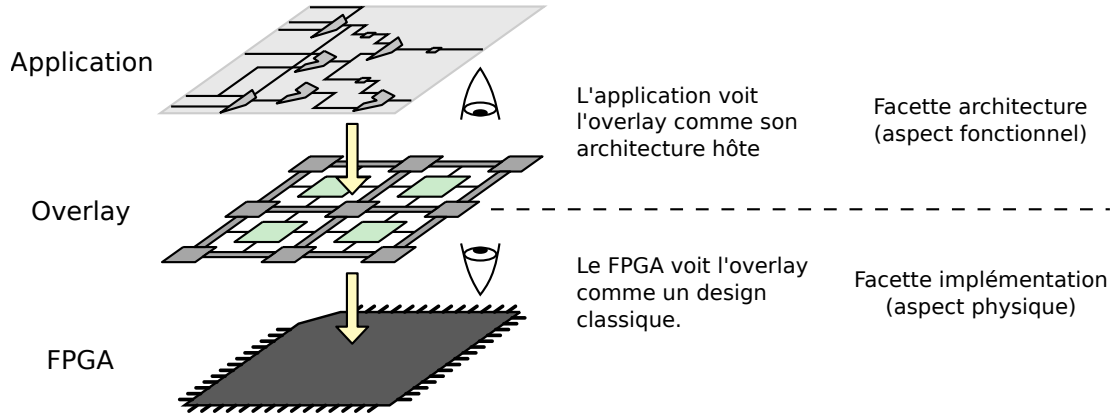


FIGURE 3.1 – Les deux aspects d'un overlay

l'aspect physique de l'overlay, c'est-à-dire sa réalisation sur un FPGA. Par exemple, un plan de calcul peut contenir des LUTs (aspect fonctionnel), tandis que la description de ces LUTs peut reposer sur des registres et des multiplexeurs (aspect physique).

Deux métriques permettent d'évaluer la qualité d'une architecture reconfigurable par rapport à son aspect fonctionnel :

La capacité logique : c'est le nombre d'unités fonctionnelles disponibles dans l'architecture ainsi que leur nature. Par exemple, la capacité logique des FPGA du commerce est évaluée en nombre de LUT à K entrées, ou est estimée en équivalent de nombre portes logiques. C'est une métrique que l'on peut quantifier à partir des spécifications du plan de calcul l'architecture.

La routabilité : c'est la flexibilité des ressources de routage du plan de calcul de l'architecture. Elle est le reflet de la largeur des canaux de routages et du nombre d'interconnexions possibles entre les piste des canaux de routage. Plus une architecture est routable, moins cela demande d'effort à l'outil de synthèse virtuelle pour synthétiser un circuit applicatif sur l'architecture. Entre deux architectures de capacité logique équivalente, plus de circuits applicatifs pourront être synthétisés sur l'architecture la plus routable. De plus la routabilité d'une architecture reconfigurable impacte la fréquence maximale de fonctionnement des applications implémentées : une bonne routabilité implique moins de congestion lors du routage, des chemins et donc des délais de routage plus courts. Pour différentes architectures de capacités logiques équivalentes, le taux de routabilité peut être quantifiée expérimentalement et exprimé par le nombre de circuits applicatifs d'un ensemble de benchmarks synthétisables pour chacune des architectures. Ce résultat expérimental dépend cependant aussi des performances de l'outil de synthèse et de son adéquation avec les architectures ciblées.

De même, deux métriques permettent d'évaluer la qualité d'un overlay par rapport à son aspect physique :

Le (sur)coût en surface : c'est la surface physique, i.e. la quantité de ressource du FPGA hôte qui est nécessaire pour implémenter l'overlay. Cette métrique

peut être évaluée expérimentalement en synthétisant l'overlay sur un FPGA. Le surcout en surface peut être évalué expérimentalement en synthétisant une même application nativement sur un FPGA, et sur un overlay ayant juste la taille nécessaire pour l'implémenter. Le surcout en surface est alors le rapport entre la surface occupée par l'overlay et la surface occupée par l'application native.

Le (sur)coût en fréquence : il s'agit des délais des ressources physiques implémentant le plan de calcul de l'overlay. Le surcout en fréquence peut être évalué expérimentalement en synthétisant une même application nativement sur un FPGA, et sur un overlay. Le surcout en fréquence est alors le rapport entre la fréquence maximale de fonctionnement de l'application sur l'overlay et la fréquence de fonctionnement de l'application implémentée directement sur le FPGA.

Ces deux axes fonctionnel et physique des overlays sont orthogonaux. Cependant, le plan de calcul a un impact sur les métriques physiques de l'overlay : par exemple, améliorer la routabilité du plan de calcul peut se faire en augmentant le nombre de ressources de routage et leurs interconnexions possibles, ce qui augmente la surface occupée par l'overlay sur son hôte. Ainsi, il est nécessaire de prendre ces deux axes en compte pour apprécier la qualité globale d'un overlay. La figure 3.2 illustre la zone de qualité d'un overlay suivant ces axes.

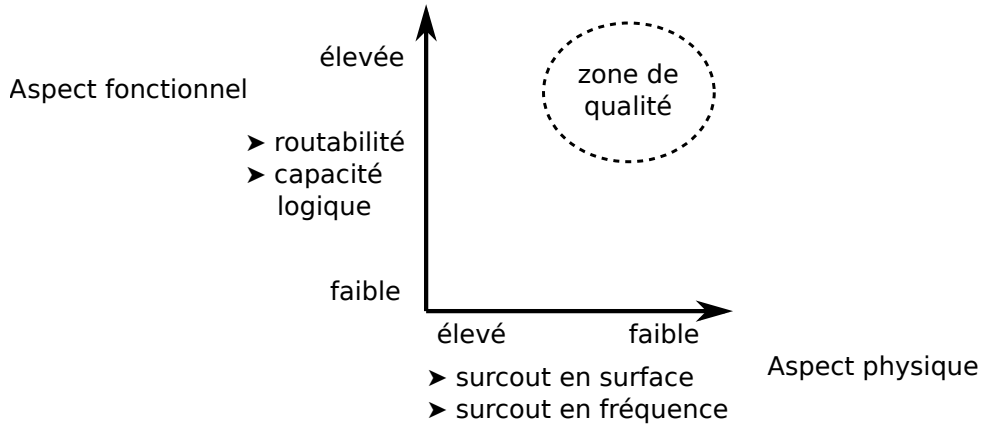


FIGURE 3.2 – Évaluation d'un overlay

La conception d'un overlay consiste donc en la définition d'un plan de calcul, puis son implémentation. Les ressources du plan de calcul sont dimensionnées selon le compromis qualité fonctionnelle / coût physique, et l'implémentation physique est raffinée et optimisée afin d'augmenter la qualité de l'overlay produit. La définition et le dimensionnement du plan de calcul demande de réaliser une exploration suivant les deux axes physique et fonctionnel. La figure 3.3 montre les deux boucles itératives mis en œuvre lors de cette exploration. Pour pouvoir les effectuer, il est nécessaire :

— de générer la description RTL de l'overlay, la synthétiser sur FPGA, puis évaluer et éventuellement modéliser son coût en ressources. Ces trois points seront abordés dans ce chapitre.

— d'avoir un outillage de synthèses d'applications permettant de cibler le plan de calcul réalisé tout en s'adaptant à des changements de paramètres de celui-ci. Cet outillage fait l'objet du chapitre 5.
— d'intégrer l'overlay dans un système qui fournisse un accès à celui-ci de l'extérieur du FPGA, pour pouvoir vérifier son fonctionnement et l'utiliser. L'intégration des overlays est traitée dans le chapitre 4.

La définition du plan de calcul est présentée dans la section suivante.

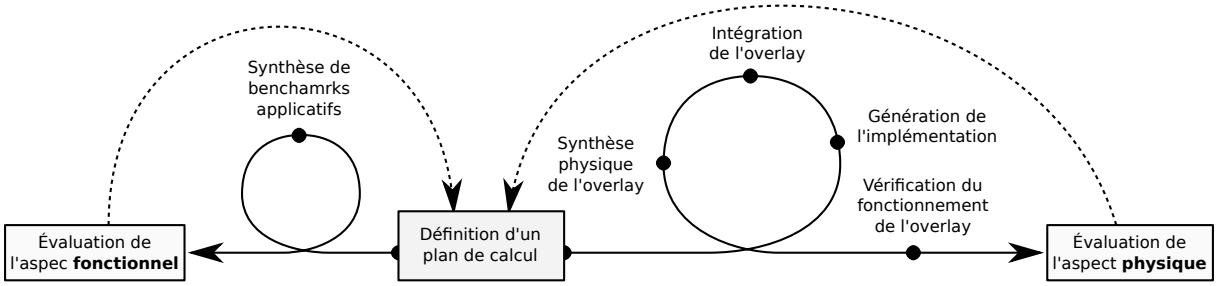


FIGURE 3.3 – Les itérations lors de l'exploration de l'espace de conception d'un overlay

3.1 Architectures fonctionnelle : plan de calcul

Le plan de calcul est l'ensemble des ressources reconfigurables de l'overlay que l'outil de synthèse virtuelle peut manipuler pour implémenter les circuits applicatifs. Nous le distinguons du plan de configuration, qui stocke la configuration du circuit applicatif et fournis au plan de calcul les signaux de configuration de ses ressources reconfigurables. Nous le distinguons aussi d'un troisième plan, le plan de snapshot, que nous verrons plus loin (cf 5.2.3), qui permet le chargement et l'extraction des valeurs des éléments séquentiels du plan de calcul. L'architecture du plan de calcul correspond à l'agencement des ressources logiques (LUTs), séquentielles (flip-flops) et de routage (canaux de routage et Switch Box), ainsi qu'à leur types, comme le nombre d'entrées par LUT ou les possibilités d'interconnexion entre les canaux de routage par les Switch Box.

Dans ces travaux, deux modèles d'architectures de plan de calcul ont été réalisés, que nous avons nommé vFPGA-restreint et vFPGA-flexible. Pour permettre une exploration rapide, le modèle vFPGA-restreint a un nombre restreint de paramètres : la dimension de la matrice, le nombre d'entrées par LUT et le nombre de pistes par canal de routage. Le modèle vFPGA-flexible quant à lui est beaucoup plus flexible, et permet d'explorer un espace de conception plus large.

3.1.1 Architecture vFPGA-restreint : un modèle restreint pour une exploration rapide

L'architecture de ce premier overlay grain fin est de type îlots de calculs, c'est-à-dire que l'on y trouve des unités fonctionnelles reliées par une matrice d'interconnexion. La figure 3.4 montre une telle architecture : les unités fonctionnelles sont représentées par les blocs gris clairs tandis que la matrice d'interconnexion est représentée en noir.

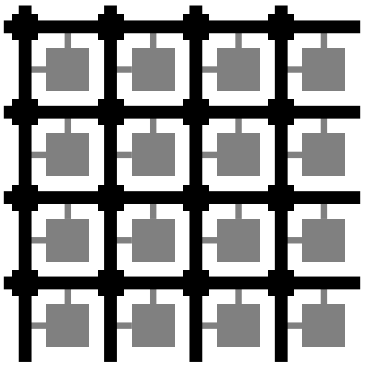


FIGURE 3.4 – Matrice reconfigurable de type îlots de calculs

Les unités fonctionnelles sont reconfigurables, c'est-à-dire que le traitement réalisé sur leurs entrées dépend d'une configuration qui peut être chargée et rechargée sans modification de l'architecture. De même les blocs de connexion reliant les fonctions logiques à la matrice d'interconnexion sont aussi reconfigurables et permettent de choisir vers quelles pistes de routage sont dirigées les sorties des unités fonctionnelles et de quelles pistes sont extraites leurs entrées.

La matrice d'interconnexion est un ensemble de pistes rassemblées en canaux de routage verticaux et horizontaux. Des switchs reconfigurables (représentés par des carrés noirs sur la figure 3.4) présents aux intersections de ses canaux permettent d'aiguiller les signaux d'une piste d'un canal vers une piste d'un autre canal.

La figure 3.4 montre que l'architecture de la matrice reconfigurable est la répétition d'un même motif sur deux dimensions. Nous appelons ce motif une tuile. Il se compose d'une fonction logique et ces blocs de connexions associés, ainsi qu'un switch, un canal de routage horizontale et un vertical. Le détail de l'architecture d'une tuile est présenté figure 3.5. On y retrouve le switch, la fonction logique (nommée func), les blocs de connexion horizontaux et verticaux (cbh et cbv). Une tuile comporte donc une fonction logique ainsi qu'une portion de la matrice d'interconnexion.

La fonction logique est réalisée par une look-up-table (LUT) qui n'est autre qu'une mémoire stockant 2^K mots de 1 bit, K étant le nombre de bits adressant cette mémoire et correspondant aux entrées de la LUT. Une telle LUT peut ainsi implémenter n'importe quelle fonction logique comportant au plus K entrées et

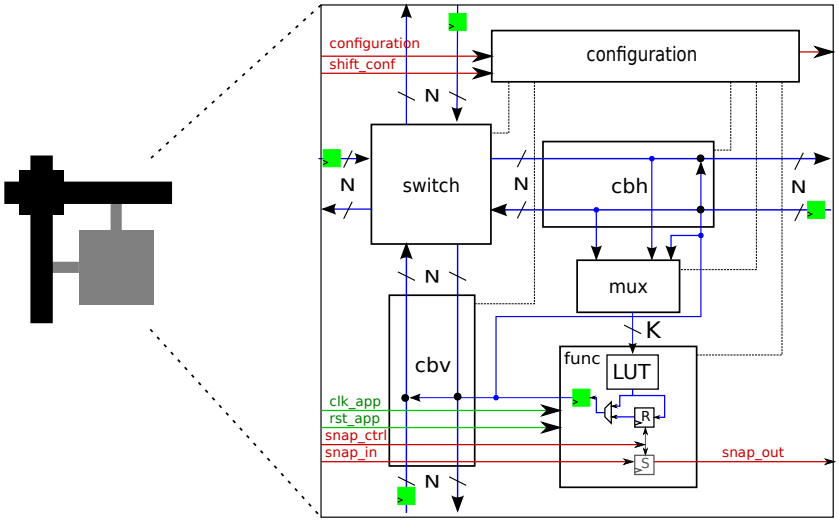


FIGURE 3.5 – Architecture d'une tuile

une sortie. Un multiplexeur permet de choisir si la sortie de la LUT est tamponnée dans un registre applicatif ou non, ce qui autorise l'implémentation de circuits combinatoires aussi bien que séquentiels sur cette architecture.

La configuration de chaque tuile est stockée dans un registre représenté en haut de la figure 3.5. La valeur de chacun des champs de ce registre vient paramétrer un des éléments de la tuile. Les registres de configuration de toutes les tuiles de l'overlay sont chaînés les uns à la suite des autres de façon à former un unique registre à décalage traversant l'ensemble des tuiles, est dont la concaténation de tous les bits forme le bitstream virtuel de l'overlay, c'est-à-dire sa configuration.

3.1.2 Architecture vFPGA-flexible : un modèle plus flexible pour un espace de conception plus large

Le plan de calcul de l'overlay vFPGA-flexible est aussi une architecture de type îlots de calcul (figure 3.6). Contrairement au vFPGA-restreint, cette architecture comporte un niveau de hiérarchie supplémentaire entre la matrice de routage et les unités fonctionnelles de base : il s'agit des CLBs (Configurable Logic Blocs). Les CLBs on chacun I entrées et N sorties, ils sont composés de N BLEs (Basic Logic Element). Un BLE a une LUT de K entrées et un registre connecté à la sortie de la LUT. Selon la configuration, la sortie du BLE est soit la sortie de la LUT, soit la sortie du registre. Dans le CLB, les entrées des BLEs sont issues d'un crossbar de $I + N$ entrées : les I entrées du CLB et les N sorties des BLEs, ce qui permet de chaîner les BLEs d'un même CLB sans passer par le routage externe. L'overlay est composé d'une matrice de $Largeur \times Hauteur$ CLBs.

Comme le routage bidirectionnel des pistes de routage n'est pas adapté à une implémentation sur FPGA (émulation de signaux à trois états), le plan de calcul utilise plutôt des pistes de routage unidirectionnelles. Les W pistes unidirection-

nelles de chaque canal de routage sont connectables aux autres pistes des canaux de routage adjacents de manière configurable dans les Switch Box (SB). Chaque élément de cette architecture est piloté par un multiplexeur dont les signaux de sélection sont issus du registre de configuration.

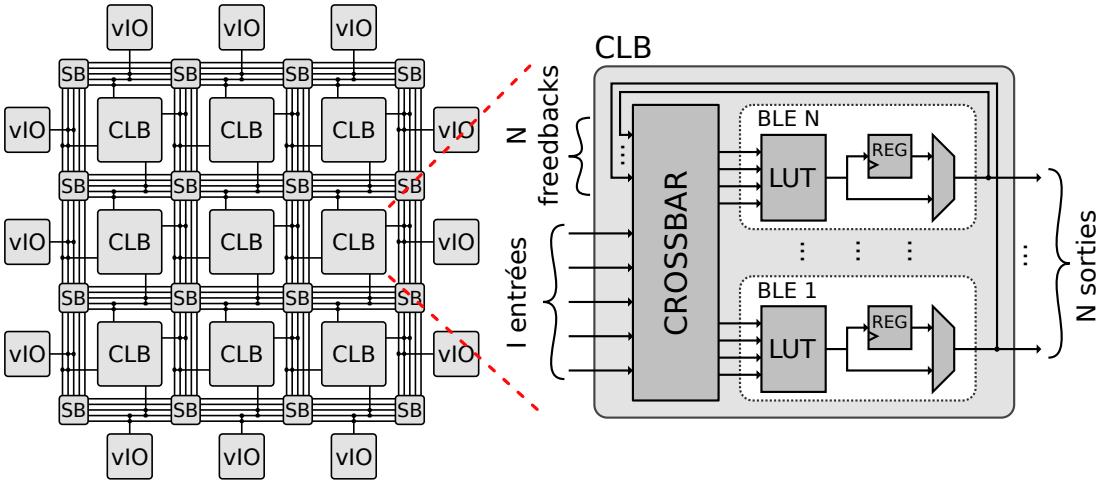


FIGURE 3.6 – Illustration du plan de calcul du vFPGA-flexible pour une matrice de 3×3 CLBs

3.2 Implémentation et instrumentation des overlays

Une fois l'architecture fonctionnelle déterminée (type de ressources, agencement, etc), l'overlay peut être implémenté. Implémenter l'overlay correspond à produire sa description RTL pour pouvoir ensuite le synthétiser. L'implémentation d'un overlay présente deux aspects :

L'implémentation du plan de calcul : Il s'agit de la forme RTL sous laquelle est implémentée chaque ressource atomique. Par exemple, le crossbar des CLB peut être implémenté à l'aide de multiplexeurs génériques ou par un réseau de Clos utilisant les LUT physiques reconfigurables dynamiquement du FPGA hôte comme briques de base [ZUMA].

L'instrumentation de l'overlay : Il s'agit de ressources additionnelles qui viennent instrumenter l'architecture fonctionnelle. Elles ne sont pas visibles dans le plan de calcul de l'overlay (et sont donc transparentes pour les circuits applicatifs), mais sont nécessaires au fonctionnement de l'overlay et à l'ajout de fonctionnalités.

La première instrumentation de l'overlay est ce que nous appelons le *plan de configuration*, il s'agit de la partie de l'overlay chargée de stocker et fournir au plan de calcul chaque bit de configuration de ce dernier. Une architecture reconfigurable classique telle qu'un FPGA peut être décomposée conceptuellement en un plan de calcul et un plan de configuration. Dans ces travaux, un troisième plan a été ajouté

au overlays : le *plan de snapshot*. Le plan de snapshot est une instrumentation de l'overlay qui permet le chargement est l'extraction du contenu des registres applicatifs du plan de calcul, et donne ainsi accès à l'état d'exécution de l'application sur le plan de calcul. La figure 3.7 représente les trois plans de l'implémentation d'un overlay.

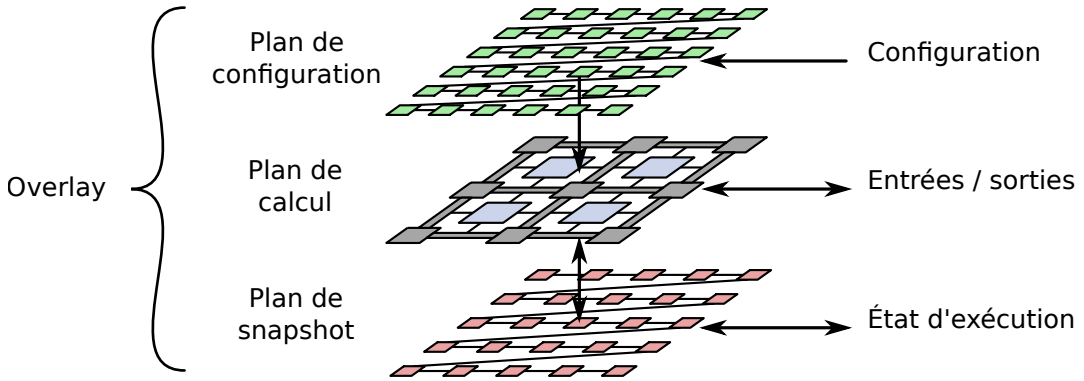


FIGURE 3.7 – Décomposition de l'implémentation d'un overlay en trois plans conceptuels

Les compromis tels que la portabilité de l'overlay versus son surcout en surface peuvent être adressés au niveau de l'implémentation de l'overlay. Cependant, les optimisations les plus poussées reposant sur des spécificités de chaque FPGA hôte et des outils de synthèse physique qui leur sont associés, elles compliquent le travail de portage de l'overlay. Par exemple, dans l'overlay ZUMA, les LUT-RAMs à 6 entrées sont utilisées comme brique de base de l'implémentation. Une LUT-RAM est une LUT du FPGA physique pour laquelle le circuit synthétisé sur le FPGA peut lui-même modifier le contenu. Les LUTs ainsi que les multiplexeurs configurables du plan de calcul sont implémentés par ces LUT-RAMs, et les crossbars sont implémentés en réseaux de Clos mettant en œuvre ces LUT-RAMs. L'utilisation de LUT-RAM comme brique de base permet des économies de surface, car la configuration est stockée directement dans la LUT physique l'implémentant, et non dans des flip-flop configurables.

Lors de la conception agile d'un overlay, dans un premier temps, il est préférable de réaliser une implémentation générique de l'overlay. Une fois le premier cycle de développement réalisé, c'est-à-dire que l'overlay a été intégré, synthétisé, et que son fonctionnement a été vérifié avec des applications sur FPGA, il est alors possible de s'intéresser aux optimisations possibles de son implémentation, et de raffiner celle-ci par itérations. Dans ces travaux, nous avons voulu rester générique, et nous n'avons donc pas cherché à optimiser l'implémentation de nos prototypes d'overlay au-delà de ce qu'il est possible de faire avec du VHDL portable, sans exploiter les spécificités d'un FPGA en particulier ni en reposant sur un outil de synthèse donné. Cependant nos travaux n'empêchent pas ce type d'optimisations, et nos prototypes d'overlay peuvent très bien bénéficier d'optimisations telles que [Zuma] et [implém du routage V dans le phy]. Cette section présente l'implémentation des ressources atomiques du plan de calcul et de son instrumentation.

3.2.1 Implémentation des ressources atomiques

Les différentes ressources atomiques que l'on peut trouver dans le plan de calcul d'un overlay sont des pistes de routage, des switch boxes et des crossbars. Dans les FPGAs du commerce, la logique à trois états ne peut être réalisée que sur les entrées/sorties du FPGA. La logique à trois états interne présente dans les applications doit être émulée lorsqu'elle est implémentée sur FPGA. Du fait que les overlays sont implémentés sur FPGA, il est donc préférable de ne pas inclure d'éléments logiques à trois états dans le plan de calcul de l'overlay, de façon à simplifier son implémentation et à la rendre plus efficace. Ne pas utiliser de logique à trois états dans le plan de calcul de l'overlay implique que tous les éléments sont unidirectionnels. Quelle que soit la granularité de l'overlay, les pistes de routage peuvent alors être implémentées par de simples signaux tandis que les interconnexions configurables présentes dans les SBs et les crossbars qui pilotent ces pistes peuvent être implémentés par des multiplexeurs, dont les signaux de sélection sont issus du plan de configuration. La figure 3.8 illustre l'implémentation par des multiplexeurs d'un SB de topologie Wilton avec quatre pistes par canal de routage.

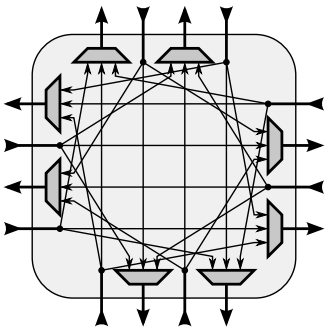


FIGURE 3.8 – Implémentation d'une Switch Box par des multiplexeurs

Dans le cadre des overlays grain fin, les LUTs peuvent aussi être implémentées par des multiplexeurs. Par exemple, une LUT à trois entrées peut être implémentée par un multiplexeur 8/1 dont les huit signaux d'entrées (la table de valeur) sont issu du plan de configuration, et les trois signaux de sélection sont les trois entrées de la LUT.

Les opérateurs plus gros grain que l'on trouve dans les overlays gros grain ou sous forme de macro-blocks dans les overlays grains fin, tels que des blocs DSP, peuvent êtres implémentés en tirant directement partit des macro-blocks présents dans le FPGA sous-jacent. Par exemple, un bloc multiplieur $32 \times 32 \rightarrow 64$ du plan de calcul peut être implémenté par quatre multiplieurs physiques $16 \times 16 \rightarrow 32$. Dans le cas d'opérateurs plus complexes, ceux-ci peuvent être implémentés de manière "soft", c'est-à-dire en utilisant les ressources (LUTs et flip-flops) du FPGA.

L'implémentation des registres applicatifs est liée à la gestion de l'horloge applicative, est sera traitée dans la sous-section 3.2.3. L'implémentation des mémoires virtuelles (l'équivalent des BLOCK-RAMs des FPGAs commerciaux) sera traitée dans la sous-section 3.2.5.

3.2.2 Plan de configuration et pré-configuration

Plan de configuration

Le *plan de calcul* de l'overlay est l'ensemble des ressources qui implémentent les circuits applicatifs et qui permettent de réaliser les fonctions logiques des circuits applicatifs synthétisés sur l'overlay. Le *plan de configuration* est l'ensemble des registres qui reçoivent et stockent la configuration de l'overlay. Chaque ressource du plan de calcul est conditionnée par les registres du plan de configuration qui lui sont associés. Ces registres de configuration sont chaînés en un registre à décalage, similaire au Boundary Scan Register du JTAG. Le chargement de la configuration de l'overlay se fait en poussant bit par bit le bitstream virtuel dans ce registre à décalage. Une fois que l'intégralité du bitstream virtuel a été chargé, chaque bit de celui-ci occupe le registre du plan de configuration qui conditionne la ressource du plan de calcul que ce bit est censé contrôler. Le contrôleur de configuration est le module responsable de charger le bitstream virtuel dans le plan de configuration. C'est un module extérieur à l'overlay, qui peut être implémenté de différentes manières : en lisant le bitstream virtuel depuis un mémoire interne du FPGA, ou une mémoire externe à l'aide d'une DMA, ou encore en poussant les bits écrit dans un registre par un processeur.

Temps de configuration

Pendant la configuration de l'overlay, le contenu du plan de configuration est modifié à chaque fois qu'une portion du bitstream virtuel est poussée dans le registre à décalage. Ainsi, le plan de configuration est reconfiguré par une configuration différente à chaque bit de configuration poussé, mais le plan de calcul n'est configuré de manière sensée qu'après la complétion du chargement. Le plan de calcul n'est donc pas opérationnel pendant le laps de temps que dure la configuration. Le temps de configuration a donc un impact sur les performances d'un système reconfigurable lorsque le système est fréquemment reconfiguré : plus le temps de reconfiguration est long et plus le temps entre deux reconfigurations est court, moins le système passe de temps à exécuter les circuits applicatifs. Pour diminuer le temps d'inactivité du plan de calcul lors de l'exploitation d'un overlay, deux solutions sont possibles au niveau de l'implémentation du plan de configuration : la première consiste à diminuer le temps de chargement, la deuxième permet de pousser une configuration sans impacter le plan de calcul, et donc d'annuler l'effet du temps de configuration sur le taux d'utilisation du plan de calcul.

Première optimisation : diminution du temps de chargement

Pour augmenter la vitesse de chargement du bitstream virtuel dans l'overlay, le plan de configuration est divisé en plusieurs chaînes de registre à décalage de manière à pouvoir pousser plusieurs bit de configuration par cycle d'horloge. Par

exemple, diviser le plan de configuration en 32 chaînes de registres à décalages permet de charger le bitstream virtuel 32 bits par 32 bits, et donc d'avoir un temps minimum de chargement 32 fois plus court que si tous les registres de configuration étaient répartis en une unique chaîne. Cependant, bien que la largeur de l'interface de configuration de l'overlay puisse être aussi large que désiré (jusqu'à faire la largeur du registre de configuration pour un temps de reconfiguration d'un seul cycle d'horloge), l'utilité d'élargir l'interface de configuration est limitée par le contrôleur de configuration. En effet, celui-ci ne peut pousser qu'un nombre limité de bits par cycle d'horloge, selon la largeur de données de la mémoire ou du registre depuis laquelle ou lequel il lit le bitstream virtuel. C'est-à-dire qu'au-delà d'une certaine largeur de configuration, le goulot d'étranglement n'est plus dans l'overlay mais en amont.

Deuxième optimisation : la pré-configuration

La pré-configuration permet de charger une configuration dans le plan de configuration sans perturber le plan de calcul. La pré-configuration consiste à doubler les registres de configuration de l'overlay, de manière à obtenir deux niveaux dans le plan de configuration, comme illustré figure 3.9. Le premier niveau est formé d'une ou plusieurs chaînes de registres à décalage comme décrit dans les paragraphes précédents. Plutôt que de contrôler directement le plan de calcul, chaque registre de ce premier niveau est appairé avec un registre du deuxième niveau du plan de configuration, qui lui est connecté au plan de calcul. Tous les registres du deuxième niveau partagent un même signal "enable" qui permet de copier le contenu du premier niveau dans le deuxième niveau en un cycle d'horloge. Il est ainsi possible de charger le bitstream virtuel dans le plan de configuration sans perturber le plan de calcul, et de changer la configuration effective du plan de calcul en un seul cycle d'horloge. La pré-configuration de l'overlay permet donc d'annuler le coût du temps de configuration

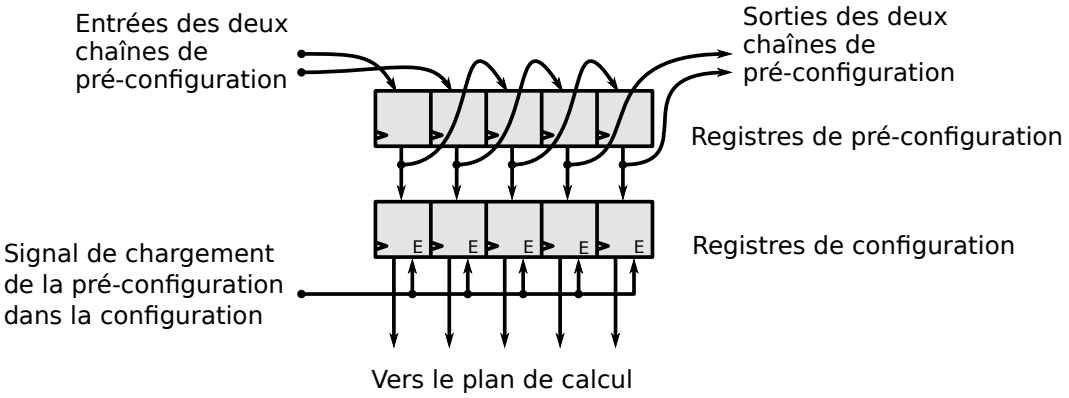


FIGURE 3.9 – Illustration du mécanisme de pré-configuration, avec deux chaînes de pré-configuration

Il est à noter que le mécanisme de pré-configuration est différent du multi-contextes : alors qu'avec la pré-configuration les registres de configuration sont

doublés, le multi-contextes multiplie le nombre de registres nécessaires dans le plan de configuration par le nombre de contextes désiré. La configuration effective du plan de calcul est issue de multiplexeurs qui permettent de passer instantanément d'un contextes à un autre. Contrairement à la pré-configuration, le multi-contextes permet donc de repasser à une configuration précédemment active sans avoir à la re-charger dans le plan de configuration, cependant implémenter un plan de configuration multi-contextes est plus coûteux en ressources car cela demande plus de registres et de chemins de configuration.

En doublant chacun des registres de configuration, le mécanisme de pré-configuration amène un surcoût en ressources du FPGA hôte par rapport à un overlay avec un plan de configuration simple. Ce surcoût sera évalué en 3.5.1, il s'étend de 20% à 30% en LUTs et de 60% à 80% en flip-flops suivant le FPGA hôte.

3.2.3 Implémentation des registres applicatifs et horloge applicative

Les registres applicatifs sont les registres apparaissant dans le plan de calcul de l'overlay, manipulables par l'outil de synthèse virtuelle et qui servent à implémenter les registres apparaissant dans la netlist du circuit applicatif. Dans le cas des plans de calculs des vFPGA-restreint et vFPGA-flexible présentés en 3.1, il s'agit des registres présent à la sortie des LUTs et qui sont être contourné ou non selon la configuration du multiplexeur sélectionnant parmi la sortie de la LUT ou du registre. La présence de registres applicatifs implique la présence d'une ou plusieurs horloges applicatives.

Nous verrons au chapitre 5 que pour un overlay donné, le chemin critique de circuits applicatifs synthétisés sur un plan de calcul varie d'un circuit applicatif à un autre. Lors de la synthèse virtuelle (synthèse d'un circuit applicatif sur l'overlay), l'outil analyse le plus grand délai parmi tous les chemins applicatifs entre deux registres applicatifs, et en déduit la fréquence maximale de fonctionnement du circuit applicatif sur l'overlay, qu'il indique dans son rapport de synthèse. Il est donc nécessaire de pouvoir adapter la fréquence de l'horloge applicative (la fréquence à laquelle les registres applicatifs réalisent leur transfert D $\rightarrow$ Q) à chaque circuit applicatif. La fréquence de l'horloge applicative ne peut donc pas être fixée lors de la synthèse de l'overlay sur FPGA.

Une solution pour réaliser un signal d'horloge à fréquence variable dans un FPGA est d'instancier un générateur d'horloge (une PLL qui est implémentée "en dur" dans le silicium). Par exemple, les blocs PLL des FPGA Xilinx offrent une interface de reconfiguration partielle dynamique (DPR) sous forme d'un ensemble de registres, permettant de modifier dynamiquement les caractéristiques du signal d'horloge généré, et ce après configuration du FPGA [1]. Cette solution simplifie l'implémentation du plan de calcul de l'overlay, mais complique le portage d'un overlay

1. http ://www.xilinx.com/support/documentation/application_notes/xapp888_7Series_DynamicRecon.pdf

d'un FPGA à un autre. En effet les IP PLL sont spécifiques à certaines familles de FPGA et par vendeur : en plus de la connectique de l'interface qui peut changer d'une IP PLL à une autre, la signification des champs de bit des registres de configurations des PLL varient aussi d'un modèle à un autre.

De plus nous verrons dans la section 3.3 que pour faciliter la synthèse physique de l'overlay, nous avons placé des registres additionnels dans les ressources de routage du plan de calcul de l'overlay. Ces registres sont nécessairement cadencés à une fréquence plus rapide que celle de l'horloge applicative, mais leur horloge doit tout de même être cohérente avec celle-ci. Il est de même préférable que les horloges internes de l'overlay soient cohérentes avec l'horloge externe de l'overlay, de manière à éviter de compliquer l'interface de l'overlay par des mécanismes complexes de synchronisation entre différents domaines d'horloges. La solution que nous avons retenue pour pouvoir gérer dynamiquement l'horloge applicative est de garder tous les signaux de l'overlay dans le même domaine d'horloge. Chaque registre physique du FPGA servant à implémenter l'overlay est donc cadencé par une horloge empruntant un arbre de distribution d'horloge physique du FPGA, ce qui permet à l'outil de synthèse physique de mieux gérer les timings de l'overlay. L'horloge applicative est quant à elle un signal "classique" généré par un compteur décrit en RTL, qui n'est actif qu'un cycle d'horloge sur N, N étant modifiable dynamiquement. Le signal d'horloge applicative est connecté à l'entrée *ENABLE* des registres applicatifs, le transfert D $\rightarrow$ Q des registres applicatifs ne se fait donc que sur front montant de l'horloge physique et lorsque le signal d'horloge applicative est actif. La figure 3.10 illustre la génération du signal d'horloge applicative et l'implémentation des registres applicatifs.

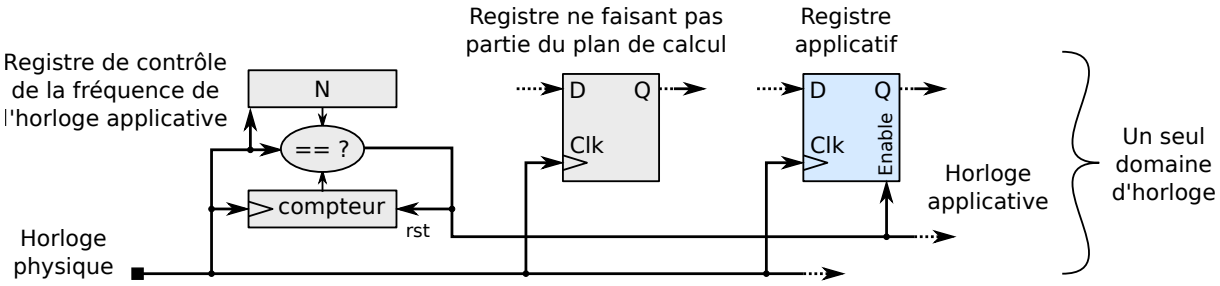


FIGURE 3.10 – Implémentation des registres applicatif et de l'horloge applicative.

Ce mécanisme de génération de l'horloge applicative est portable et simplifie l'implémentation de l'overlay tout en assurant que chaque registre soit cadencé par un signal d'horloge issu d'un arbre de distribution d'horloge physique du FPGA. De plus ce mécanisme est plus flexible que l'utilisation d'une PLL : il est en effet possible de changer instantanément la fréquence de l'horloge applicative, de la stopper, la lancer, ou encore de la lancer pour un nombre de cycles applicatifs déterminé. Retarder le prochain cycle d'horloge applicative permet de geler le plan de calcul, ce qui peut être utile dans l'implémentation de mémoires virtuelles (voir 3.2.5).

3.2.4 Plan de snapshot, accès au contexte d'exécution

Comme nous le verrons dans le chapitre 6, le partage temporel d'une ressource entre plusieurs applications permet une meilleure utilisation de celle-ci et permet une plus grande flexibilité sur la gestion de celle-ci. Par exemple, dans un ordinateur, le système d'exploitation (OS) partage dans le temps la ressource processeur entre différents processus. La figure 3.11 illustre le diagramme de transition d'états d'un processus lors de l'ordonnancement par l'OS. Les processus prêts sont les processus qui sont en état d'être exécutés sur le processeur. Le processus élut est le processus qui a été choisi par l'OS parmi les processus prêts pour être exécuté, et donc bénéficier de la ressource processeur. Lorsqu'un processus en cours d'exécution (élu) demande une ressource qui n'est pas encore disponible (par exemple lire une donnée qui n'est pas encore accessible), l'OS stoppe ce processus, le place dans la liste des processus bloqués et élit un nouveau processus parmi ceux qui sont prêts pour qu'il soit exécuté. Lorsque la ressource demandée par le processus qui a été bloquée est enfin disponible, l'OS replace le processus dans la liste des processus prêts. Ainsi, le processeur n'est jamais utilisé par des processus inactifs car en attente d'une ressource indisponible, ce qui augmente le taux d'utilisation effectif du processeur. Aussi, si un processus est élu pendant plus d'un quantum de temps sans avoir été bloqué, l'OS le replace la liste des processus prêts pour en élire un autre (préemption). Ceci assure le partage du processeur entre tous les processus sur une échelle de temps donné.

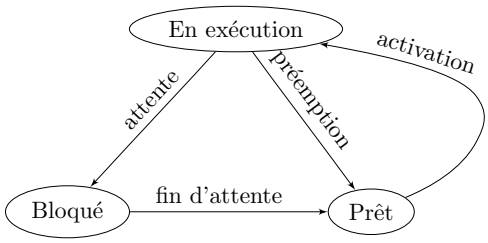


FIGURE 3.11 – Diagramme de transition d'états d'un processus

Une gestion de type OS assurerait aux architectures reconfigurables les mêmes avantages que pour les processeurs (un partage flexible et une amélioration du taux d'utilisation) en permettant :

- qu'un circuit applicatif n'occupe pas inutilement l'architecture lorsqu'il est en attentes de données ;
- de pouvoir exécuter différents circuits sur une même architecture à une échelle de temps donnée ;
- de donner à l'architecture reconfigurable une flexibilité temporelle similaire à celle des processeurs. En effet, lorsqu'une architecture reconfigurable est occupé à 100% par un ensemble de circuits, elle est saturée spatialement et ne peut pas exécuter un circuit de plus. Par contre si les différents circuits peuvent être ordonnancés sur cette architecture, alors l'architecture peut exécuter tous les circuits en augmentant le temps total d'exécution.

Cependant, bien que les architectures reconfigurables soient reconfigurables par nature et permettent donc de remplacer le circuit en cours d'exécution par un autre, réaliser une gestion OS demande en plus de pouvoir sauvegarder et restaurer le

contexte d'exécution des circuits applicatifs sur l'architecture, c'est-à-dire l'état des registres et des mémoires utilisés par ces circuits.

La sauvegarde et la restauration du contexte peut être mise en place à trois niveaux :

— au niveau sources : le concepteur peut instrumenter son circuit de façon à fournir un accès aux registres et aux mémoires ;
— à la synthèse : le circuit peut être instrumenté automatiquement par l'outil de synthèse ;
— au niveau de l'architecture : celle-ci peut fournir un accès extérieur aux registres et aux mémoires de son plan de calcul.

Dans ces travaux nous nous intéressons à la troisième solution, car elle ne demande pas aux concepteurs de modifier le design de leur circuits ni de modifier leur flot de synthèse pour s'adapter à des exigences propres à la plateforme d'exécution. Au contraire la troisième solution est complètement transparente du point de vue des concepteurs et des circuits applicatifs qui voient l'overlay comme leur étant entièrement dédié. Les circuits applicatifs ont donc une vue virtuelle de l'overlay.

Nous avons donc instrumenté le plan de calcul de nos overlays pour qu'il fournisse un accès externe aux registres applicatifs et aux mémoires de leur plan de calcul. Nous appelons l'ensemble des ressources additionnelles permettant le chargement et l'extraction du contexte d'exécution des circuits applicatifs le *plan de snapshot* de l'overlay. Cette sous-section traite du mécanisme d'accès aux registres applicatifs, tandis que les mémoires seront vues dans la sous-section suivante.

Le registre de snapshot

L'accès au contenu des registres applicatifs du plan de calcul est réalisé en chaînant les registres applicatifs en un ou plusieurs registres à décalage, permettant le chargement de l'état des registres de manière similaire au chargement d'un bitstream virtuel dans le plan de configuration. La queue de la chaîne ainsi formée est rendue visible depuis l'extérieur de l'overlay de façon à aussi pouvoir extraire l'état des registres applicatifs. Nous appelons registre de snapshot ce registre à décalage qui permet l'extraction et le chargement du contenu des registres applicatifs.

Dans [1], un tel mécanisme a été mis en place pour pouvoir migrer une application d'un cœur d'overlay à un autre. La figure 3.12 montre l'implémentation réalisée pour chaîner les registres : l'implémentation du registre de snapshot demande juste l'ajout d'un multiplexeur devant le registre applicatif sélectionnant entre la sortie de la LUT (fonctionnement normal) et la sortie du registre applicatif précédent (extraction / chargement). Bien que cette implémentation soit économe en ressources, le plan de calcul n'est pas opérationnel le temps que le contenu des registres soit chargé ou extrait de l'overlay.

La figure 3.13 montre notre implémentation du registre de snapshot, les éléments spécifiques au mécanisme de snapshot apparaissant en rouge. De manière

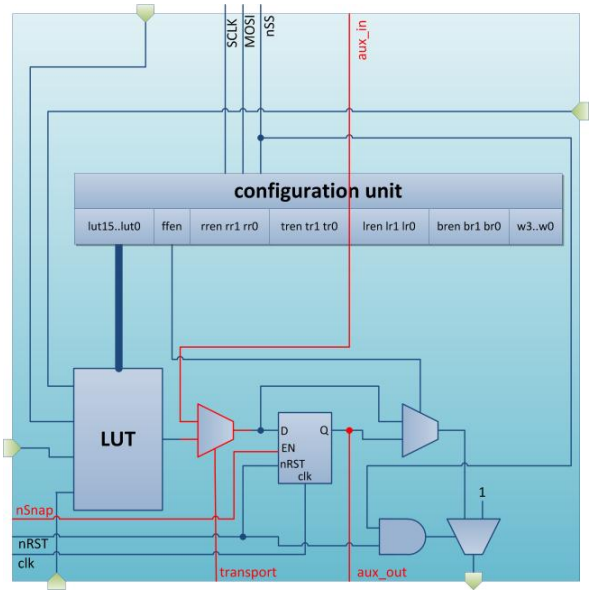


FIGURE 3.12 – CLB avec mécanisme de snapshot [1]

similaire au registre de pré-configuration, chaque registre applicatif est appairé avec un registre additionnel. Ce sont ces registres additionnels qui sont chaînés entre eux pour former le registre de snapshot. Deux multiplexeurs permettent de copier le contenu du registre applicatif dans le registre de snapshot (sauvegarde), de copier le contenu du registre de snapshot dans le registre applicatif (restauration), ou de transférer le contenu du registre de snapshot depuis ou vers l'extérieur de l'overlay (chargement / extraction). Ce mécanisme de snapshot permet de sauvegarder / restaurer l'état des registres applicatifs vers / depuis le registre de snapshot en un seul cycle d'horloge. De plus, les registres applicatifs ne sont pas perturbés par le chargement ou l'extraction du registre de snapshot, le plan de calcul reste donc opérationnel pendant les transferts.

Il est intéressant de noter que même lorsque le registre applicatif d'un BLE n'est pas utilisé par l'application (le multiplexeur sélectionne la sortie de la LUT et non celle du registre applicatif pour fournir la sortie du BLE), le registre applicatif capture quand même la sortie de la LUT tous les cycles d'horloge applicative. La sortie de la LUT est donc copiée dans le registre de snapshot lors d'une sauvegarde, que le BLE soit utilisé en mode combinatoire ou séquentiel par l'application. Ainsi, lors de l'extraction d'un snapshot à des fins de débogage, il est possible d'examiner la valeur de chaque BLE, aussi bien séquentiel que combinatoire.

En appairant chaque registre applicatif avec un registre de snapshot, le mécanisme de snapshot amène un surcoût en ressources du FPGA hôte par rapport à un overlay sans plan de snapshot. Cependant, les registres applicatifs sont des ressources assez rares dans l'overlay (par rapport au nombre de pistes de routage par exemple) ; l'ajout du plan de snapshot a donc un impact assez faible sur la surface occupée par l'overlay sur son hôte. Ce surcoût sera évalué en 3.5.1, il s'étend de 1.8% à 2.6% en LUTs et de 1.6% à 2.0% en flip-flops suivant le FPGA hôte.

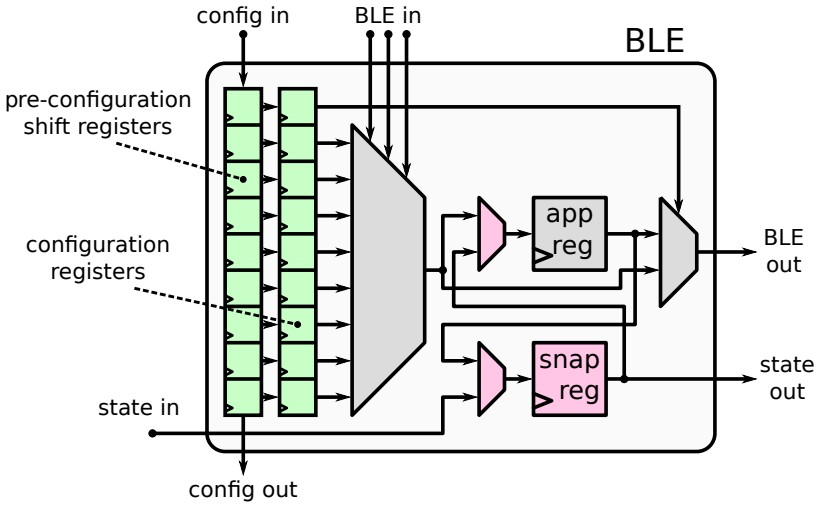


FIGURE 3.13 – Appairage d'un registre applicatif avec un registre de snapshot dans l'implémentation d'un BLE.

3.2.5 Implémentation des mémoires virtuelles

La majorité des applications utilisent des mémoires permettant de stocker un nombre important de données. Dans le cas d'une architecture reconfigurable de type FPGA, implémenter les mémoires du circuit applicatif par des flip-flops consomme un nombre important de ressources reconfigurables, et limite donc la taille des mémoires applicatives réalisables. Les constructeurs de FPGA ont pallié ce problème en intégrant des mémoires directement sur le silicium, en tant que primitives reconfigurables (au même titre qu'une LUT par exemple).

Dans le cas des overlays, la même solution peut être adoptée : intégrer des mémoires pour éviter d'avoir à implémenter les mémoires applicatives via les flip-flops virtuelles. Nous appelons ces mémoires des *mémoires virtuelles*. Cependant, avec le niveau d'abstraction supplémentaire apporté par les overlays, différentes questions se posent quant à l'implémentation et l'utilisation de ces mémoires virtuelles :

— Comment présenter la mémoire virtuelle ? Comme une primitive du plan de calcul similaire aux block-RAM des FPGAs, ou comme des mémoires externes accessibles via les IOs virtuelles de l'overlay ?
— Comment implémenter la mémoire virtuelle ? Via des block-RAM distincts du FPGA hôte, ou via une mémoire externe au FPGA offrant une capacité plus large ?
— Comment accéder à la mémoire virtuelle de l'extérieur du plan de calcul de l'overlay pour permettre de configurer/extraire son contenu ?

La position conceptuelle des mémoires virtuelles : en dehors ou dans le plan de calcul ?

Deux solutions sont possibles pour offrir aux concepteurs la possibilité d'utiliser des mémoires dans leurs circuits applicatifs. La première solution est similaire à

VirtualRC [22]. VirtualRC n'est pas un overlay, mais une couche d'abstraction qui agit comme un "wrapper" : le concepteur écrit sont circuit dans une entité "top level", l'interface de laquelle fournit des connexions à des mémoires, des multiplieurs et des canaux de communication. VirtualRC est portée sur différents FPGA, implémentant selon les spécificités de chaque FPGA les mémoires, multiplieurs et canaux de communication, tout en gardant une interface fixe pour le "top level" offert au concepteur. Le code applicatif écrit dans le top level de VirtualRC est donc portable tel quel sur tous les FPGA pour lesquels VirtualRC a été porté. Une première solution pour offrir des mémoires aux concepteurs de circuits applicatifs ciblant un overlay est de placer ces mémoires à l'extérieur de la matrice de l'overlay, et d'offrir l'interface à ces mémoires via les IO virtuelles de la matrice. Cette solution est illustrée figure 3.14 gauche, elle est simple mais consomme des IO virtuelles de la matrice, et le nombre de mémoires utilisables est ainsi limité.

La deuxième solution consiste placer les mémoires directement dans le plan de calcul de l'overlay en tant que primitives prises en charge par l'outil de synthèse virtuelle, de manière similaire à comment sont intégrés les blocs mémoires et DSP dans la matrice de CLB des FPGAs commerciaux. Cette solution est illustrée figure 3.14 droite. Contrairement à la première solution, la deuxième n'est pas limitée par le nombre d'IO de la matrice et est transparente pour le concepteur de circuits applicatifs.

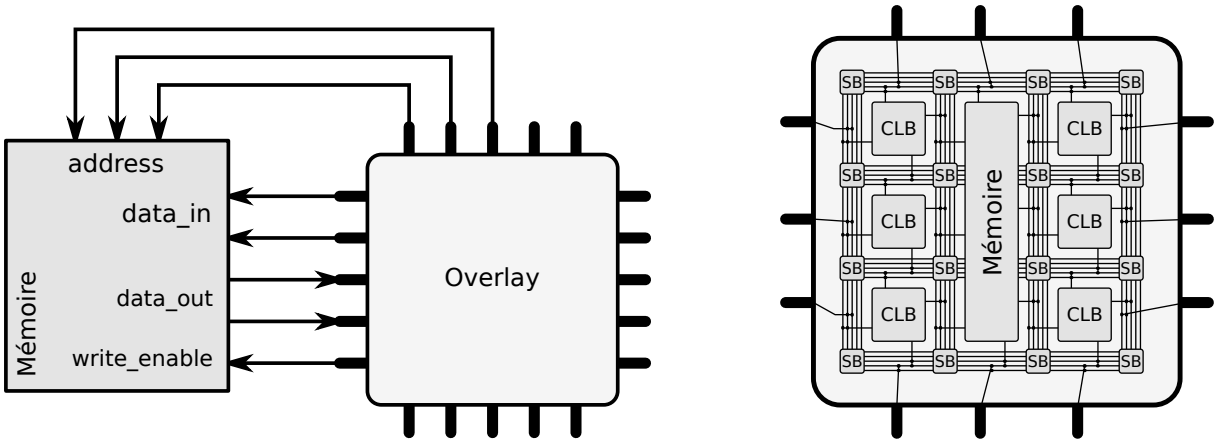


FIGURE 3.14 – Accès à la mémoire par le circuit applicatif. À gauche : via les IOs virtuelles, à droite : la mémoire est intégrée dans le plan de calcul de l'overlay, accès direct via les ressources de routage

L'implémentation des mémoires virtuelles : dans des block-RAM de l'hôte ou dans une mémoire externe au FPGA ?

Physiquement, ces mémoires peuvent être implémentées en utilisant les mémoires internes du FPGA hôte, ou être chacune mappées sur des espaces mémoire distincts d'une ou plusieurs mémoires externes au FPGA. Utiliser les mémoires internes du FPGA (block RAM) est la solution la plus simple : ces mémoires ont une latence fixe d'un cycle d'horloge et il est possible d'instancier plusieurs de ces mé-

moires de manière distincte (elles ont chacune leur propre interface). Il suffit donc de connecter l'interface de la mémoire hôte instanciée à l'interface de la mémoire virtuelle, comme illustré figure 3.15 gauche.

Utiliser une mémoire externe pour implémenter les mémoires virtuelles a cependant l'avantage de permettre des mémoires virtuelles de plus grande capacité. L'inconvénient d'utiliser une mémoire externe est que la latence d'accès à une donnée est longue et n'est pas fixe. En effet l'accès à la mémoire externe doit être partagé entres les différentes mémoires virtuelles et possiblement d'autres modules indépendants de l'overlay (un processeur softcore par exemple). Si le contenu de toutes les mémoires virtuelles est placé dans la même mémoire physique, un contrôleur doit arbitrer l'accès des interfaces des mémoires virtuelles à la mémoire physique, comme illustré figure 3.15 droite. Pour gérer cette latence variable, plutôt que de proposer une interface classique (`data_in`, `data_out`, `addr`, `we`), l'interface des mémoires virtuelles peut disposer de signaux supplémentaires indiquant quand une opération mémoire est demandée, et quand elle a bien été réalisée. Les mémoires virtuelles ne sont donc plus vues comme des mémoires, mais plutôt comme des bus, ce qui impact le concepteur et les circuits applicatifs utilisant ces mémoires.

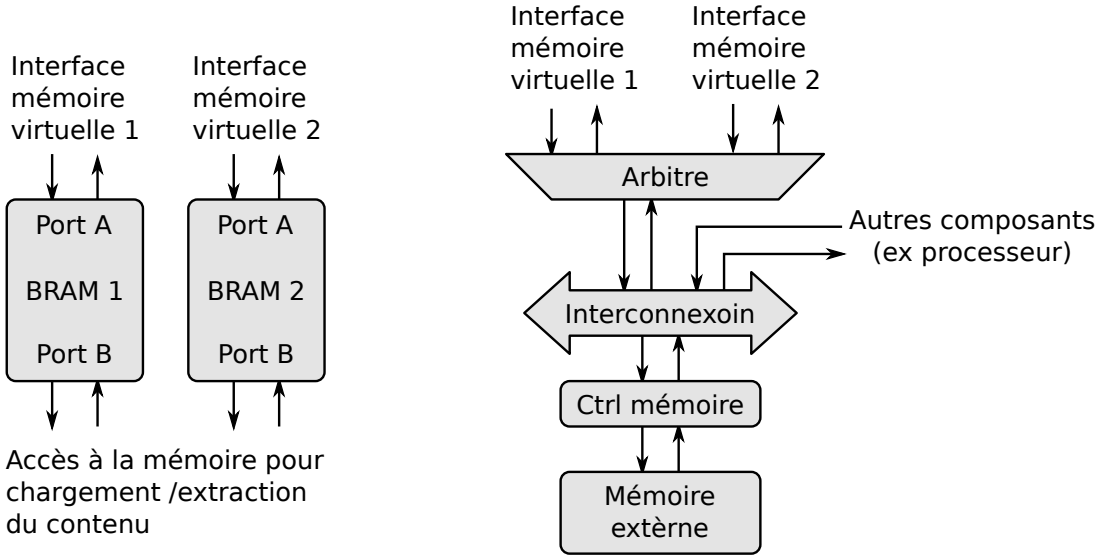


FIGURE 3.15 – Implémentation des mémoires virtuelles via des mémoires block RAM (gauche), ou une mémoire externe (droite)

Une solution permettant d'utiliser une mémoire externe comme support des mémoires virtuelles sans pour autant devoir modifier leur interface consiste à geler le plan de calcul de l'overlay tant que la requête d'écriture ou de lecture d'une mémoire virtuelle n'a pas encore été servie. Cela est possible en gardant non actif le signal de l'horloge applicative et en empêchant les accès aux IOs virtuelles de l'extérieur de l'overlay. Cette méthode permet donc de bénéficier de la grande capacité d'une mémoire externe sans impacter le concepteur et ses circuits applicatifs par une interface du type bus. Cependant, plus le circuit applicatif utilise (instancie) de mémoires virtuelles distinctes, plus le plan de calcul passera de temps à être gelé. Le surcout de cette méthode en termes de temps d'inactivité du plan de calcul

dépend :

- du chemin critique du circuit applicatif, c'est-à-dire du nombre de cycles d'horloge physique nécessaires entre deux cycles d'horloge applicative ;
- du nombre de mémoires virtuelles utilisées par le circuit applicatif et qui peuvent possiblement devoir accéder à la mémoire externe dans le même cycle d'horloge applicatif (changement sur au moins un des signaux `addr`, `data_in` ou `we`);
- de la latence de la mémoire externe et du taux d'utilisation du bus qui permet d'y accéder.

Implémenter un cache en mémoire interne (block RAM) par mémoire virtuelle peut limiter ce surcout.

Changement de contexte : accès au contenu des mémoires virtuelles

Le fait que le plan de calcul de l'overlay est décrit en HDL et est implémenté sur FPGA permet de donner un accès auxiliaire aux mémoires de manière assez simple, par exemple en utilisant le deuxième port des mémoires block RAM ou en multiplexant l'accès à leur interface entre plan de calcul et accès externe à l'overlay. Si une mémoire externe est utilisée via un bus, il est possible d'accéder au contenu des mémoires virtuelle directement depuis ce bus. Lorsque le contexte d'exécution d'un circuit applicatif doit être remplacé par un autre, l'intégralité du contenu des mémoires virtuelles doit donc être copié vers une autre location mémoire, puis remplacé par le contenu des mémoires virtuelles du nouveau circuit applicatif. Pour des raisons de sécurité (isolation des circuits applicatifs), même si un circuit n'utilise pas de mémoire virtuelle, leur contenu doit quand même être effacé, pour garantir qu'un circuit ne puisse pas accéder aux données d'un autre circuit.

La lecture plus l'écriture de l'intégralité du contenu des mémoires virtuelles à chaque changement de contexte applicatif prend un temps important (qui dépend de la taille cumulée des mémoires virtuelles et de la latence de la mémoire physique mises en œuvre) par rapport au temps de transfert d'un snapshot. De la même manière que pour la pré-configuration ou le registre de snapshot, notre solution pour ne pas impacter le temps pendant lequel le plan de calcul de l'overlay n'est pas opérationnel lors d'un changement de contexte est de mettre en place un système de double buffer, c'est-à-dire d'utiliser deux fois plus de mémoire physique que le plan de calcul ne propose de mémoire virtuelle. Chaque mémoire virtuelle est donc mappée sur deux espaces mémoires physiques. Pendant que le premier espace mémoire est utilisé depuis le plan de calcul par le circuit courant, le contenu de la mémoire du circuit précédant peut être sauvegardé depuis le deuxième espace mémoire et le contenu de la mémoire du prochain circuit peut ensuite y être écrit. Lors du changement de contexte qui suit, l'espace mémoire physique mappé par l'interface de la mémoire virtuelle passe du premier au deuxième. Et ainsi de suite. Si les deux espaces mémoires sont physiquement consécutifs, changer le bit de poids for de l'adresse permet simplement de changer l'espace mappé. Cette méthode permet donc de réduire le temps de changement de contexte des mémoires virtuelles à un seul cycle d'horloge, tant que la période de reconfiguration de l'over-

lay est inférieure au temps nécessaire pour sauvegarder puis restaurer l'intégralité du contenu des mémoires implémentant les mémoires virtuelles. La figure 3.16 illustre cette utilisation des mémoires physique pour implémenter une mémoire virtuelle.

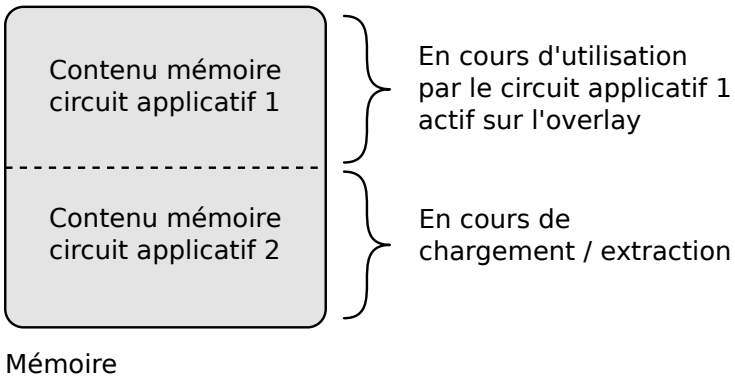


FIGURE 3.16 – La mémoire physique contient simultanément deux contenus de la mémoire virtuelle qu'elle implémente

3.3 Synthèse physique des overlays

3.3.1 Spécificités d'un overlay en tant que design FPGA

La majorité des circuits qui sont implémentés sur FPGA ne comportent pas de boucles combinatoires, et quand ils en ont, il s'agit souvent d'une erreur de design. Les algorithmes d'analyse de timing, de placement et de routage des outils de synthèse physique sont prévus pour travailler sur des designs qui ne présentent pas de boucle combinatoire. En effet, pour respecter les contraintes de timing lors du placement et du routage, l'outil de synthèse physique calcul, pour la netlist du circuit sur lequel il travaille, le délai de chaque chemin combinatoire séparant deux registres (*timing path*). Lorsque des boucles combinatoires sont présentes dans la netlist, l'outil les casse (pour son analyse) de manière à ce qu'il n'y ait plus que des chemins directs d'un registre à un autre, et peut alors calculer le délai de chaque timing path. Cependant, du fait de casser une boucle combinatoire à un endroit arbitraire, plusieurs chemins potentiels ne sont pas analysés. L'analyse de timing ainsi que le placement et le routage qui en dépendent sont donc impactés par la présence de telles boucles, ce qui diminue la qualité du circuit synthétisé.

Du point de vue RTL, un overlay se distingue d'un design classique sur deux points :

— il présente un nombre important de boucles combinatoires, qui ne sont pas des erreurs de design ;
— il présente un nombre important de timing paths (chemin combinatoire d'un registre à un autre) ;

Par exemple, pour un overlay vFPGA-flexible de taille 6×6, 4 BLE de LUT-4 par CLB et une largeur des canaux de routage de 14, l'outil `trce` d'analyse de timing

de la suite ISE indique dans son rapport avoir trouvé 4714 boucles combinatoires et couvert plus de $6 \cdot 10^{158}$ timing paths, tout en précisant qu'à cause des boucles certains chemins peuvent ne pas avoir été analysés.

En effet, lorsqu'il est synthétisé sur un FPGA, l'overlay n'est par définition pas programmé (par un bitstream virtuel) puisqu'un overlay implémente une architecture qui est elle-même reconfigurable. Les ressources de routages d'un overlay – et en particulier pour un overlay grain fin – sont très flexibles, c'est-à-dire qu'un grand nombre d'interconnexions programmables est possible entre différentes ressources de routage. La figure 3.17 gauche montre deux exemples de boucles combinatoires dans le routage inter CLB, tandis que la figure 3.17 droite illustre une boucle combinatoire à l'intérieur d'un CLB. De plus, les ressources de routage permettent de relier deux registres applicatifs par un très grand nombre de chemins possibles (illustré figure 3.18).

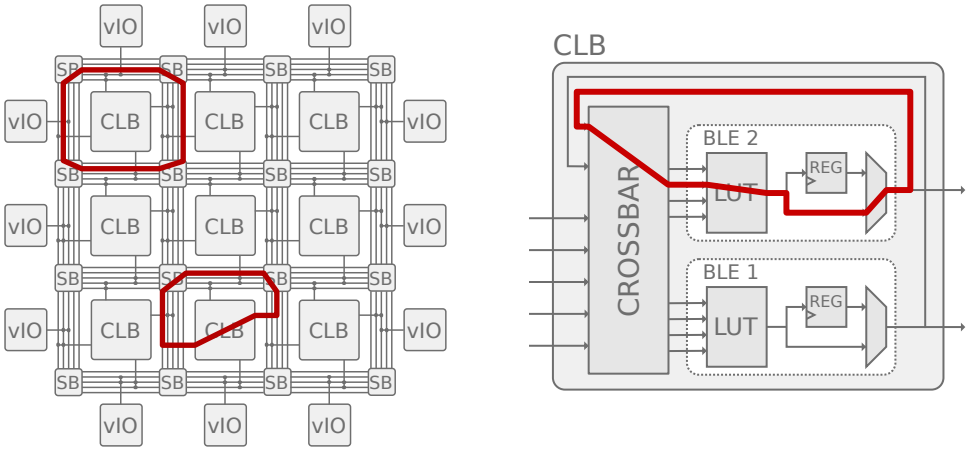


FIGURE 3.17 – Exemples de boucles combinatoires dans le routage inter CLB (à gauche), et à l'intérieur d'un CLB (à droite).

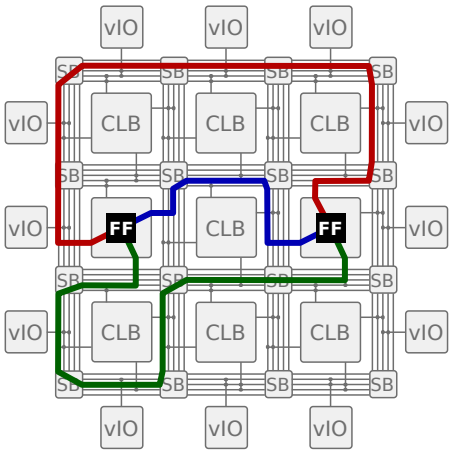


FIGURE 3.18 – Exemples de différents chemins possibles reliant deux registres applicatifs

La présence d'un tel nombre de boucles combinatoires et de timing paths augmente le temps de synthèse, peut aussi empêcher la synthèse d'aboutir (par exemple

avec l'outil Quartus d'Altera), diminue la compréhension qu'a l'outil du design sur lequel il travaille, et dégrade ainsi la qualité de l'overlay synthétisé. Un des effets de cette dégradation est que les ressources du plan de calcul de l'overlay peuvent être placées sur le FPGA selon une configuration qui ne correspond pas à la représentation conceptuelle du plan de calcul. Par exemple les ressources du CLB(1,1) peuvent être physiquement placées plus proches des ressources du CLB(9,7) que du CLB(1,2). Ainsi les délais des ressources atomiques similaires du plan de calcul peuvent avoir des délais physiques différents. Par exemple, rien n'assure que toutes les pistes des canaux de routages aient le même délai.

3.3.2 Synthèse des overlays : passage à l'échelle

Le temps de synthèse et l'effort demandé au synthétiseur limitent la tailles des overlays qu'il est possible de synthétiser. Pour pallier ce problème ainsi que pour augmenter la qualité ainsi que la régularité de l'overlay synthétisé, l'approche de [50] est de ne pré-synthétiser qu'une tuile de l'overlay (un CLB et son routage environnant) puis d'utiliser un outil de floorplan pour répliquer la tuile suivant la matrice de l'overlay. Le fait de ne synthétiser qu'une tuile supprime (lors de la synthèse) toutes les boucles et les timing paths inter CLB, et la réplication de cette même tuile sur l'ensemble de la matrice de l'overlay assure la régularité du circuit synthétisé. Cependant, l'utilisation d'un outils de floorpan dans cette approche implique de sortir du flot de synthèse classique pour FPGA, et demande une étape manuelle pour chaque synthèse. Cette approche est donc adaptée pour l'optimisation fine d'un overlay, mais pas pour les itérations exploratoires des cycles de développement lors de la conceptions d'un overlay. Une autre approche utilisée lors de la synthèse d'architectures reconfigurable sur des ASIC est de contraindre chaque ressource du plan de calcul. Cependant, sur FPGA, nous n'avons pas trouvé le moyen de contraindre des sous-segments de chemins combinatoires, les outils FPGA semblant n'accepter des contraintes de délai que pour chemins allant d'un registre à un autre.

La solution utilisée dans ces travaux consiste à placer des registres additionnels parmi les ressources de routage de façon à casser toutes les boucles combinatoires ainsi qu'à limiter le nombre de chemins combinatoires possibles entre deux registres à une quantité raisonnable (< 10). Cette approche étant réalisée au niveau RTL de l'implémentation de l'overlay, elle ne modifie pas le flot de synthèse et peut être mise en œuvre sur tout FPGA quels que soient leurs outils de synthèse associés. Comme illustré figure 3.19, ces registres additionnels sont placés sur chaque piste de routage du plan de calcul. Pour équilibrer la longueur des chemins combinatoires, dans les CLBs, les registres additionnels ont été placés à la sortie du crossbar et non à la sortie des BLEs.

Avec ces registres additionnels, lors de la synthèse physique de l'overlay sur FPGA, l'outil peut alors effectuer une analyse de timing correcte du design de l'overlay et réaliser un placement et routage "au mieux" de celui-ci. L'effort demandé au synthétiseur et donc le temps de synthèse sont diminués, et l'overlay

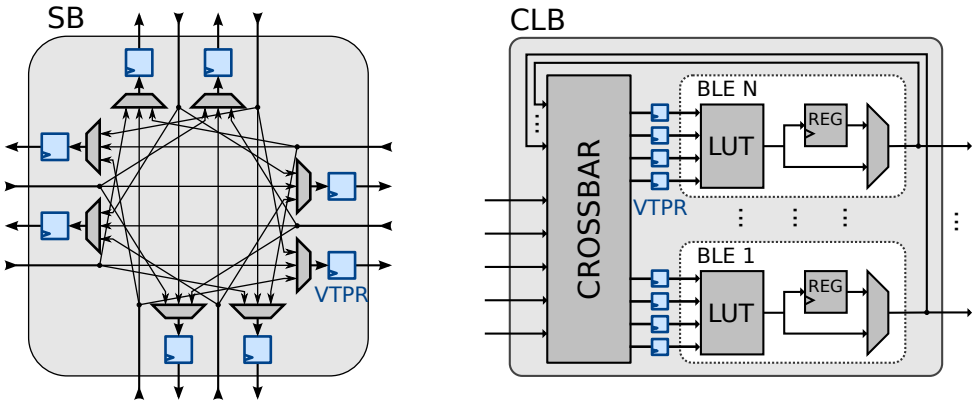


FIGURE 3.19 – Des registres additionnels qualifiés de VTPRs sont ajoutés dans l'implémentation du plan de calcul pour casser les boucles combinatoires et les timing paths.

synthétisé est de meilleur qualité. En effet, pour répondre à la contrainte de la période de l'horloge, le synthétiseur travaille de façon à ce que le délai de tout chemin combinatoire entre deux registres soit au maximum la période de l'horloge. Le synthétiseur travaille donc de façon à égaliser les délais de toutes les ressources atomiques du plan de calcul séparées par deux registres.

Ces registres additionnels sont qualifiés de VTPRs, pour Virtual Time Propagation Register, car ils permettent d'émuler la propagation des signaux applicatifs dans le plan de calcul de l'overlay. Nous verrons au chapitre 5 section 5.4 l'avantage des VTPRs pour l'analyse de timing des circuits applicatifs sur l'overlay par l'outil de synthèse virtuelle. Conceptuellement, les VTPRs ne font pas partie de l'architecture du plan de calcul, mais de son implémentation, c'est-à-dire qu'ils ne concernent pas l'aspect *fonctionnel* de l'overlay, mais son aspect *physique*. En effet, au contraire des registres applicatifs (en sortie des LUT dans les BLE), les VTPRs sont transparents pour les circuits applicatifs, ce ne sont pas des ressources configurables ni manipulables par l'outil de synthèse virtuelle. Le rôle des VTPRs est de simplifier la synthèse physique de l'overlay sur FPGA ainsi que d'abstraire le plan de calcul des caractéristiques physique de l'overlay synthétisé.

L'ajout d'un VTPR sur chacune des pistes de routage de l'overlay ainsi qu'aux sorties des crossbars des CLBs a un impact sur l'occupation en surface de l'overlay sur son hôte. Ce surcoût sera évalué en 3.5.1, il représente 35% en LUTs et 22% en flip-flops par rapport à une implémentation sans VTPRs.

3.3.3 Temps de synthèse avec ou sans les VTPRs

Le tableau 3.1 présente les résultats expérimentaux du temps en secondes mis par les différents outils de la suite ISE de Xilinx pour synthétiser un vFPGA-flexible, généré pour plusieurs dimensions, sans (colonnes "Nu") et avec VTPRs (colonnes "VTPRs"). XST est l'outil de synthèse logique, il prend en entrée les sources RTL et génère une netlist générique d'éléments logiques. MAP est converti la net-

TABLE 3.1 – Temps de synthèse avec et sans VTPRs

Dimensions		XST			MAP			PAR			TRCE			Temps de synthèse total		
Taille	BLEs	Nu	VTPR	Ratio	Nu	VTPR	Ratio	Nu	VTPR	Ratio	Nu	VTPR	Ratio	Nu	VTPR	Ratio
2×2	16	32	32	1.00	133	120	1.11	103	66	1.56	35	33	1.06	303	251	1.21
4×4	64	68	61	1.11	228	238	0.96	344	138	2.49	71	39	1.82	711	476	1.49
6×6	144	159	146	1.09	530	298	1.78	60538	155	390.57	219	49	4.47	61446	648	94.82
8×8	256	328	354	0.93	909	524	1.73	3088	220	14.04	493	63	7.82	4818	1161	4.15
10×10	400	661	715	0.92	1842	763	2.41	4940	350	14.11	1208	84	14.38	8651	1912	4.52
12×12	576	1296	1371	0.94	4838	1179	4.10	39856	474	84.08	3289	105	31.32	49279	3129	15.75
14×14	784	2343	2487	0.94	4613	3904	1.18	18617	654	28.47	5007	154	32.51	30580	7199	4.24

list générique en une netlist d'éléments disponibles dans le FPGA ciblé. L'outil MAP réalise aussi le placement des éléments de la nouvelle netlist sur les ressources de la cible. PAR réalise le routage des éléments placé suivant les ressources de routage disponibles dans le FPGA ciblé. Finalement, l'outil TRCE réalise l'analyse de timing du circuit synthétisé. Ces résultats ont été publiés dans [58].

Le temps mis par XST pour réaliser la synthèse logique ne semble pas impacté de manière importante par la présence ou non de VTPRs. Il est même légèrement plus rapide sans VTPRs : en effet, les VTPRs sont des ressources additionnelles. Le rapport du temps mis par l'outil MAP sans VTPRs par rapport au temps mis avec VTPRs s'étend de 1 à 4. Ce rapport ne suit pas les dimensions de l'overlay et semble erratique. De même, le temps mis par PAR sans VTPRs ne suis pas les dimensions de l'overlay, au contraire du temps mis avec VTPRs. Le rapport du temps sans et avec VTPRs s'étend de 1.5 à 390. Le rapport des temps pour l'outil d'analyse de timing TRCE est régulier, il suit les dimensions de l'overlay et s'étend de 1 à 32.

La figure 3.20 montre le temps de synthèse total avec et sans VTPRs, rapporté par le nombre de BLE. Il est visible que le temps de synthèse avec VTPRs est plus prédictible. Les pics des temps de synthèse sans VTPRs, pour les matrices de 6×6 et 12×12 peuvent s'expliquer par les heuristiques des outils qui cassent aléatoirement les boucles combinatoires pour analyser les chemins.

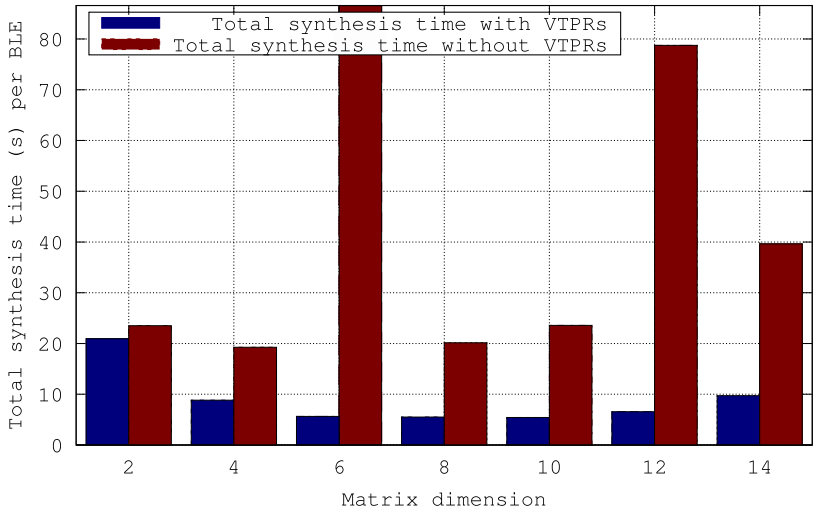


FIGURE 3.20 – Temps de synthèse par BLE sur un FPGA Artix-7, avec et sans VTPRs.

3.4 Génération des overlays

Lors de l'exploration de l'espace de conception d'un overlay, la vérification de son fonctionnement et l'évaluation de son coût et de ses performances physiques demandent de pouvoir générer l'implémentation de l'overlay en fonction de la description de son plan de calcul, pour pouvoir ensuite intégrer et synthétiser cette implémentation. Dans les sections précédentes il a été vu ce qu'était un plan de calcul, comment implémenter ce plan de calcul est les ressources additionnelles constituant un overlay, ainsi que l'utilisation des VTPRs pour permettre de synthétiser l'overlay. Cette section traite de la génération de l'implémentation d'un overlay à partir de la description de son plan de calcul. La génération de l'overlay consiste en la génération des fichiers sources décrivant l'implémentation RTL de l'overlay dans un langage HDL. Cette description RTL de l'implémentation comprend les trois plans conceptuels de l'overlay qui sont le plan de configuration, le plan de calcul et le plan de snapshot, ainsi que l'insertion des VTPRs. Deux méthodes différentes ont été utilisées pour la génération des overlays vFPGA-restreint et vFPGA-flexible : génération par template et génération par parcours du modèle.

3.4.1 vFPGA-restreint : génération par template

Lors de la phase exploratoire, il est désirable de pouvoir faire varier les paramètres du plan de calcul tout en assurant que chaque ensemble de paramètre choisi est valide (c'est-à-dire que chaque ensemble de paramètre permet de générer une architecture fonctionnelle) et que le plan de calcul inféré par ces paramètres est exploitable par l'outil de synthèse virtuelle. Il est aussi désirable que n'importe quel outil indépendant du générateur de l'overlay puisse cibler une instance d'overlay.

Ainsi, l'architecture vFPGA-restreint ne repose que sur quatre paramètres qui sont : L la largeur de la matrice, H sa hauteur, W le nombre de pistes par canal de routage et K le nombre d'entrées par LUT. Au-delà de ces quatre paramètres, le méta-modèle de l'architecture (i.e. le modèle du modèle de l'architecture vFPGA-restreint) est fixe. C'est-à-dire que le plan de calcul est entièrement inféré des paramètres L, H, W et K. Le plan de configuration et le plan de snapshot étant dérivés du plan de calcul, l'implémentation de l'overlay est entièrement définie par ces quatre paramètres. Par exemple, la position et la signification de chaque bit du bitstream sont conditionnées uniquement par l'ensemble $\{L, H, W, K\}$.

Le fait que le méta-modèle soit fixe et que le modèle ne dépende que de quatre paramètres permet à n'importe quel outil indépendant du générateur de pouvoir cibler une instance de vFPGA-restreint juste à partir de ces paramètres. Dans ces travaux, l'outil Madeo [48] a été utilisé pour réaliser le placement, le routage ainsi que l'extraction des bitstreams de circuits applicatifs pour l'architecture vFPGA-restreint. De plus, tant que $L > 0$, $H > 0$ et $2 \leq K \leq W$, l'architecture vFPGA-restreint est valide et fonctionnelle, même si les rapports entre les différents paramètres ne sont pas nécessairement judicieux. Ainsi, il est très simple de générer

rapidement une collection d'instances d'architectures en faisant varier les paramètres pour étudier l'impact de chacun d'eux sur le coût et les performances physiques de l'overlay ainsi que sur sa flexibilité fonctionnelle.

Le vFPGA-restreint est une architecture très régulière, dans le sens où le même motif est répliqué à l'identique le long des deux dimensions de la matrice. Le modèle de chaque élément hiérarchique de l'architecture (par exemple une tuile, une LUT, ou une SB) est paramétré par le même ensemble $\{L, H, W, K\}$. Ces différents éléments correspondent à :

— la LUT, de K entrées ;
— l'unité fonctionnelle, qui intègre la LUT, le registre applicatif ainsi que son registre de snapshot associé ;
— la Connection Box, qui connecte les entrées et la sortie de l'unité fonctionnelle dans un canal de routage (de W pistes) ;
— la Switch Box, qui connecte les différentes pistes des quatre canaux de routage afférents entre elles de manière programmable suivant le schéma Wilton ;
— le registre de configuration, qui contient les bits de configuration relatifs à chaque élément configurable d'une tuile ;
— la tuile, entité hiérarchique qui intègre une unité fonctionnelle, une switch box, un canal de routage vertical et un horizontal, qu'une connection box par canal de routage ainsi qu'un registre de configuration ;
— la matrice, qui intègre $L \times H$ tuiles et raboute leurs canaux de routages ainsi que les serpentins de configuration et de snapshot, et rassemble les IOs en périphéries en un vecteur d'entrées et un de sorties.

Cependant, le fait que le méta-modèle de l'architecture est fixe limite l'exploration de l'espace de conception à l'ensemble des paramètres $\{L, H, W, K\}$. Il peut être désirable d'augmenter le méta-modèle de l'architecture par l'ajout de nouveaux éléments, par exemple pour y intégrer d'autres opérateurs que les LUTs, tels que des blocs mémoires ou des blocs DSP. De nouveaux paramètres doivent alors être ajoutés au méta-modèle, tels que la taille des mémoires ou des blocs DSP, ainsi que le pourcentage de blocs mémoire et DSP par rapport LUTs.

Outre la simplicité de définition d'une architecture, la régularité de celle-ci et le fait que le modèle de chaque élément est conditionné par un même ensemble restreint de paramètres permet de générer simplement l'implémentation de l'overlay à partir de templates, c'est-à-dire que la description RTL de chaque élément hiérarchique est produit à partir d'un gabarit configuré par l'ensemble de paramètres. L'écriture du générateur correspond donc à l'écriture des gabarits. Mis à part la prise en compte des paramètres, l'écriture des gabarits se fait directement dans le langage HDL ciblé, ce qui facilite leur conception. Ainsi, la modification du méta-modèle de l'architecture (par exemple l'ajout de blocs DSP) est prise en compte facilement dans le générateur car elle n'implique que la modification d'un ou plusieurs gabarits.

3.4.2 vFPGA-flexible : génération par parcours de modèle

La génération par template facilite l'exploration de l'espace de conception en limitant le méta-modèle de l'architecture à un nombre restreint de paramètres. Cependant, un changement de l'architecture au-delà de la variabilité autorisée par les paramètres demande une modification du méta-modèle lui-même. Bien que cette modification est facilement prise en compte au niveau du générateur du fait de la génération par template, elle demande la réécriture manuelle des gabarits. De plus, la modification du méta-modèle casse la compatibilité avec les outils de synthèse virtuelle, qui doivent eux aussi être adaptés pour prendre en compte les modifications du méta-modèle.

Le but de l'architecture vFPGA-flexible est de permettre une plus grande flexibilité des architectures qu'il est possible d'instancier, sans avoir à modifier le méta-modèle. Aussi, nous voulions avoir la possibilité d'utiliser cette architecture avec différents outils de synthèse virtuelle, notamment Madéo et VPR [43]. En effet, VPR est un outil largement utilisé pour l'exploration de l'aspect fonctionnel des architectures reconfigurables, et c'est l'outil open source qui implémente les algorithmes les plus aboutis de l'état de l'art.

Le modèle et son expressivité

La description de l'architecture fournie à VPR pour qu'il puisse cibler celle-ci lors de sa synthèse ne permet pas d'exprimer finement les détails de l'architecture. En effet, VPR construit le modèle détaillé de l'architecture à partir des informations haut niveau qui lui sont fournies dans la description. Cela est équivalent à la modélisation du vFPGA-restreint à partir de quelques paramètres, bien que les descriptions destinées à VPR soient tout de même beaucoup plus flexibles que seulement quatre paramètres. Ainsi, VPR a son propre méta-modèle. Le modèle détaillé de l'architecture que construit VPR présente de nombreuses irrégularités : par exemples la topologie d'interconnexion entre les pistes de routage et les entrées et sorties des CLB n'est pas tout à fait la même selon que le CLB se situe à gauche, en bas ou au cœur de la matrice. Un autre exemple est la topologie d'interconnexion des SBs en périphérie de la matrice, qui ne correspond pas à la topologie de celle du cœur, qui n'est pas décrite dans la description de l'architecture fournie à VPR et qui diffère selon la largeur des canaux de routage. Pour que l'architecture produite par le générateur corresponde à l'architecture modélisée par VPR, tout en gardant le générateur séparé de VPR (pour pouvoir aussi utiliser d'autres outils de synthèse virtuelle), il faut que le méta-modèle utilisé par le générateur corresponde au méta-modèle de VPR. Cependant, cela implique de spécialiser l'architecture pour VPR. Aussi, un changement de VPR impactant son méta-modèle (par exemple lors d'un changement de version) impliquerait d'adapter le générateur. C'est pourquoi l'architecture vFPGA-flexible a son propre méta-modèle, mais celui-ci est assez flexible et bas niveau pour permettre de modéliser finement l'architecture et donc de modéliser des irrégularités dans celle-ci (par exemple la topologie d'interconnexion précise de toutes les SBs). Il est ainsi possible de générer

des architectures vFPGA-flexible qui sont compatibles avec VPR.

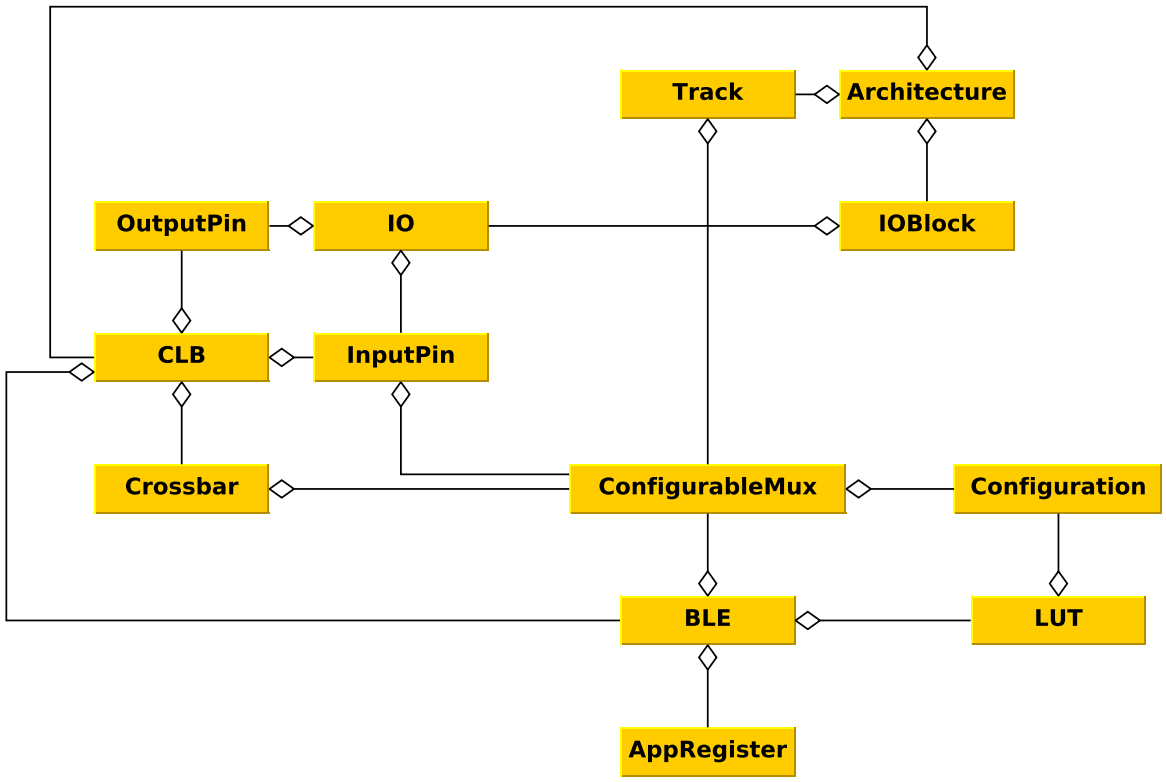


FIGURE 3.21 – Méta-modèle simplifié des architectures

Le méta-modèle de l'architecture vFPGA-flexible est présenté figure 3.21. L'ensemble des architectures qu'il permet de modéliser sont des architectures en îlots de calculs, composées d'une matrice de CLBs entourés de canaux de routage, le tout couronné d'IO. Les BLEs comportent une LUT simple et un registre applicatif qui peut être contourné. Plusieurs BLEs peuvent être regroupés dans un CLB, avec une matrice d'interconnexion locale au CLB qui permet d'alimenter les BLEs avec les entrées du CLB et les sorties des BLEs. Par contre, ce méta-modèle ne permet pas d'exprimer toutes les architecture VPR car il ne permet pas encore de modéliser des éléments comme des pistes de routage de longueur supérieure à un CLB, des LUTs fracturables ou encore des macro-blocks.

Le méta-modèle de l'architecture vFPGA-flexible permet donc de modéliser à bas niveau une instance d'architecture. Pour cela, un ensemble de paramètres numériques tels que pour le vFPGA-restreint ne suffit pas : le modèle doit aussi contenir une partie de la structuration de l'architecture. Le modèle se complexifie, et l'exprimer demande un langage textuel. Pour cela, un DSL (Design Specific Language) a été conçu, celui-ci repose sur les S-expressions (expressions symboliques), car celles-ci proposent une syntaxe simple à utiliser, simple à analyser par un programme, et permet d'organiser de manière souple des données complexes.

Un exemple de fichier de description en S-expression d'une architecture vFPGA-flexible et la représentation du plan de calcul issu de cette description sont donnés

en annexe A.1. Au niveau de l'implémentation de l'overlay, la description permet d'indiquer la présence ou non des VTPRs et du plan de snapshot, ainsi que l'implémentation ou non de la pré-configuration dans le plan de configuration. Au niveau du plan de calcul, il est possible de décrire différent CLBs selon leur paramètres N (nombre de BLEs et de sorties du CLB), I (nombre d'entrées du CLB) et K (nombre d'entrées par LUT). Pour chaque CLB décrit, il est possible de détailler plusieurs topologies d'interconnexion des entrées et des sorties du CLB aux pistes de routages entourant celui-ci. De même, il est possible de décrire différents blocs d'IOs selon les connexions de chaque IO aux pistes de routage. La disposition dans la matrice des blocs d'IOs et des CLB suivant leur connexion au routage est précisé dans la description. La largeur des canaux de routage doit être présente dans la description, ainsi que la disposition des Switch Boxes dans la matrice. Le mot clef "Wilton" permet d'instancier une SB de topologie Wilton [wilton]. Il est aussi possible de décrire manuellement d'autres topologies de SB.

Utilisation du modèle et génération

Un framework nommé *ArGen* a été développé pour utiliser le vFPGA-flexible. Cet outil lit la description S-expressions d'une architecture pour en élaborer le modèle. L'architecture est modélisée "à plat", c'est-à-dire que le routage ne comporte plus que des pistes de routage et il n'apparait plus aucune notion de canaux, de switch box ni de connexion box, qui sont des constructions hiérarchiques utilisées seulement pour décrire le plan de calcul. Ensuite, ArGen peut générer le code RTL de l'architecture, lire les fichiers de placement et routage produits par VPR, effectuer une analyse de timing d'un circuit applicatif placé et routé, et produire un bitstream virtuel correspondant au circuit applicatif et qui est compatible avec l'overlay généré en RTL.

Comme la matrice de l'architecture peut être irrégulière, le vFPGA-flexible se prêtent mal à une génération par template, où la même tuile est répliquée le long de la matrice. Dans l'outillage *ArGen*, nous profitons d'avoir le modèle de l'architecture construit en mémoire pour générer "à plat" le code RTL correspondant au routage inter CLB (pistes de routage et pins d'entrées des CLBs et des IOs). Les CLBs ainsi que les BLEs et les crossbars qui les composent sont quant à eux toujours écrits de manière hiérarchique par template, car ils ne sont définis que par les paramètres {N, I et K}.

Les pins d'entrées des CLB et des pads de sortie ainsi que les pistes de routage peuvent se connecter à d'autres pistes, et pins de sorties. Ces éléments de routage du plan de calcul de l'architecture sont donc chacun implémentés par un multiplexeur dont les entrées sont reliées aux ressources de routage afférentes, et dont les bits de sélection sont pilotés par les registres de configuration. La totalité du routage inter CLB du plan de calcul de l'architecture peut donc se décrire au niveau RTL en parcourant le modèle construit et en instanciant les multiplexeurs des pistes de routage et des pins d'entrées, tout en les connectant entre eux et avec les pins de sorties.

Comme le montre le méta-modèle de l'architecture vFPGA-flexible présenté figure 3.21, les seuls éléments du modèle auxquels est associé une configuration sont les multiplexeurs et les LUT. Lors de la génération de l'implémentation RTL de l'overlay, le nombre de bit de configuration associé à chaque multiplexeur est inféré à partir du nombre de ses connexions afférentes, tandis que la taille de la configuration des LUT est inférée à partir de leur nombre d'entrées. Le registre du plan de configuration est instancié par un registre de longueur égale à la somme du nombre de bit de configuration de chaque multiplexeur et LUT du modèle. Ces derniers sont ensuite mis à jour avec la position dans le registre de configuration des bits qui leur sont associés. Lors de l'écriture des multiplexeurs dans le code généré, ceux-ci sont connectés sur leurs signaux de sélection par les bits correspondant du registre de configuration. Si le mécanisme de pré-configuration doit être implémenté, le registre de configuration est doublé. Le registre de configuration est ensuite chaîné en un ou plusieurs registres à décalage pour permettre la configuration de plusieurs bits par cycle d'horloge (voir 3.2.2). Les registres du plan de snapshot sont instanciés dans les BLE. Ils sont chaînés de la même manière que le plan de configuration.

3.5 Évaluation et modélisation du coût en ressources

3.5.1 Évaluation du coût de l'instrumentation

Le but de cette sous-section est d'évaluer le coût additionnel en ressources FPGA consommées par l'intégration des VTPRs, du plan de snapshot et du mécanisme de pré-configuration du plan de configuration. Le plan de calcul utilisé pour cette expérience est une architecture vFPGA-flexible de 10×10 CLBs de 4 BLEs comportants des LUTs à 4 entrées, avec 10 entrées par CLB. Pour observer comment les surcoûts des VTPRs, du snapshot et de la pré-configuration évoluent avec la flexibilité du plan de calcul, différentes meusures ont été réalisées en faisant varier le nombre de pistes par canal de routage (paramètre W) de 8 à 24. La flexibilité des entrées et des sorties des CLBs – c'est-à-dire le rapport du nombre de piste de routage auxquelles peut se connecter une entrée (ou une sortie) sur le nombre de piste par canal de routage W – a été gardé constant à $fc_{in} = \frac{1}{4}$ pour les entrées et $fc_{out} = \frac{1}{2}$ pour les sorties. Chacun des overlays issu de ces paramètres a été synthétisée sur un FPGA Xilinx Artix-7 et un FPGA Altera Cyclone-V, avec VTPRs, sans VTPRs, avec VTPR et snapshot et avec VTPR et pré-configuration.

Les figures 3.22 et 3.24 présentent le taux d'utilisation des LUTs pour l'implémentation de chaque overlay pour les FPGAs Artix-7 et Cyclone-V, tandis que les figures 3.23 et 3.25 présentent le taux d'utilisation des flip-flops. L'outil Quartus d'Altera n'est pas parvenu à réaliser les synthèses sans VTPRs, ce qui confirme l'avantage des VTPRs dans leur rôle de faciliter (et dans le cas de l'outil Quartus, de permettre) la synthèse physique de l'overlay. Les résultats avec VTPRs ne sont donc fournis que pour l'Artix-7. Le tableau 3.2 présente pour une largeur de

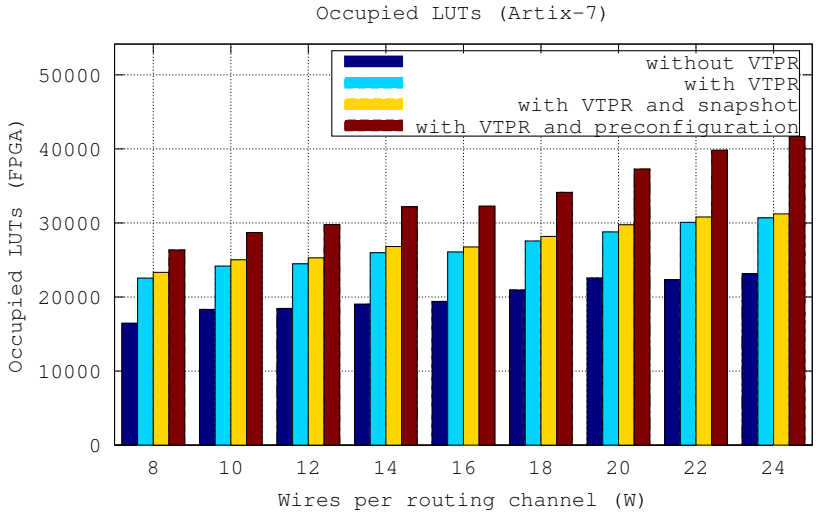


FIGURE 3.22 – Utilisation des LUTs du FPGA hôte Xilinx Artix-7 par l'overlay.

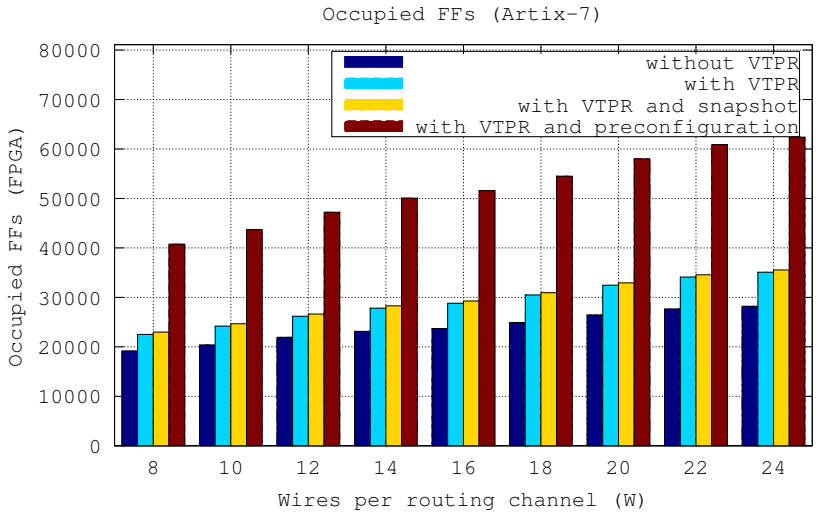


FIGURE 3.23 – Utilisation des flip-flops du FPGA hôte Xilinx Artix-7 par l'overlay.

routage $W = 16$ le surcoût des VTPRs par rapport à un overlay sans VTPR, et les surcoûts en pourcentage du plan de snapshot et du mécanisme de pré-configuration par rapport à une implémentation seulement avec VTPRs.

Le plan de snapshot à un faible surcoût aussi bien en LUTs qu'en flip-flop. Sont coût vient de chacun des registres de snapshot appairés à chaque registre applicatif et des mécanismes de transferts de entre les registre applicatifs et de snapshot ; le coût du snapshot ne varie donc pas en fonction de W.

L'insertion des VTPRs présente un coût important en LUT et en flip-flops. Cependant, ce coût est nécessaire pour permettre une analyse de timing précise de l'application virtuelle et garantir le fonctionnement correct de l'application sur l'implémentation de l'overlay. De plus, les VTPRs facilitent la synthèse physique de l'overlay, et sont même indispensables dans le cas de l'outil Quartus pour que

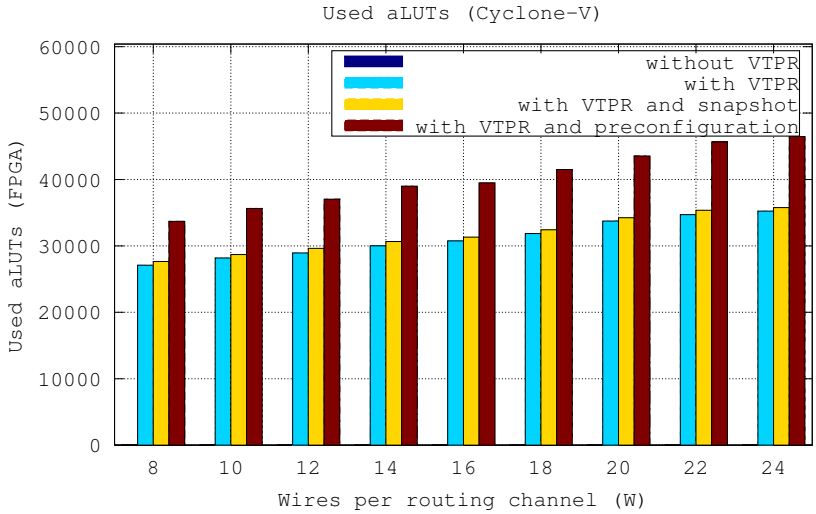


FIGURE 3.24 – Utilisation des aLUTs du FPGA hôte Altera Cyclone-V par l'overlay.

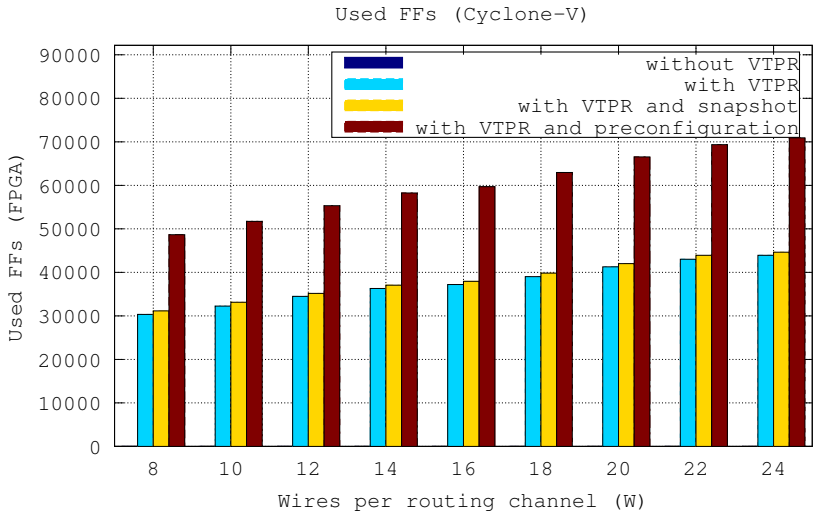


FIGURE 3.25 – Utilisation des flip-flops du FPGA hôte Cyclone-V par l'overlay.

la synthèse aboutisse à une implémentation. Bien que l'insertion des VTPRs ne corresponde qu'à l'insertion de registres additionnels (au niveau de la description RTL de l'overlay), le surcoût de leur implémentation est plus important en LUTs qu'en flip-flops. Ceci peut s'expliquer par le fait qu'en diminuant la longueur des chemins combinatoires dans l'implémentation de l'overlay, l'addition des VTPRs ne permet pas à l'outil de synthèse physique de maximiser l'utilisation de chaque LUT du FPGA hôte. Un VTPR étant attribué à chaque piste de routage, le surcoût des VTPRs croit avec W.

La pré-configuration est la fonctionnalité la plus coûteuse de l'overlay. Dans le cas d'un overlay grain fin comme utilisé ici, le nombre de registres de configuration est important et représente une part importante des ressources consommées par l'overlay. C'est pourquoi la multiplication par deux des registres de configurations impliquée par le mécanisme de pré-configuration entraine un surcoût qui

peut monter jusqu'à 80% de flip-flops additionnelles. La pré-configuration ne mettant pas particulièrement en œuvre de mécanismes logiques, le surcoût en LUTs parait important. Une analyse des rapport de synthèse montre que le surcoût de la pré-configuration est presque entièrement dû à l'utilisation de LUTs "route-thru" par l'outil de synthèse physique. Les LUTs dites route-thru sont des LUTs qui sont utilisées non pas comme éléments logiques mais comme "fils" pour atteindre le registre en sortie de la LUT.

		VTPRs	Snapshot	pré-configuration
Xilinx Artix-7	LUT	34.3%	2.6%	23.7%
	FF	21.6%	1.6%	79.1%
Altera Cyclone-V	LUT	–	1.8%	28.3%
	FF	–	2.0%	60.5%

TABLE 3.2 – Pourcentage du surcoût en ressources pour $W = 16$.

3.5.2 Modélisation du coût

Le temps de synthèse d'un design sur FPGA est relativement long, et s'étend de quelque minute à plusieurs heures, voir plus d'une journée. Ceci peut rendre l'exploration du coût d'un overlay suivant son espace de conception extrêmement longue. Pour pallier ce problème, un modèle de coût peut être réalisé à partir de plusieurs synthèse pour estimer le coût d'un overlay suivant des paramètres pour lesquels il n'a pas encore été synthétisé. Par la suite, ce modèle permet par exemple de déterminer le FPGA minimal requis pour implémenter un overlay suivant des paramètres donnés.

Le but de cette sous-section et de modéliser le coût en ressources d'un overlay en fonction des paramètres de son plan de calcul. Pour ce faire, plusieurs synthèses sont réalisées en faisant varier les paramètres de l'overlay, puis un modèle analytique est établi à partir de ces synthèses. Dans cette expérience, pour observer la régularité des coût mesurés et ainsi évaluer la possibilité de réaliser un modèle de coût réaliste, une synthèse a été réalisée pour chaque point de l'espace de conception. Le modèle vFPGA-flexible étant très flexible, il se prête mal à une exploration exhaustive des ses paramètres : même en se limitant à une dizaine de paramètres haut niveau, en faisant varier chacun de ces paramètres sur quelques valeurs, le nombre de synthèses à réaliser dépasse rapidement plusieurs centaines de milliers. À l'inverse, le modèle vFPGA-restreint ne présente que trois paramètres, ce qui permet de réaliser une synthèse pour chaque point de son espace de conception.

Pour réaliser les mesures, plus de 900 synthèses ont été réalisées sur un serveur. Le flot de synthèse a été automatisé à l'aide du programme GNU Make qui permet de gérer automatiquement la parallélisation des étapes de synthèses indépendantes sur les 64 cœurs du serveur utilisé. De plus, l'utilisation d'un Makefile

permet de combiner les outils de synthèse constructeur avec nos propres scripts et programmes, comme le générateur d'architectures. Pour chaque ensemble de paramètre D (dimension de la matrice), K (nombre d'entrées par LUT) et W (nombre de piste par canal de routage), le code RTL de l'overlay est généré puis instancié dans un top-level. Les outils de synthèse constructeur sont ensuite appelés. Finalement, le rapport de synthèse est archivé. Une fois toutes les synthèse réalisées, un script vient analyser les rapports de synthèse pour extraire les informations pertinentes. Ainsi, cette expérience peut être reproduite à peu de frais (mis à part le temps de calcul processeur) après tout changement du modèle de l'architecture.

Les FPGAs ciblés dans cette expérience sont un Cyclone IV d'Altera et un FPGA Artix-7 de Xilinx. L'Artix-7 appartient à la famille "7 series" de Xilinx, qui inclue aussi les FPGA ZYNQ, Kintex-7 et Virtex-7. Tous les FPGAs de cette famille partagent la même architecture de CLB, slice et réseau de routage. Ainsi, le modèle extrait des mesures de l'Artix-7 devrait s'appliquer à tous les FPGAs de cette famille.

Les dimension de la matrice ont été variées de 10×10 à 16×16 par pas de 2. Le W a été varié de 2 à 10, et K de 2 à 8. Le modèle à ensuite été réalisé à l'aide d'une régression linéaire multiple. En l'absence d'un modèle théorique, les variables explicatives ont été choisies par essais. Il semble normal que le coût de la matrice croisse linéairement avec le nombre de tuiles, soit D^2. K, K^2, W, W^2 et $K \cdot W$ ont été intégrés pour obtenir un modèle polynomial d'ordre 2. 2^K a aussi été ajouté pour prendre en compte la croissance des LUT selon leur nombre d'entrées. Les quatre régressions réalisée présentent des coefficients de détermination allant de 0.989 à 0.999, et des erreurs moyennes allant de 1.2% à 3.1%. Les équations suivantes présentent les modèles analytiques obtenus pour le nombre de LUTs et de flip-flops pour les deux FPGAs ciblés.

$$\begin{aligned} LUT_{Artix7} = -2101 + D^2 \cdot (&11.48 \cdot K + 1.011 \cdot 2^K - 1.968 \cdot K^2 + 11.54 \cdot W \\ &- 0.006038 \cdot W^2 + 0.4413 \cdot K \cdot W) \end{aligned} \tag{3.1}$$

$$\begin{aligned} FF_{Artix7} = -103.4 + D^2 \cdot (&6.132 \cdot K + 1.074 \cdot 2^K - 0.4306 \cdot K^2 + 16.85 \cdot W \\ &- 0.01460 \cdot W^2 + 0.06870 \cdot K \cdot W) \end{aligned} \tag{3.2}$$

$$\begin{aligned} LUT_{CycloneIV} = -462.4 + D^2 \cdot (&8.721 \cdot K + 1.680 \cdot 2^K - 0.2553 \cdot K^2 + 24.48 \cdot W \\ &- 0.02887 \cdot W^2 + 1.329 \cdot K \cdot W) \end{aligned} \tag{3.3}$$

$$\begin{aligned} FF_{CycloneIV} = 419.7 + D^2 \cdot (&4.464 \cdot K + 0.9974 \cdot 2^K + 0.01893 \cdot K^2 + 16.46 \cdot W \\ &- 0.01578 \cdot W^2 + 0.04502 \cdot K \cdot W) \end{aligned} \tag{3.4}$$

La figure 3.26 montre quatre coupes du modèle et des mesures des LUTs et flip-flops suivant les plans $K = 4$ et $W = 12$ pour le FPGA Artix-7. La figure 3.27 montre les mesures et les surfaces du modèle pour l'occupation en LUT de l'Artix-7.

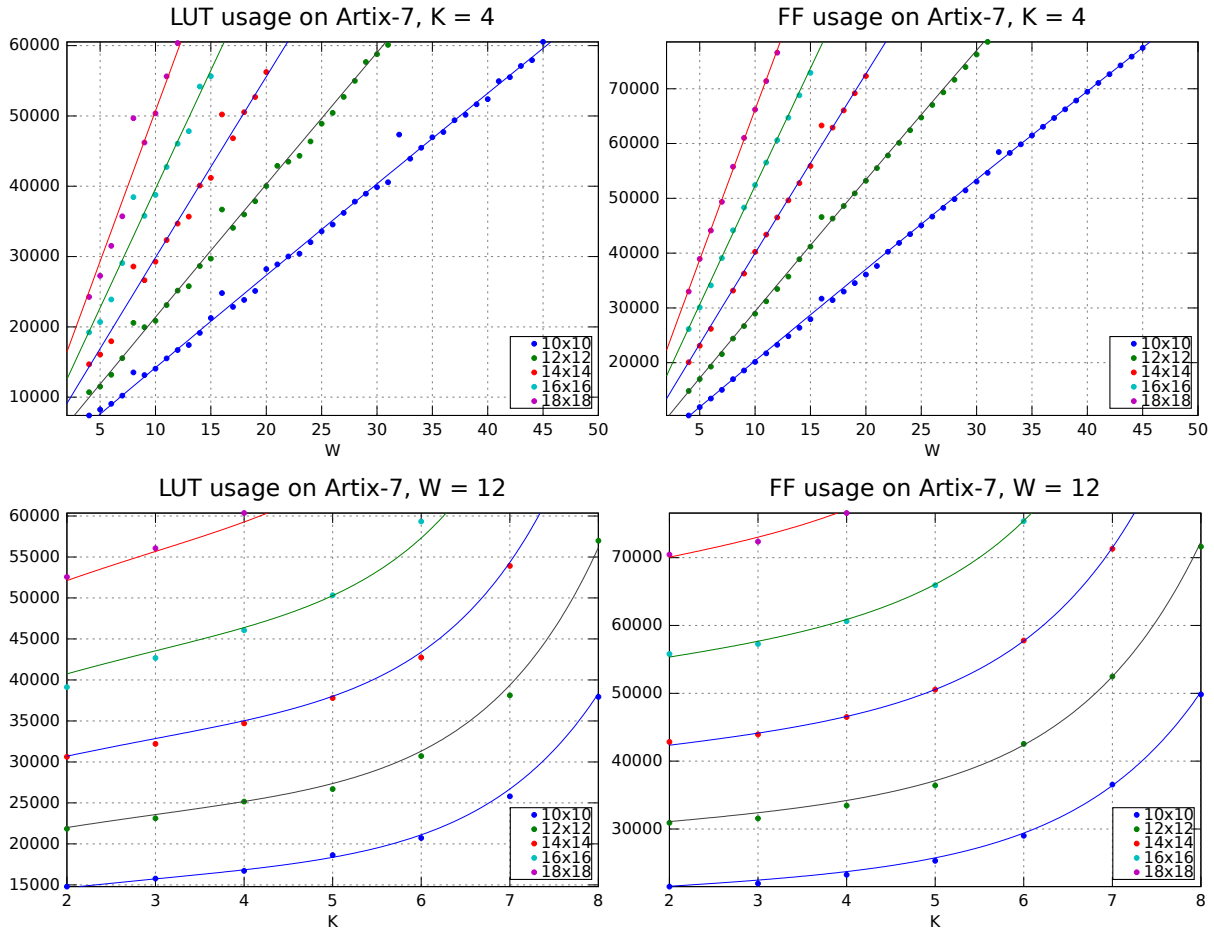


FIGURE 3.26 – Mesures (points) et modèles (lignes) de l'occupation par le vFPGA-restreint en LUTs (gauche) et en flip-flops (droite) sur un FPGA Artix-7, en fixant $K = 4$ (haut) et $W = 12$ (bas) pour des matrices de taille 10×10, 12×12, 14×14, 16×16 et 18×18.

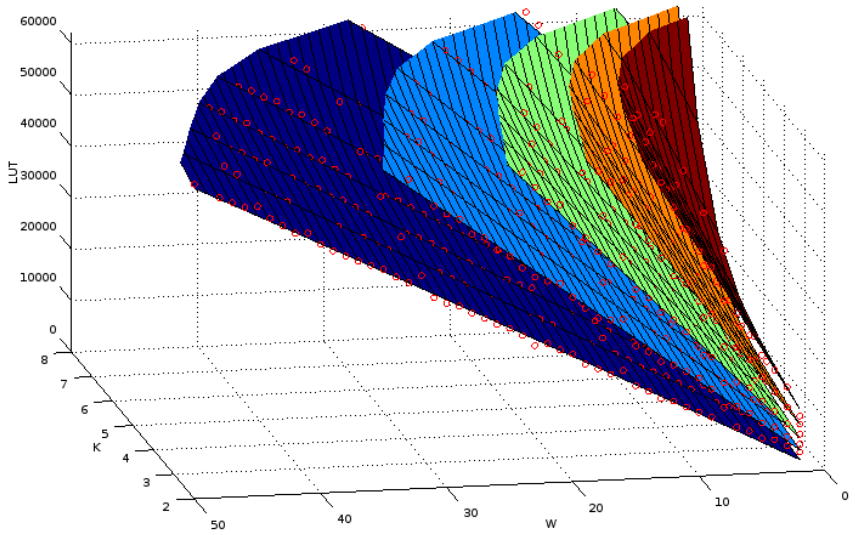


FIGURE 3.27 – Mesures (ronds) et modèles (surfaces) de l'occupation par le vFPGA-restreint en LUTs sur un FPGA Artix-7 suivant K et W pour des matrices de taille 10×10, 12×12, 14×14, 16×16 et 18×18.

3.6 Résumé

Dans ce chapitre ont été présentées l'architecture fonctionnelle d'un overlay correspondant à l'agencement de ces ressources reconfigurables, ainsi que les ressources additionnelles permettant de former un overlay complet. Notamment, l'implémentation de l'horloge applicative et l'ajout d'un plan de snapshot à l'overlay permet de sauvegarder et restaurer l'état d'exécution des circuits applicatifs. Il a ensuite été montré comment implémenter un overlay, et notamment comment rendre cette implémentation synthétisable sur FPGA grâce aux VTPRs, sans que la taille de l'overlay n'influe sur la qualité de la synthèse et ne demande d'intervention manuelle de l'utilisateur lors de la synthèse. Deux méthodes pour générer automatiquement l'implémentation à partir de modèles ont été présentées. Finalement, le coût en ressources du FPGA hôte a été évalué et une méthode permettant de le modéliser suivant les paramètres de l'overlay a été présentée.

Chapitre 4

Intégration des overlays : utilisabilité système

Une fois l'overlay conçu et son implémentation synthétisable générée, il faut pouvoir le déployer sur des plateformes FPGA physiques. Comme nous l'avons vu dans le chapitre 3, un overlay "nu" est une architecture dont l'interface comporte les IOs virtuelles ainsi que les signaux de l'interface de configuration, l'horloge applicative, et éventuellement l'interface du mécanisme de snapshot. L'intégration de l'overlay dans une plateforme FPGA doit donc permettre la gestion de ces interfaces et d'alimenter son plan de calcul en données. Pour pouvoir distribuer l'overlay à des utilisateurs, il faut donc l'intégrer dans un système donnant y accès de l'extérieur du FPGA et permettant de le contrôler et de l'alimenter en données. Ce système doit supporter à la fois la variabilité de l'overlay intégré et la variabilité des plateformes FPGA hôtes utilisées.

La gestion bas niveau de l'overlay – qui correspond à sa configuration, la gestion de l'horloge applicative, des snapshots et de son alimentation en données – est réalisée aux niveaux matériel et logiciel : un ensemble de contrôleurs matériels vient s'interfacer avec l'overlay pour permettre sa gestion par un logiciel que nous appelons *hyperviseur*. L'overlay et ses contrôleurs matériels sont rassemblés en une IP. L'hyperviseur étant logiciel, un processeur est nécessaire pour l'exécuter. Deux cas de figures se présentent :

— Soit le FPGA est relié à un processeur physique (par exemple une carte FPGA connectée en PCI dans un ordinateur), auquel cas l'hyperviseur peut être exécuté sur le processeur de l'ordinateur hôte. Ce cas est illustré figure 4.1 à gauche.

— Soit le FPGA est utilisé seul en mode "standalone", c'est-à-dire qu'il est utilisé seul sans processeur et qu'il est directement relié au réseau. Dans ce cas, l'hyperviseur doit être exécuté sur le FPGA sur un processeur softcore implémenté via les ressources reconfigurables du FPGA. Ce cas est illustré figure 4.1 à droite.

Ce chapitre présente l'intégration de l'overlay en tant qu'IP, et comment cette IP est intégrée dans un FPGA. La partie bas niveau de l'hyperviseur est ensuite pré-

sentée.

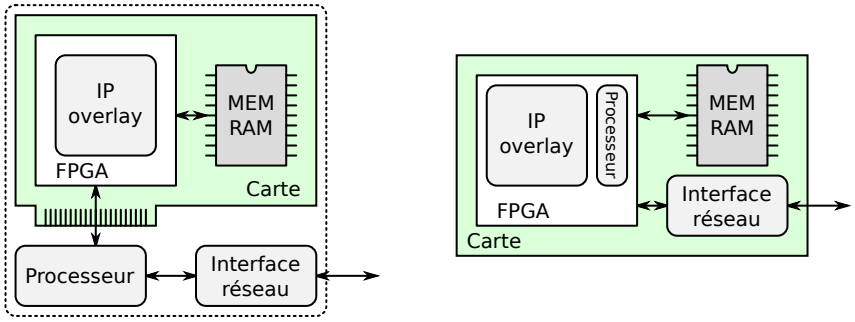


FIGURE 4.1 – Exemples d'intégration de l'overlay, l'hyperviseur est exécuté sur le processeur dans le FPGA (droite) ou à l'extérieur (gauche)

4.1 De la matrice à l'IP

L'intégration de l'overlay peut varier d'une plateforme à une autre, et l'overlay lui-même peut être amené à varier (par exemple pendant la phase de conception). Un changement des paramètres de l'overlay ne doit pas demander une adaptation de son intégration, et à l'inverse, un changement d'intégration ne doit pas demander une adaptation des contrôleurs matériels de l'overlay. Pour cela, l'overlay et ses contrôleurs ont été rassemblés en une IP : l'IP overlay. Pour pouvoir changer l'overlay utilisé sans impacter le reste du design, l'interface de l'IP overlay est fixe quels que soient les paramètres de la matrice et de ses contrôleurs associés.

De même, la position et la signification de chaque bit des registres de configuration des contrôleurs instanciés dans l'IP ne dépendent pas des paramètres de l'overlay (cf fig. 4.2). Afin que le code de l'hyperviseur n'ait pas à être adapté lorsque les paramètres de la matrice ou de l'IP overlay sont modifiés, les trois premiers registres occupant l'espace mémoire de l'interface esclave de l'IP sont des constantes indiquant les paramètres de la matrice et de l'IP overlay lors de leurs génération. L'hyperviseur utilise ces registres (que nous appelons *registres de présentation*) pour activer ou non certaines de ses fonctionnalités suivant la configuration de l'IP.

Pour que l'hyperviseur puisse accéder à l'overlay et le contrôler, différents contrôleurs matériels doivent êtres couplés avec l'overlay pour former l'IP overlay (cf fig. 4.2) :

— Un contrôleur de configuration, interfacé avec le plan de configuration de l'overlay. Il permet d'écrire le bitstream virtuel pour configurer le plan calcul.
— Un contrôleur de snapshot, interfacé avec le plan de snapshot de l'overlay. Il permet de lire et écrire le contexte d'exécution du plan de calcul.
— Un contrôleur d'horloge. Il génère le signal d'horloge applicative, permet de le mettre en pause, de l'activer ou de le stopper, de modifier dynamiquement sa fréquence et de générer un nombre paramétrable de cycles d'horloge applicative avant de la stopper et de générer un signal d'interruption.

— Un contrôleur DMA. Il permet d'alimenter les IOs virtuelles de l'overlay en données d'entré et sauvegarder les données produites en sortie. Le contrôleur DMA peut générer une interruption indiquant qu'un buffer d'entrée est vide ou qu'un buffer de sortie est plein.
— Un contrôleur de mémoire virtuelle. Suivant l'implémentation des mémoires virtuelles (cf 3.2.5), ce contrôleur peut jouer différents rôles. Si le contenu des mémoires virtuelles est physiquement placé dans des mémoires externes à l'IP overlay, le contrôleur assure l'accès à ce contenu depuis l'application exécutée sur l'overlay. Si le contenu des mémoires virtuelles est physiquement placé dans des mémoires internes à l'IP overlay, le contrôleur assure l'accès à ce contenu depuis l'extérieur de l'IP overlay pour permettre la sauvegarde et la restauration de ces mémoires.

Pour simplifier l'intégration de l'IP overlay dans différents systèmes, l'interface de l'IP se présente sous forme de bus mémoire. Que l'overlay soit utilisé en mode standalone ou couplé avec un processeur externe au FPGA, l'accès aux registres de configuration de ces contrôleurs se fait via ce bus. De même, le contrôleur DMA et éventuellement le contrôleur de mémoire virtuelle ont besoin d'accéder à une ou plusieurs mémoires externes à l'IP overlay, et donc d'être maîtres du bus. L'interface de l'IP overlay est ainsi composée d'une connexion de bus esclave et d'une connexion de bus maître (voir fig. 4.2). L'ensemble des registres de configuration des différents contrôleurs présents dans l'IP overlay sont mappés sur l'interface esclave du bus, tandis que les contrôleurs DMA et de mémoire virtuelle accèdent aux mémoires externes à l'overlay via l'interface maître du bus. De plus, pour permettre aux contrôleurs de notifier l'hyperviseur d'un évènement, un signal d'interruption est aussi présent sur l'interface de l'IP overlay.

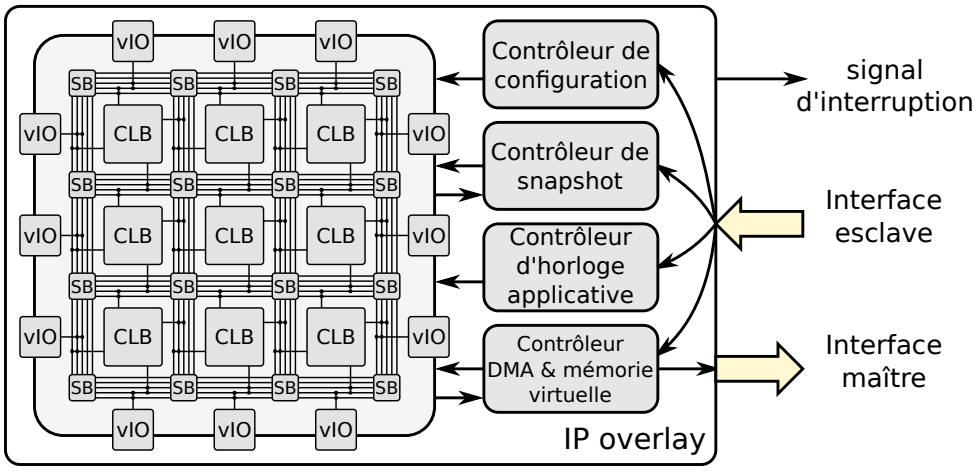


FIGURE 4.2 – L'overlay est couplé à des contrôleurs matériels pour former l'IP overlay.

Ces contrôleurs doivent pouvoir s'adapter aux besoins de l'utilisateur. Notamment, la largeur des mots DMA et des mots de données et d'adresse pour la mémoire virtuelle doivent être choisis en fonction de la classe d'application visée (par exemple des mots DMA 8-bits pour des applications traitant des caractères, ou 32-bits pour des applications traitants des images). Pour cela, six paramètres permettent de modifier la génération de l'IP et des contrôleurs :

— la possibilité ou non d'activer un signal d'interruption à partir du front montant d'une sortie virtuelle de la matrice ;
— la présence ou non d'un contrôleur DMA ;
— la largeur des mots de données DMA ;
— la présence ou non d'un contrôleur de mémoire virtuelle ;
— la largeur des mots de données de la mémoire virtuelle ;
— la largeur du mot d'adresse de la mémoire virtuelle.

Les contrôleurs étant intimement liés aux interfaces de l'overlay, le code HDL des contrôleurs et de l'IP est généré en même temps que celui de la matrice. Aussi, pour permettre à l'hyperviseur de déterminer les possibilités de l'IP, ces paramètres sont écrits dans les registres de présentation lors de la génération de l'IP.

4.1.1 Les contrôleurs de configuration et de snapshot

Le contrôleur de configuration permet de pousser le contenu d'un registre (de 32 bits) dans le serpentin de configuration de l'overlay : lorsque l'interface esclave de l'IP réalise une écriture à l'adresse de ce registre, le signal `config_valid` de l'interface de configuration de l'overlay est activé pour un cycle d'horloge, tandis que le signal `config_in` reçoit le mot écrit depuis le bus, ce qui a pour effet de pousser chacun des 32 bits de données dans les chaînes de configuration du plan de configuration. Si l'overlay a été généré en divisant le plan de configuration en 32 chaînes de registres à décalage, il est donc possible d'écrire 32 bits effectifs de configuration par cycle de bus. La figure 4.3 illustre un même ensemble de registre de configuration chainé de deux manières différentes : de manière linéaire (une chaîne de configuration), et en utilisant trois chaînes de configuration.

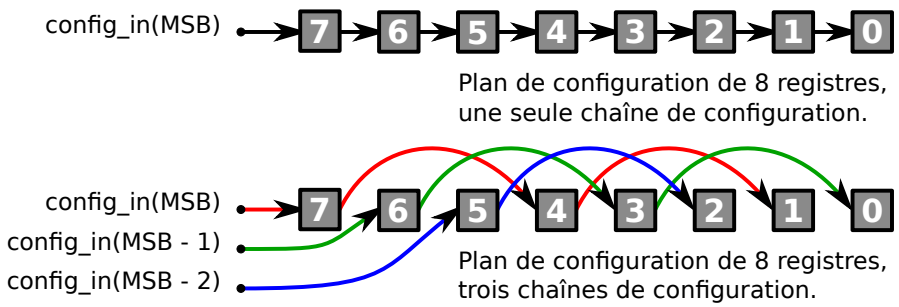


FIGURE 4.3 – Deux chaînages possibles du plan de configuration

Lors de l'écriture du bitstream virtuel via le contrôleur de configuration dans un overlay dont le plan de configuration est divisé en C chaînes, les C bits du dernier mot écrit occupent les C premiers bits du plan de configuration. Pour formater le bitstream virtuel (le fichier de configuration de l'overlay) à partir du bitstream conceptuel (c'est-à-dire la chaîne de bits correspondant à la vision linéaire du plan de configuration) pour un overlay qui comporte C chaînes de configuration, il faut donc :

— concaténer des bits à la fin du bitstream conceptuel de façon à ce que le nombre de bit total soit un multiple de C, la valeur de ces bits n'importe pas

car lorsque le dernier mot du bitstream sera chargé, ces bits de "padding" seront poussés hors du plan de configuration ;
— diviser la chaîne de bit obtenue en mots de C bits ;
— inverser l'ordre de la suite de mots obtenue ;
— concaténer la suite de mot obtenu en une nouvelle chaîne de bit que l'on peut écrire dans un fichier : le bitstream virtuel.

Le bitstream virtuel ainsi formaté peut ensuite être lu linéairement par l'hyperviseur depuis une mémoire, par morceaux de C bits, chacun de ces morceaux étant envoyé au contrôleur de configuration.

Ce formatage du bitstream virtuel peut être réalisé lors de la synthèse de l'application ; cela implique que l'outil de synthèse virtuelle ait connaissance du nombre de chaînes de configuration du plan de configuration de l'overlay. Cependant, deux overlays ayant le même plan de calcul mais étant implémentés avec un nombre de chaînes de configuration différent dans leur plan de configuration ne demandent pas le même formatage de bitstream virtuel, bien que leurs bitstreams conceptuels soient compatibles. Si le nombre C de chaînes de configuration n'est pas connu au moment de la synthèse virtuelle, l'hyperviseur doit réaliser le formatage du bitstream virtuel à la volé, en se basant sur le nombre C indiqué dans les registres de présentation de l'overlay.

Le fonctionnement du contrôleur de snapshot est le même que celui du contrôleur de configuration, mais il permet en plus d'extraire le contenu du plan de snapshot : lorsque l'interface esclave de l'IP réalise une lecture à l'adresse du registre de données du contrôleur de snapshot, les derniers bits des chaînes de snapshot sont renvoyés à l'interface et les chaînes de snapshot sont décalées d'un bit. Un snapshot extrait par lectures successives du registre de données du contrôleur de snapshot peut par la suite être chargé tel quel dans l'overlay par écritures successives dans ce même registre. L'hyperviseur peut utiliser les registres de présentation de l'IP pour déterminer le nombres de registres applicatifs de la matrice et en inférer le nombre de lectures/écritures à effectuer pour extraire/charger l'intégralité du snapshot.

4.1.2 Le contrôleur d'horloge

Comme nous le verrons en 5.4, le chemin critique des circuits applicatifs synthétisés sur l'overlay est propre à chaque application. Il en découle que la fréquence de l'horloge applicative Clk_{app} doit pouvoir être modifiée pour chaque application. Le contrôleur d'horloge génère l'horloge applicative de l'overlay et permet de modifier sa fréquence via un registre de contrôle.

Via son registre de contrôle, le contrôleur permet aussi de stopper ou d'activer l'horloge applicative. Il comporte aussi un deuxième compteur configurable qui permet d'activer l'horloge applicative pour X cycles applicatifs, et de générer une interruption lorsque les X cycles ont été réalisés. Il est ainsi possible de stopper Clk_{app} avant de configurer l'overlay et de charger un snapshot, puis une fois que

l'overlay est prêt, de lancer à nouveau l'horloge applicative.

Comme nous le verrons dans la sous-section qui suit, la matrice de l'overlay communique avec le contrôleur DMA et le contrôleur de mémoire virtuelle via ces IOs virtuelles, suivant une poignée de main (handshaking) : l'application fait une requête, le contrôleur lui indique quand celle-ci est servie en activant un signal d'acquittement pendant un cycle d'horloge applicative. Cependant, dans le cas où l'application virtuelle est stoppée après qu'elle ai effectué une requête mais juste avant que le contrôleur n'active son acquittement, l'application sera inactive lorsque l'acquittement sera activé, et ne le prendra donc pas en compte. Une fois restaurée, l'application attendra indéfiniment l'acquittement qui ne viendra jamais puisque la requête a déjà été acquittée. Pour éviter cette situation (similaire à un "deadlock"), les contrôleurs DMA et de mémoire virtuelle indiquent au contrôleur d'horloge si un cycle de handshaking est en cours. Lorsque Clk_{app} est stoppée via le registre de contrôle ou que les X cycles applicatifs ont été réalisés, le contrôleur d'horloge continue à générer des cycles applicatifs tant qu'un cycle de handshaking est en cours.

Comme il a été vu en 3.2.3, le signal d'horloge applicative est généré à partir d'un compteur. Ce signal n'est actif qu'un cycle d'horloge physique sur N, N étant la valeur écrite dans le registre de contrôle. Du fait que N est une valeur entière, le profile de la fréquence de Clk_{app} est en $\frac{1}{N}$, donc les valeurs possibles pour la fréquence de l'horloge applicative sont inégalement réparties (elles ne sont pas espacées deux à deux d'une même valeur). Cependant ceci n'est pas un problème, car comme il sera vu en 5.4, l'information de timing d'une application virtuelle remontée par l'outil de synthèse virtuel est la valeur N. Les fréquences qu'il est possible de générer pour l'horloge applicative correspondent donc exactement aux fréquences que peuvent requérir les applications virtuelles.

4.1.3 Le contrôleur DMA et de mémoire virtuelle

Pour simplifier le développement de nos prototypes d'overlay et la chaîne d'outils qui les supportent, nos prototypes d'overlay n'implémentent pas l'équivalent de mémoires bloc RAM comme primitives de leur plan de calcul, mais offrent au circuit applicatif la possibilité d'accéder à une mémoire via leurs IOs virtuelles. Nous appelons cette mémoire une mémoire virtuelle (cf 3.2.5). De manière similaire à la virtualisation de la mémoire par un système d'exploitation, l'espace mémoire présenté à l'application virtuelle ne correspond pas nécessairement à l'espace mémoire utilisé dans la mémoire physique. De même que pour les buffers DMA, le contenu de cette mémoire virtuelle n'est pas stocké dans l'IP overlay elle-même. Ainsi, les contrôleurs DMA et de mémoire virtuelle de l'overlay accèdent à leurs mémoires via l'interface maître de l'IP overlay. Les buffers DMA et le contenu de la mémoire virtuelle peuvent être stockés sur n'importe quelle mémoire physique accessible via le bus auquel est attaché l'interface maître de l'IP overlay, que ce soit une mémoire externe ou interne au FPGA, les mémoires internes étant plus rapide et les mémoires externes fournissant une plus grande capacité.

Comme les mémoires physiques sont accédées par ces contrôleurs via un bus, la latence du temps d'accès à ces mémoires n'est pas fixe et varie en fonction du type de mémoire adressée et du taux d'occupation du bus (qui peut être partagé par plusieurs maîtres). Comme nous l'avons vu en 3.2.5, pour gérer cette latence variable, l'interface entre ces mémoires et l'application exécutée sur l'overlay peut soit implémenter un handshaking, soit stopper l'horloge applicative tant que la requête n'a pas eu de réponse. Dans notre implémentation, nous avons choisi la solution du handshaking, car stopper l'horloge applicative fige l'ensemble de l'application, y compris des parties qui peuvent éventuellement s'exécuter indépendamment d'un transfert de donné via l'interface DMA ou mémoire.

La deuxième conséquence au fait que les deux contrôleurs accèdent à leurs mémoires via le même bus (et la même interface à ce bus) est qu'ils ne peuvent pas accéder simultanément à leurs mémoires respectives. Pour cette raison, le contrôleur DMA et le contrôleur de mémoire virtuelle ont été implémentés en une même machine d'état.

Pour permettre d'accéder aux IOs virtuelles de la matrice, celles-ci sont mappées sur des registres accessibles dans l'espace mémoire de l'IP overlay. Les registres correspondant aux sorties virtuelles sont accessibles en lecture seule, tandis que les registres correspondants aux entrées virtuelles sont accessibles en lecture/écriture. Pour permettre aux applications n'utilisant pas la DMA ou la mémoire virtuelle d'utiliser les IOs connectées à ces deux contrôleurs en tant que simples IOs (et donc augmenter le nombre d'IOs utilisables par ses applications), deux bits dans le registre de contrôle du contrôleur permettent d'activer ou non de manière indépendante la DMA et la mémoire virtuelle. Lorsque la DMA et/ou la mémoire virtuelle sont activées, les entrées virtuelles utilisées pour communiquer avec le contrôleur DMA/mémoire virtuelle ne sont plus pilotées par les bits leur correspondant des registres d'IO mappés en mémoires, mais par les signaux issus du contrôleur. Pour que l'utilisateur puisse réaliser des circuits applicatifs utilisant la DMA et/ou la mémoire virtuelle, les positions des IOs du plan de calcul de l'overlay utilisées pour communiquer avec le contrôleur doivent être fixées et documentées lors de la génération de l'IP overlay. Ainsi, l'utilisateur peut contraindre la position des signaux de l'interface top-level de son application pour correspondre aux positions attendues par le contrôleur.

Les signaux d'interface utilisés entre le circuit applicatif et le contrôleur (DMA) sont les suivants :

— `DMA_IN_REQ_O` : l'application active ce signal pour demander un mot de donnée.
— `DMA_IN_VALID_I` : le contrôleur active ce signal pendant un cycle d'horloge applicative pour notifier l'application que le signal DMA_DAT_IN_I est valide. Lorsque ce signal est activé, l'application doit désactiver le signal `DMA_IN_REQ_O` si elle ne veut pas initier un deuxième transfert.
— `DMA_DAT_IN_I` : le mot de donnée lu depuis la DMA, validé par le signal `DMA_IN_VALID_I`. Ce signal reste inchangé jusqu'à la prochaine requête DMA de lecture.
— `DMA_OUT_VALID_O` : l'application active ce signal pour demander au contrô-

leur de sauvegarder un mot de donnée.
- DMA_OUT_ACK_I : le contrôleur active ce signal pendant un cycle d'horloge applicative pour notifier l'application que le mot a bien été sauvegardé. L'application peut alors désactiver le signal DMA_OUT_VALID_O à moins qu'elle ne veuille initier un autre transfert dans la foulée.
- DMA_DAT_OUT_O : le mot de donnée écrit par l'application.

Les signaux d'interface utilisés entre le circuit applicatif et le contrôleur (mémoire virtuelle) sont les suivants :
- MEM_CYC_O : l'application active ce signal pour initier un transfert. Les signaux MEM_WE_O, MEM_ADDR_O et éventuellement MEM_DAT_O doivent être positionnés avant ou en même temps que MEM_CYC_O est activé.
- MEM_WE_O : l'application demande une lecture ou une écriture.
- MEM_ADDR_O : adresse dans l'espace de la mémoire virtuelle.
- MEM_ACK_I : le contrôleur notifie l'application que le mot MEM_DAT_O a bien été écrit ou que le mot MEM_DAT_I est valide. L'application doit alors désactiver le signal MEM_WE_O à moins qu'elle ne veuille initier un autre cycle.
- MEM_DAT_I : mot lu depuis la mémoire.
- MEM_DAT_O : mot à écrire dans la mémoire.

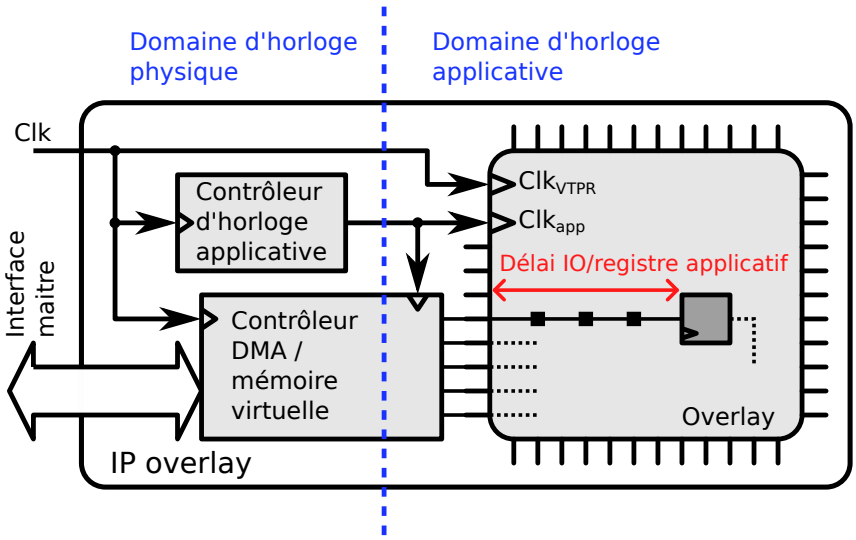


FIGURE 4.4 – Le contrôleur DMA/mémoire virtuelle est à l'interface des domaines d'horloge physique et applicative.

La fréquence d'horloge applicative $f_{Clk_{app}}$ est propre à chaque application, et est donc différente de la fréquence d'horloge de l'interface et des contrôleurs de l'IP overlay. Lors des transferts DMA/mémoire virtuelle, le contrôleur doit donc synchroniser les signaux entre l'interface maître de l'IP et les signaux issus du circuit applicatif implémenté sur l'overlay. Ainsi, le contrôleur doit attendre que le signal de l'horloge applicative soit actif pour positionner/échantillonner les signaux vers/-depuis les IOs de l'overlay. Cependant, il faut aussi prendre en compte le délai de propagation dans le routage de l'overlay (en terme de nombre de VTPRs) entre les IOs virtuelles reliées au contrôleur et les registres applicatifs qui y sont connectés. Ce délai dépend du placement et routage du circuit applicatif sur l'overlay, il est illustré en rouge dans la figure 4.4. Comme il sera vu en 5.4, la fréquence

$f_{Clk_{app}}$ maximale retournée par l'outil de synthèse virtuelle prend en compte le délai de propagation maximum entre registres applicatifs et entrées/sorties primaires. Le délai $D_{IO/registre}$ est donc au maximum égale à la période de $f_{Clk_{app}}$. Lorsque le registre applicatif pilotant un des signaux de requête (`DMA_IN_REQ_O`, `DMA_OUT_VALID_O` ou `MEM_CYC_O`) change d'état, le contrôleur n'en est notifié qu'au prochain cycle de Clk_{app}. Lorsque le contrôleur active un des signaux d'acquittement (`DMA_IN_VALID_I`, `DMA_OUT_ACK_I` ou `MEM_ACK_I`), il doit donc attendre non pas un mais deux cycles d'horloge applicative avant de pouvoir à nouveau prendre en compte les signaux de requêtes : le temps que le signal d'acquittement arrive jusqu'à son registre applicatif de destination, produise un changement sur le signal de requête et que celui-ci revienne jusqu'au contrôleur. Dans le cas où la fréquence applicative est basse, ce délai supplémentaire d'un cycle d'horloge applicative peut devenir non négligeable par rapport à la bande passante de la mémoire physique, et peut donc limiter la bande passante des contrôleurs mémoire et DMA par rapport à celle offerte par la mémoire physique.

Le contrôleur possède deux registres par canal DMA pour indiquer l'adresse de début du buffer, et un pour indiquer l'adresse de fin. À chaque transfert, le registre d'adresse de début est incrémenté de la taille d'un mot DMA par le contrôleur. Lorsque la valeur du registre d'adresse de début atteint celle du registre d'adresse de fin, le contrôleur génère une interruption et désactive le canal DMA. La mémoire virtuelle est gérée par un unique registre qui représente l'adresse d'offset de l'espace mémoire réservé à la mémoire virtuelle dans l'espace mémoire du bus maître de l'IP.

Ainsi, ce contrôleur permet d'alimenter l'overlay en données et lui fournit une interface de mémoire virtuelle. L'interface avec le plan de calcul est de type bus, pour permettre le stockage physique des données sur une mémoire indépendante de l'IP overlay. Le choix du type de mémoire physique utilisée est effectué lors de l'intégration de l'IP overlay. Ce contrôleur permet une gestion dynamique par l'hyperviseur, qui peut à tout moment modifier les adresses physiques des buffers DMA et de l'espace mémoire physique sur lequel est mappé l'espace mémoire virtuel.

4.1.4 Interruption générée par l'application

Les différents contrôleurs de l'IP overlay peuvent activer le signal d'interruption de l'IP pour notifier l'hyperviseur qu'un évènement a eu lieu qui requiert son intervention, comme par exemple qu'un buffer DMA a été consommé. Lorsque l'hyperviseur reçoit l'interruption, la lecture des registres de statut de l'IP lui indique quel contrôleur et quel évènement a généré l'interruption, ce qui lui permet de réaliser l'action adéquate pour réagir à l'évènement.

Cependant, il peut aussi arriver que le circuit applicatif ait besoin de notifier l'hyperviseur d'un évènement. Pour cela, un contrôleur d'interruption applicative scrute une des sorties virtuelles de la matrice à chaque cycle d'horloge applicative. Si l'interruption applicative est autorisée par l'hyperviseur (via un bit dans un

registre de contrôle de l'IP), l'interruption est générée lorsque cette sortie passe à 1 alors qu'elle était à 0 au cycle applicatif précèdent.

Un cas d'usage de cette interruption applicative est lors de la phase de conception des applications, lors du débogage, pour mettre en place des points d'arrêt matériels. Les assertions sont un moyen puissant pour de vérification du comportement des applications en développement, elles sont décrites dans un langage tel que PSL (Propoerty Specification Language) [59]. Lors de la simulation de l'application, les assertions sont vérifiées à chaque pas de simulation. Cependant, le temps de simulation peut limiter les scénarios de simulation. Ainsi, synthétiser les mécanismes d'assertion avec l'application [60] permet d'exécuter matériellement (et donc plus rapidement) les scénarios, et permet donc une plus grande couverture lors de la vérification. Dans le cas des overlays, l'interruption applicative peut ainsi être utilisée pour notifier l'hyperviseur qu'une des assertions synthétisées avec l'application a échoué. Lors de cet évènement, l'horloge applicative est stoppée et le snapshot est extrait de façon à permettre le concepteur de l'application d'analyser quelle assertion a échoué et la cause de cet échec. Plus généralement, l'interruption applicative permet de mettre en place des points d'arrêt matériels.

4.1.5 Réorganisation des IOs

Si l'on désire synthétiser un circuit applicatif dans le but d'ensuite l'exécuter sur l'architecture ciblée, il est nécessaire de contraindre le placement des signaux applicatifs de l'interface du top-level sur des IOs spécifiques de l'architecture. C'est le cas pour l'usage de l'IP overlay : les signaux applicatifs communicant avec le contrôleur DMA/mémoire virtuelle et le signal générant une interruption doivent être positionnés lors de la synthèse virtuelle sur les IOs virtuelles auxquelles sont câblées les signaux correspondants du contrôleur DMA/mémoire virtuelle et d'interruption. La contrainte du placement d'éléments de la netlist applicative sur des ressources spécifiques du plan de calcul de l'architecture cible est donc une fonctionnalité nécessaire des outils de placement et routage dont le but est de produire un circuit physiquement exécutable et exploitable (par exemple Madeo [48] ou les outils de synthèse des vendeurs d'FPGAs tels que Quartus et ISE). Les éléments dont le placement est contraint sont placés en premier suivant ces contraintes, et ne sont pas déplacés par la suite par les heuristiques de l'outil. Celles-ci optimisent le placement des éléments non contraints de façon à minimiser l'occupation ou le chemin critique du circuit final en prenant en compte tous les éléments (contraints et non contraints).

Bien que VPR [43] soit un outil de placement et routage puissant lors de la phase d'exploration des architectures fonctionnelles des overlays, un inconvénient de son utilisation est qu'il ne permet pas de contraindre la position des IOs de manière flexible lors de la synthèse virtuelle. En effet, VPR est un outil de placement et routage qui a été développé en premier lieu pour évaluer et comparer des architectures reconfigurables, et non pour synthétiser des circuits dans le but de les exploiter ultérieurement. Bien que la synthèse de VPR soit de qualité et qu'elle

aboutisse à des circuits fonctionnels, la gestion fine des contraintes de placement des IO n'a pas été implémentée dans VPR (du moins à la version 7.0). Comme nous le verrons au chapitre 5, notre souhait est de permettre l'utilisation de plusieurs outils de synthèse virtuelle avec les overlays, par exemple pour pouvoir expérimenter avec différents outils, ou réutiliser des outils déjà existants dans un nouveau flot de synthèse.

Dans [51], les auteurs intègrent ZUMA dans un SoC et utilisent VPR pour synthétiser les applications. Ils ont rencontré le problème de placement des IOs avec VPR, et proposent deux solutions. La première consiste à ajouter autour des IOs virtuelles de la matrice une couche de réorganisation des IOs qui permet de permuter les IOs de la matrice aux positions attendues par l'utilisateur. Cette couche est composée de larges multiplexeurs configurables, leur taille croît donc en $O(n^2)$ avec les dimensions de la matrice, ce qui consomme des ressources supplémentaires.

Le flot normal de VPR est de packer les éléments logiques de la netlist d'entrée en CLB et de packer les IOs en IO block (car un IO block peut contenir plusieurs IOs). Ensuite les CLBs et les IO blocks sont placés, puis le design est routé. La deuxième solution proposée par les auteurs de [51] consiste à interrompre le flot de VPR après la phase de packing, et d'utiliser une option de VPR permettant d'indiquer à l'outil les contraintes de placement des IO blocks avant de continuer par le placement et routage. Cette solution permet de s'affranchir de la couche de réorganisation des IOs.

Le problème de la deuxième solution est que VPR ne permet de contraindre que les IO blocks et non les IOs individuellement. Il n'est donc pas possible de contraindre précisément les IOs d'un design via VPR si l'architecture comporte des IO blocks de plus d'une IO. Or c'est toujours le cas dans notre implémentation de l'overlay. En effet, dans un FPGA classique (physique), les pads d'entrée/sortie sont soit utilisés en entrée, soit en sortie, et sont donc chacun relié à une entrée et une sortie interne, qui sont utilisées exclusivement. Dans le cas d'un overlay, qui est une architecture implémentée sur un FPGA physique, il n'y a pas besoin de pads d'entrées/sorties configurables qui demanderaient de simuler de la circuiterie à trois états, il est plus simple de n'exposer que des IOs qui ne soient que entrées ou que sorties. Ainsi, chaque IO block consiste donc au minimum en une paire d'IOs : une entrée et une sortie. Donc si VPR assemble dans un même IO block une entrée et une sortie, la deuxième solution ne permet pas de dissocier l'entrée et la sortie pour les placer dans des IO blocks différents.

Pour permettre de contraindre finement le positionnement des signaux toplevels applicatifs sur les IOs de l'overlay tout en utilisant un outil qui ne prend pas en compte ces contraintes lors de la synthèse virtuelle (comme VPR), une couche de permutation des IOs a été ajoutée. Celle-ci est optionnelle et sa présence doit être spécifiée lors de la génération de l'IP overlay. Il s'agit d'un contournement permettant l'utilisation telle quelle de VPR, c'est-à-dire sans modification du code source de l'outil pour ajouter la prise en compte des contraintes de placement des IOs lors de la synthèse. Le coût matériel des ressources additionnelles que cette solution demande (qui croît en $O(n^2)$ avec le nombre d'IO) ne permet pas de justi-

fier son utilisation dans le cadre de l'exploitation d'overlays, la solution naturelle étant d'implémenter la gestion des contraintes dans VPR, ou d'utiliser un outil qui le permette (tel que Madeo).

Pour rendre transparente cette couche de mapping reconfigurable des IOs de la matrice sur les IOs de l'overlay, et de permettre de n'avoir qu'un bitstream virtuel par circuit applicatif, la configuration des multiplexeurs de réarrangement des IOs est chaînée à la suite de la configuration de la matrice dans le plan de configuration. Une phase est rajoutée dans le flot de synthèse virtuelle pour extraire le placement des IOs par VPR pour ensuite calculer la configuration de la couche de réarrangement suivant un fichier de contraintes de placement d'IO fourni par l'utilisateur.

4.2 De l'IP au système sur carte

L'IP overlay est donc un module matériel portable, qui comporte une interface fixe (une interface bus esclave, une interface bus maître, et un signal d'interruption), et simplifie l'intégration d'overlay sur une plateforme. L'interface bus qui a été choisie pour l'IP overlay est l'interface Wishbone [61], car il s'agit d'une interface bien documentée, placée dans le domaine public, et relativement simple (il est aisé d'écrire un adaptateur Wishbone vers un autre type de bus tel qu'Avalon ou AXI). De plus, l'IP overlay abstrait l'accès à l'overlay en fournissant une interface de contrôle fixe elle aussi quels que soient les paramètres de la matrice de l'overlay, via des registres mappés en mémoire. Cependant, le contrôle de l'IP overlay reste bas niveau, et dépend du système dans lequel il s'intègre, par exemple des espaces mémoire et leurs adresses qui sont disponibles pour les canaux DMA et la mémoire virtuelle. La deuxième couche d'abstraction de l'overlay est une couche d'abstraction logicielle que nous appelons l'hyperviseur. Cet hyperviseur est lui-même divisé en deux niveaux d'abstraction : une couche bas niveau qui correspond aux appels systèmes accédant aux registres de configuration de l'IP overlay, et une couche de plus haut niveau que nous verrons au chapitre 6, qui repose sur ces appels systèmes pour exécuter les requêtes de l'utilisateur.

Nous appelons *nœud de calcul* l'ensemble formé par l'overlay, l'hyperviseur ainsi qu'une mémoire locale. L'utilisateur accède au nœud de calcul via une interface réseau. Ainsi, le nœud de calcul abstrait la plateforme physique qui l'implémente en fournissant un accès à l'hyperviseur via un protocole réseau fixe, défini et documenté. L'implémentation physique d'un nœud de calcul requiert donc un FPGA, un processeur pour exécuter l'hyperviseur, de la mémoire RAM accessible par le processeur et l'overlay, ainsi qu'une interface réseau. L'un des buts de l'overlay et de son abstraction en nœud de calcul est la portabilité des circuits applicatifs ainsi qu'une gestion unifiée pour un ensemble de matériel physique hétérogène. Or, toutes les plateformes susceptibles d'implémenter un overlay ne comportent pas nécessairement de processeur, qu'il soit discret ou embarqué dans le FPGA. Les deux sous parties qui suivent présentent respectivement l'intégration de l'IP

overlay dans une plateforme comportant un processeur physique, et dans une plateforme n'en comportant pas.

4.2.1 Intégration de l'IP overlay avec processeur physique

Aspect matériel

Lorsqu'un processeur physique est présent dans la plateforme, il peut être soit externe au FPGA, soit embarqué de manière fixe dans le FPGA avec un ensemble de périphériques. Si le processeur est embarqué dans le FPGA, comme illustré figure 4.5, alors il a un accès direct à la matrice reconfigurable du FPGA, et l'intégration de l'IP overlay ne demande qu'un adaptateur entre les interfaces Wishbone maître et esclave de l'IP et les ports de connexions à la partie fixe, ainsi que de connecter le signal d'interruption généré par l'IP au contrôleur d'interruption du processeur dans la partie fixe. Par exemple, dans le cas de la famille de FPGA Zynq de Xilinx, l'intégration de l'IP demande un adaptateur Wishbone → AXI et un adaptateur AXI → Wishbone. Le processeur peut ainsi avoir accès aux registres de configuration de l'IP, et l'IP et le processeur partagent l'accès à une même mémoire RAM.

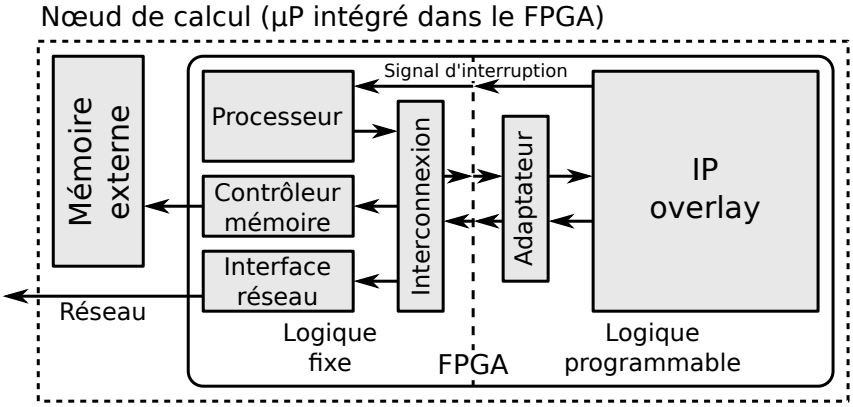


FIGURE 4.5 – Intégration de l'overlay dans un FPGA comportant un processeur implémenté sur le silicium.

Lorsque le processeur est externe au FPGA, comme illustré figure 4.6, il faut un canal de communication entre le processeur et le FPGA, dont le processeur soit maître. Un exemple d'un tel canal qui est souvent présent dans les plateformes de développement FPGA est le bus PCI, comme c'est le cas de la carte APF6 SP d'Armadeus [62] qui intègre un processeur ARM et un FPGA CycloneV, ou dans les carte de développement qui peuvent se connecter en PCI à une carte mère d'ordinateur. Des canaux plus simples comme l'UART ou un bus SPI peuvent être utilisés, cependant leurs faibles débits seraient un goulot d'étranglement pour les performances du système.

Comme le FPGA n'est pas maître du canal qui le relie au processeur, il ne peut pas accéder à des mémoires via le canal. Il faut donc que le FPGA ait accès à une

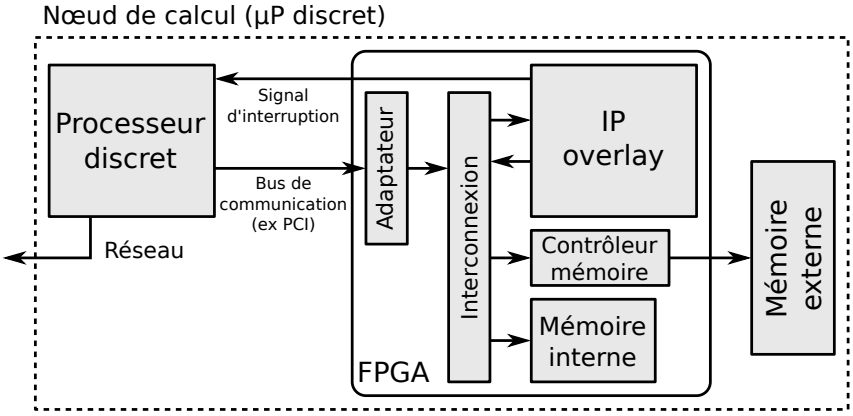


FIGURE 4.6 – Intégration de l'overlay, le processeur étant externe au FPGA.

mémoire, qu'elle lui soit interne ou externe, et qu'il fournisse au processeur un accès à cette mémoire via le canal qui l'y relie. Ainsi, lorsque le processeur est externe au FPGA, l'intégration de l'IP overlay dans le FPGA demande quatre composants : l'IP overlay, un contrôleur de mémoire externe ou directement une mémoire interne, un adaptateur entre l'interface esclave du canal et une interface maître du bus interne, et un bus d'interconnexion qui relie la mémoire, les interfaces maîtres et esclave de l'IP ainsi que l'interface maître de l'adaptateur.

Logiciel embarqué

Lorsqu'un processeur physique est présent sur la plateforme, le système d'exploitation et les pilotes permettant l'accès aux différents contrôleurs est déjà fourni avec la plateforme. Il s'agit par exemple d'une distribution GNU avec un noyau linux et ses modules, comme c'est le cas pour les plateformes Zynq de Xilinx (processeur ARM embarqué) ou de la plateforme APF6 SP d'Armadeus (processeur ARM discret et FPGA CycloneV). Ainsi, la gestion des processus, l'accès au système de fichier et la pile réseau sont déjà mis en place et gérés par l'OS.

L'hyperviseur peut alors être développé en espace utilisateur, et donc de manière portable. La seule partie logicielle qu'il est nécessaire d'adapter à la plateforme est l'accès aux registres de contrôle de l'IP overlay et à la mémoire partagée avec l'overlay, ainsi que la prise en compte du signal d'interruption de l'IP. Cette partie peut demander le développement d'un module noyau, et dépend de la connexion entre le processeur et l'overlay (bus PCI, mapping mémoire, etc). Dans ces travaux, l'IP overlay a été intégrée sur deux plateformes utilisant un processeur externe : sur l'APF6 SP où le FPGA est relié au processeur par un bus PCIe, et sur une carte Nexys 4 reliée en UART à un ordinateur.

4.2.2 ZeFF : un SoC minimaliste pour le prototypage et l'exploitation d'overlay sans processeur physique

Aspect matériel

Lorsque que la plateforme ne comporte pas de processeur physique, ni externe au FPGA ni embarqué, un processeur softcore doit être implémenté dans le FPGA pour pouvoir exécuter l'hyperviseur. Plus largement, pour implémenter un nœud de calcul sur une telle plateforme, il est nécessaire d'implémenter un SoC pour accompagner le processeur et l'IP overlay par une suite de périphériques tels qu'un contrôleur mémoire et un contrôleur Ethernet. Il existe plusieurs systèmes d'intégration permettant de générer rapidement des SoC sur FPGA et qui viennent avec l'outillage de compilation du logiciel embarqué, tels que Qsys d'Altera ou EDK de Xilinx. Cependant, ces systèmes d'intégration sont fournis par les vendeurs de FPGAs et sont limités exclusivement aux FPGAs qu'ils vendent, c'est-à-dire que les SoC générés ne sont pas portables. L'hyperviseur abstrayant la plateforme, le type de SoC utilisé pour implémenter un nœud de calcul est transparent pour l'utilisateur. Cependant, le déploiement de nœuds de calculs sur un ensemble hétérogène de plateformes est simplifié si le SoC est portable. De même, lors du prototypage d'overlay sur différentes plateformes en vue d'extraire différentes métriques propres à l'overlay, il est préférable que le SoC soit fixe et non un facteur variant avec la plateforme.

Un ensemble de composants a donc été assemblé pour former ZeFF [63], un SoC minimaliste permettant d'intégrer les composants fondamentaux d'un nœud de calcul sur n'importe quelle plateforme FPGA dotée d'assez de mémoire et d'une connectivité Ethernet. ZeFF a été développée dans le but d'être minimaliste, portable, simple et flexible, avec pour objectif de mutualiser les développements nécessaires pour supporter différentes plateformes. ZeFF tend à être minimaliste par rapport à la surface occupée par le SoC, de façon à laisser le plus de ressources physiques du FPGA hôte disponible pour implémenter l'overlay. Ainsi nous avons privilégié une faible surface aux performances lors du choix des composants. ZeFF tend à être le plus portable possible pour simplifier son implémentation sur une nouvelle plateforme FPGA. Comme le processeur utilisé est un processeur softcore, seul la partie matérielle de ZeFF a besoin d'être portable, pas nécessairement le logiciel embarqué. Cependant, les circuits discrets accompagnant le FPGA et leurs interfaces peuvent changer d'une plateforme à une autre, et il n'est donc pas possible de faire un SoC complètement portable. Par exemple, la technologie et le type de mémoire externe peut changer (mémoire RAM statique ou dynamique, avec débit de donné simple ou double), ou encore l'interface du transceiver Ethernet, qui bien que normalisée, présente plusieurs variantes (MII, RMII, GMII, RGMII [64]). Cela implique d'adapter les contrôleurs de ZeFF à la plateforme utilisée, comme le contrôleur mémoire ou le MAC (Media Access Control) Ethernet. ZeFF tend à être flexible, pour faciliter l'ajout de fonctionnalités à la plateforme lors du prototypage, que ce soit au niveau matériel ou du logiciel embarqué. ZeFF tend à rester simple, autant pour conserver sa portabilité que sa flexibilité.

Pour faciliter la portabilité de ZeFF, le SoC est organisé autour d'un bus Wishbone [61]. Ainsi, lors de l'adaptation des périphériques internes du FPGA aux composants externes de la plateforme, de nombreuses IP peuvent être récupérées sur le site opencores [65]. De plus l'interface du bus est relativement simple et documentée, ce qui permet aisément de réaliser son propre périphérique ou d'utiliser un adaptateur (souvent déjà disponible) pour connecter l'interface Wishbone à une autre utilisée par une IP spécifique. Un générateur de bus d'interconnexion Wishbone a été réalisé pour faciliter l'ajout de modules maîtres ou esclaves au SoC. Les composants au cœur de ZeFF n'utilisent pas d'IP spécifiques à des vendeurs, et les mémoires block RAM sont instanciées par inférence.

Pour limiter la surface du SoC, le ZPU [66] a été choisi comme processeur softcore. Ce processeur est annoncé comme étant le plus petit processeur 32 bits qui bénéficie d'un compilateur C (gcc). Le ZPU correspond à la philosophie de ZeFF : il est minimaliste, portable, simple et flexible. Il s'agit d'un processeur à pile qui comporte 17 instructions de base, et 30 instructions complémentaires qui peuvent soit être implémentées physiquement pour augmenter les performances, soit émulées de manière transparente par les instructions de base. Une implémentation du ZPU utilisant l'interface Wishbone a été réalisée, tout en permettant l'ajout de nouvelles instructions. La gestion des interruptions a été modifiée de manière à faciliter le portage d'OS sur ZeFF, notamment par l'ajout d'instructions permettant d'activer ou de masquer les interruptions, d'attendre une interruption et reprendre le flot de contrôle au point antérieur à l'interruption.

L'exploitation d'un overlay peut nécessiter que la plateforme accède à une mémoire persistante et de taille importante, par exemple pour héberger un système de fichier contenant des bitstreams virtuels et des jeux de données à traiter. La plupart des plateformes FPGA intègre un connecteur pour carte SD (Secure Digital) relié aux IOs du FPGA. Ces cartes sont des mémoires mortes de plusieurs gigaoctets capables d'héberger un système de fichier persistent. Il est possible d'accéder à une carte SD via un protocole SPI [67], qui n'est pas le protocole natif des cartes SD mais qui est simple à mettre en œuvre sur FPGA. Ainsi, un contrôleur SPI est compris dans ZeFF pour permettre l'implémentation d'un système de fichier sur la plateforme.

Au niveau matériel, la mise en place d'une connexion Ethernet sur un système embarqué demande plusieurs composants, qui prennent en charge la couche physique et de liaison du modèle OSI :

— un connecteur pour pouvoir connecter un câble Ethernet à la carte ;
— un transformateur Ethernet qui permet d'isoler électriquement l'interface du réseau tout en assurant le passage des données ;
— un transceiver Ethernet, gérant la couche physique du réseau, c'est-à-dire la génération et la réception des signaux analogiques sur le support physique ;
— un composant appelé MAC (Media Access Control) gérant la couche de liaison, c'est-à-dire le format numérique des trames Ethernet reçues et émises sur le réseau.

La plupart des plateformes FPGA intègre un connecteur RJ45, un transformateur Ethernet ainsi qu'un transceiver Ethernet discret. Si la plateforme comporte un

processeur discret ou embarqué dans le FPGA, une MAC Ethernet est déjà intégrée comme périphérique avec le processeur et connectée au transceiver. Dans le cas de ZeFF, la MAC Ethernet softcore choisie est l'IP Minimac [68]. Il s'agit d'une MAC minimaliste utilisant très peu de ressources, ce qui est en accord avec la philosophie de ZeFF. Elle permet juste l'envoi et la réception de paquets Ethernet bruts ; tout ce qui relève de l'insertion et du filtrage d'adresse MAC, de l'insertion et de la vérification de checksum doit être pris en charge de manière logicielle par le pilote qui permet à la pile IP d'accéder à la MAC. Cette MAC comporte une interface MII pour se connecter aux transceivers externes 10 ou 100 Mbps ayant une interface MII, ou une interface RMII via un adaptateur RMII/MII ; cependant les transceivers gigabit Ethernet (ayant une interface GMII ou RGMII) ne peuvent pas être utilisés avec cette MAC.

Ainsi, lors du portage de ZeFF sur une nouvelle plateforme, les composants fondamentaux à porter si besoin sont :

— le contrôleur de mémoire RAM ;
— la MAC Ethernet, si le transceiver n'utilise pas une interface MII ou RMII ;
— le contrôleur d'accès à une mémoire morte (par exemple une mémoire flash) si la plateforme ne comporte pas de connecteur pour carte SD.

Avec un processeur, un contrôleur mémoire, un contrôleur SPI et une MAC Ethernet, le SoC ZeFF (illustré figure 4.7) a les composants matériels nécessaires pour exécuter un OS fournissant un système de fichiers et une pile réseau, qui permet d'exécuter l'hyperviseur.

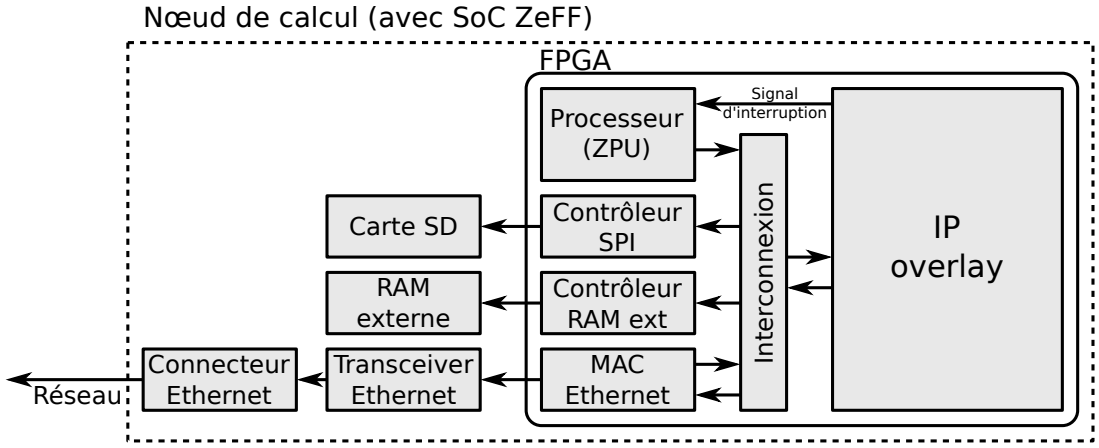


FIGURE 4.7 – Le SoC ZeFF permet d'exécuter l'hyperviseur pour gérer l'IP overlay sur une plateforme n'ayant pas de processeur physique.

Logiciel embarqué

L'OS temps réel FreeRTOS [69] a été porté sur ZeFF. Cet OS est bien documenté, il tient en trois fichiers C et un fichier assembleur, qui est le seul fichier à adapter lors du portage. FreeRTOS permet l'ordonnancement de tâches avec priorités, ainsi que des mécanismes d'IPC classiques (gestion de la mémoire, sémaphores et files de messages).

Le module de système de fichiers FatFs [70] a été intégré à ZeFF. Il s'agit d'un middleware écrit en ANSI C, qui permet d'utiliser un système de fichier FAT/exFAT. FatFs fournit une API applicative (gestion et accès aux fichiers et dossiers) ainsi qu'une API d'accès au média physique de stockage (notamment pour lire et écrire un secteur), ce qui le rend indépendant du type de média physique utilisé. De plus, FatFs s'adapte à n'importe quel environnement RTOS via la définition d'appels systèmes manipulant les sémaphores et allouant/libérant de la mémoire, permettant ainsi l'accès au système de fichiers depuis différents processus concurrents. Un simple contrôleur SPI ainsi qu'un pilote logiciel permettant la lecture et l'écriture de secteurs suivant ce protocole permettent de donner à FatFS accès aux cartes SD, et ainsi de permettre l'utilisation sur ZeFF de système de fichier FAT de plusieurs gigaoctets.

La pile TCP/IP utilisée sur ZeFF est lwIP [71], qui est une pile TCP/IP portable, largement utilisée pour le développement de systèmes embarqués. Tout comme FatFs, lwIP fournit une interface utilisateur, une interface d'accès au média physique (ici une MAC Ethernet), ainsi que la définition d'appels systèmes fournis par l'OS pour pouvoir s'y intégrer. lwIP propose l'interface Berkeley sockets comme API utilisateur, ce qui facilite le développement d'applications réseaux sur ZeFF.

Les pilotes logiciels doivent être adaptés selon les contrôleurs utilisés tout en respectant les patrons attendus par lwIP et FatFs. Lors du prototypage, l'ajout d'un périphérique (par exemple un périphérique permettant des mesures précises de temps ou pour récupérer des traces) passe par l'intégration du périphérique sur le bus d'interconnexion central et éventuellement la remontée du signal d'interruption du nouveau périphérique sur le contrôleur d'interruption.

La plateforme ZeFF a donc les ressources matérielles et logicielles lui permettant d'exécuter l'hyperviseur pour l'exploitation d'un overlay. Pour le prototypage, ZeFF dispose aussi d'une sortie VGA permettant l'affichage de caractères ainsi que d'une entrée pour clavier. L'interface offre à l'utilisateur un accès direct via un terminal, ainsi que l'affichage de journaux asynchrones par rapport aux interactions de l'utilisateur. Le terminal fournit des commandes de base de type UNIX telles que `ls` et `cd`, ainsi que des commandes d'introspection permettant d'observer l'état de la mémoire ou des interruptions, ainsi que des commandes permettant de lire et écrire n'importe quelle valeur dans l'espace mémoire, que ce soit sur une zone mémoire ou un registre d'un contrôleur. Il est aussi possible de développer une nouvelle fonctionnalité et de la rendre disponible sous forme de commande du terminal. Dans ces travaux, ZeFF a été porté sur une carte Nexys 4 comportant un Artix 7 de Xilinx, et un carte DE2 115 comportant un Cyclone IV d'Altera.

4.3 L'hyperviseur : du système sur carte au périphérique réseau

Ce que nous appelons l'*hyperviseur* est le logiciel responsable de la gestion locale d'un overlay, il est exécuté sur le nœud de calcul de l'overlay qu'il gère. L'hyperviseur est représenté figure 4.8, il est composé de trois parties :

— une première couche qui correspond aux appels systèmes bas niveaux accédant aux registres de configuration des contrôleurs de l'IP overlay ;
— un ordonnanceur qui se repose sur ces appels systèmes pour gérer de manière automatique plusieurs applications sur l'overlay ;
— un serveur relayant les requêtes depuis le réseau d'un utilisateur ou d'un contrôleur de nœuds de calculs vers l'ordonnanceur.

La description des appels systèmes permettant l'accès aux contrôleurs matériels de l'IP overlay se trouve en annexe B. La gestion haut niveau de l'overlay est automatisée par l'ordonnanceur qui séquence les appels à ces fonctions. L'ordonnanceur et le serveur seront vu au chapitre 6.

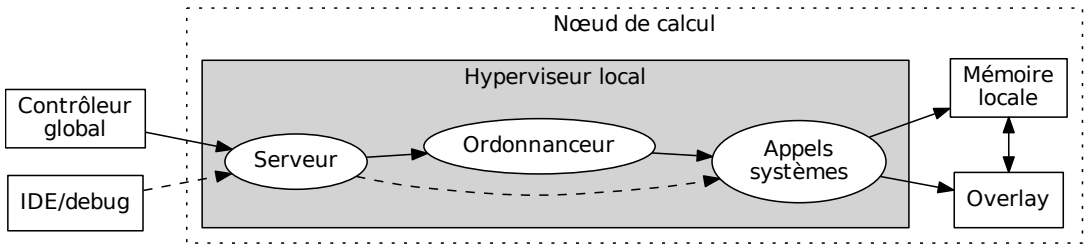


FIGURE 4.8 – Les trois composants de l'hyperviseur. L'ordonnanceur est contourné lors d'une utilisation avec un IDE.

L'ordonnanceur peut être contourné pour laisser à l'utilisateur la possibilité de gérer finement l'overlay. Ainsi, lors du prototypage, le serveur peut relayer des requêtes bas niveau de l'utilisateur directement à la couche d'appels systèmes sans passer par l'ordonnanceur qui est alors désactivé. Ces requêtes peuvent être reçues depuis le réseau ou directement depuis le terminal de la plateforme. Cette fonctionnalité peut être utilisée pour confier la gestion de l'overlay à un IDE connecté à un nœud de calcul via le réseau. La figure 4.8 montre les relations entre les trois composants de l'hyperviseur. De manière similaire à un IDE pour microcontrôleurs qui permet de programmer et déboguer sa cible via une sonde JTAG, un environnement de développement pour overlay, incluant entre autre l'outillage de synthèse virtuelle, comme RedPill [48], peut ainsi se connecter à un nœud de calcul. Un tel IDE permet de programmer l'overlay, mais aussi de lancer l'exécution en pas à pas en choisissant le nombre de cycles d'horloge par pas, tout en étant capable de récupérer l'état d'exécution de l'overlay (plan de snapshot) à chaque pas pour l'inspecter et en déduire l'état des variables définies dans le code source du circuit applicatif en cours de développement. L'IDE peut aussi tirer profit de la génération d'interruption par le circuit applicatif (vu en 4.1.4) pour instrumenter l'application par des points d'arrêts matériels à partir de propriétés PSL [60] par exemple.

4.4 Implémentations : coûts respectifs des composants

Cette section présente le coût matériel des différents contrôleurs de l'IP overlay ainsi que de l'implémentation de quatre prototypes de nœud de calcul sur des FPGAs Xilinx et Altera : deux nœuds de calcul sans processeur physique et implémentant la plateforme ZeFF, et deux nœuds de calcul utilisant un processeur physique.

4.4.1 Coût des contrôleurs de l'IP overlay

Le tableau 4.1 présente l'utilisation en ressources des différents contrôleurs de l'IP overlay. Ces mesures ont été réalisées en synthétisant sur un FPGA Artix-7 (XC7A100T) l'IP overlay générée avec différents contrôleurs puis en analysant les rapports de synthèse. L'overlay synthétisé est une matrice de 7×7 CLBs de 4 LUT-4, avec 16 pistes par canal de routage, et 56 IOs virtuelles. Le coût de l'overlay est de 7662 LUTs et 14192 FFs, il n'est pas pris en compte dans le coût de l'IP. Pour le contrôleur DMA et le contrôleur mémoire, un mot DMA ainsi qu'un mot mémoire et un mot d'adresse sont chacun de 16 bits.

Le coût de l'ensemble des contrôleurs de l'IP overlay est fixe quels que soient les dimensions de la matrice de l'overlay. Dans le cas de la matrice de 7×7, le coût des contrôleurs est de 8.8% des LUTs et de 3.8% des FFs de la matrice, ce qui est raisonnable. Par contre, la couche de réorganisation des IOs (nécessaire pour contraindre le placement des IOs lorsque VPR est utilisé, cf 4.1.5) a un coût de 27.8% des LUTs et 5.1% des FFs de la matrice. Dans ce cas d'une matrice de 56 IOs, la couche de réorganisation des IOs a à elle seule plus de trois fois le coût en LUT de l'ensemble des contrôleurs de l'IP.

Composant	LUT	FF
IP_base	236	209
Contrôleur horloge applicative	56	36
Contrôleur interruption	3	3
Contrôleur DMA	281	245
Contrôleur mémoire	165	166
Contrôleur DMA & mémoire	382	292
Réorganisation des IOs	2128	728
Tous les contrôleurs, sans réorg. des IOs	677	540
Tous les contrôleurs, avec réorg. des IOs	2805	1268

TABLE 4.1 – Utilisation des ressources FPGA par les contrôleurs de l'IP overlay.

4.4.2 Implémentations de ZeFF

ZeFF a été réalisé sur la carte de développement Nexys 4 comprenant un FPGA Artix-7 de Xilinx, puis a été porté par M. Mohamad Najem sur la carte DE2-115 comprenant un FPGA Cyclone IV E d'Altera. Ces deux cartes n'embarquent pas de processeur physique, mais offrent toutes deux différentes entrées/sorties permettant l'utilisation d'un clavier, d'une carte SD et d'un écran VGA, ainsi qu'une connectivité Ethernet et série et comporte chacune une mémoire RAM externe.

La figure 4.9 présente l'implémentation du SoC ZeFF sur les deux plateformes. Au niveau matériel, les portages de ZeFF sur les deux cartes sont quasiment identiques. En effet, les mémoires RAM utilisées sur les deux cartes sont de type mémoire statique et le contrôleur mémoire utilisé est le même pour les deux portages.

Le portage sur Nexys 4 a nécessité un adaptateur RMII vers MII pour connecter la PHY Ethernet de la carte à la MAC de ZeFF. Pour permettre d'accéder à l'espace mémoire sur SoC depuis une liaison série (par exemple pour charger le programme du ZPU dans la mémoire RAM), le module nommé UART bus_master sur la figure 4.9 a été ajouté à ZeFF. Le processeur et le SoC étant les mêmes sur les deux portages, le logiciel embarqué dans ZeFF (comprenant l'hyperviseur) n'a pas eu à être modifié, mis à part la fonction permettant de configurer la PHY Ehternet qui est différente selon la carte.

Le tableau 4.2 présente l'utilisation des ressources FPGA de chacun des modules de ZeFF pour les deux portages. Le portage sur Nexys 4 consomme 7.4% des LUTs et 1.9% des flip-flops de l'Artix-7, tandis que le portage sur DE2-115 consomme 4.6% des LUTs et 2.2% des flip-flops du Cyclone IV. Le reste des ressources du FPGA est disponible pour implémenter l'overlay.

4.4.3 Implémentations de nœuds de calcul avec processeur externe

Pour permettre le prototypage d'overlay sur une grande variété de cartes FPGA, un prototype de nœud de calcul utilisant un processeur externe communicant avec le FPGA via un port série a été réalisé. L'hyperviseur est exécuté sur un ordinateur et communique avec le FPGA via un port série implémenté par un pont USB/UART. Le pont USB/UART permet des débits allant jusqu'à 390 kio/s, ce qui n'est pas suffisant dans le cadre de l'exploitation d'overlay, mais permet de réaliser des nœuds de calcul à des fins de prototypage sur une grande variété de cartes FPGA. En effet, la communication avec le FPGA via un port série de type UART ne demande qu'une pin d'entrée et une de sortie du FPGA. Le schéma de la partie FPGA est illustré figure 4.10. Il s'agit du schéma d'intégration le plus simple : le FPGA implémente uniquement l'IP overlay, un contrôleur mémoire et/ou une mémoire interne, un bus d'interconnexion et le module d'accès à l'espace mémoire du système. Le tableau 4.3 présente l'occupation des ressources par les différents

Composant	Nexys 4 Xilinx Artix-7			DE2-115 Altera Cyclone IV E		
	LUT-6	FF	BRAM	LUT-4	FF	bits mémoire
Interconnexion Wishbone	458	48	–	562	9	–
ZPU	825	271	–	1020	274	–
Caches I&D	223	214	5	321	190	38400
MAC Ethernet	1235	558	2	1050	580	32768
Ctrl interruptions	78	61	–	50	44	–
Ctrl mémoire	110	79	–	120	114	–
Ctrl SD (SPI)	438	225	–	540	249	–
Ctrl reset	118	60	–	127	62	–
Ctrl clavier	147	110	–	183	116	–
UART bus_master	270	196	–	376	212	–
RAM interne & ROM bootloader	5	1	8	6	1	262144
Timer système	80	61	–	93	61	–
Timer généraliste	150	117	–	208	117	–
Ctrl VGA	194	194	2	225	248	24576
Ctrl IOs	341	241	–	351	231	–
Total	4672	2436	17	5232	2508	357888

TABLE 4.2 – Utilisation des ressources FPGA par le SoC ZeFF pour les plateformes Nexys 4 et DE2-115.

modules : l'infrastructure consomme moins de 1% des LUTs et moins de 0.3% des flip-flops du FPGA présent sur la carte Nexys 4.

Une implémentation plus réaliste d'un nœud de calcul utilisant un processeur physique communicant avec le FPGA via un bus PCI express a été implémenté sur la carte de développement APF6_SP d'Armadeus [62]. Le schéma d'intégration de l'IP overlay est présenté figure 4.11. L'infrastructure d'intégration a été réalisée par Mohamad Najem avec le logiciel Qsys et utilise différentes IP fournies par Altera. L'IP PCIe utilisée fournit trois interfaces maîtres ayant chacune leur espace mémoire, nommée BAR (pour Base Address Region) dans la figure 4.11. Comme l'espace mémoire adressé par chacune de ces interfaces est trop petit pour adresser l'ensemble de la mémoire RAM, un module (span extender, fournit par dans l'outil Qsys) a été intégré pour permettre d'accéder à l'ensemble de la mémoire RAM en faisant glisser la fenêtre vue par le BAR 0. Le BAR 2 permet d'accéder au registre de configuration de l'IP overlay. L'infrastructure générée par Qsys utilise des interfaces Avalon [72] entre les modules, l'intégration de l'IP overlay a donc demandé deux interfaces Avalon/Wishbone. L'infrastructure comprend aussi un module permettant au processeur de configurer le FPGA via l'interface PCIe. Le tableau 4.4 présente l'utilisation des ressources du FPGA par les composants de l'infrastructure. L'infrastructure consomme 5.6% des ressources logiques (Adaptive Logic Mo-

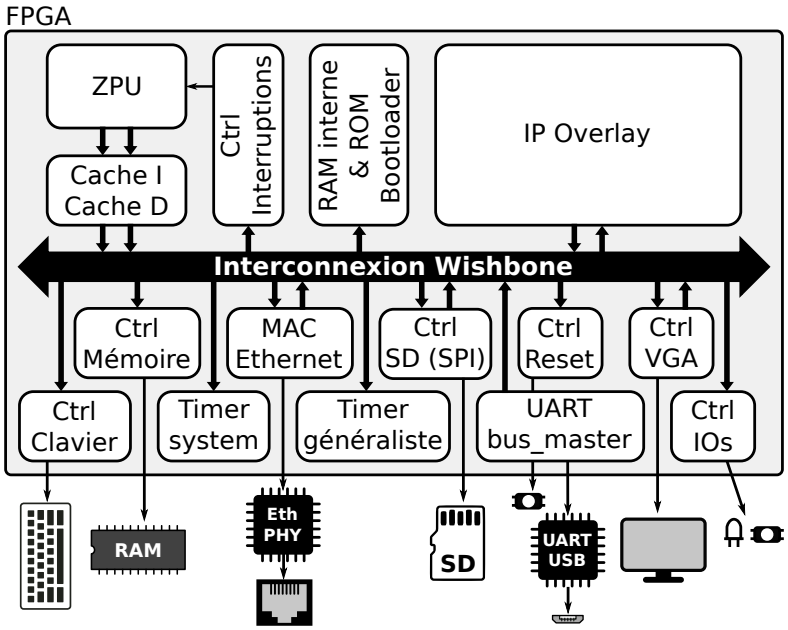


FIGURE 4.9 – Implémentation du SoC ZeFF sur les plateformes Nexys 4 et DE2-115.

Composant	LUT-6	FF	BRAM
Interconnexion Wishbone	112	9	–
RAM interne	5	1	8
Ctrl mémoire	110	79	–
UART bus_master	373	265	–
Total	610	354	8

TABLE 4.3 – Consommation des ressources d'un FPGA Artix-7 par les composants du nœud de calcul minimaliste.

dule à 8 entrées) et 1.4% des flip-flops du FPGA de l'APF6_SP.

4.4.4 Co-simulation : nœud de calcul virtuel

Pour permettre le prototypage d'overlay et la validation de circuits applicatifs sans FPGA physique, par exemple pour implémenter un overlay contenant trop de ressources pour être synthétisé sur un des FPGAs à disposition de l'expérimentateur, un nœud de calcul virtuel a été réalisé en couplant l'hyperviseur avec une simulation interactive de l'IP overlay. Ce nœud de calcul est dit *virtuel* car l'architecture fonctionnelle de l'overlay n'est pas implémentée matériellement sur un FPGA mais est simulée de manière logicielle sur un processeur.

GHDL [73] est un simulateur VHDL libre basé sur le compilateur GNU gcc, qui génère un exécutable x86 de simulation à partir de sources VHDL. GHDL fournit une interface de programmation avec d'autre langages tel que le C (via l'interface

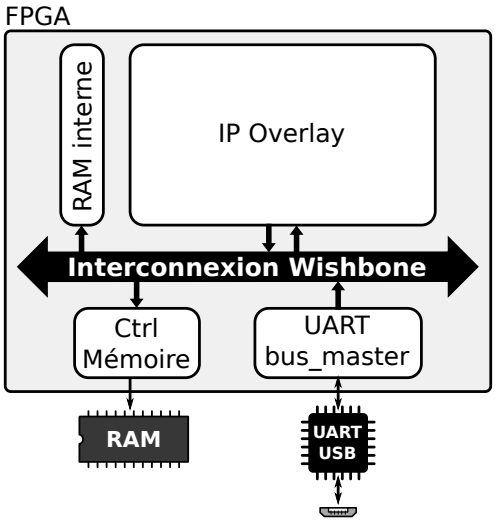


FIGURE 4.10 – Nœud de calcul minimaliste. L'hyperviseur exécuté sur un ordinateur accède à l'IP overlay et aux mémoires via un port série.

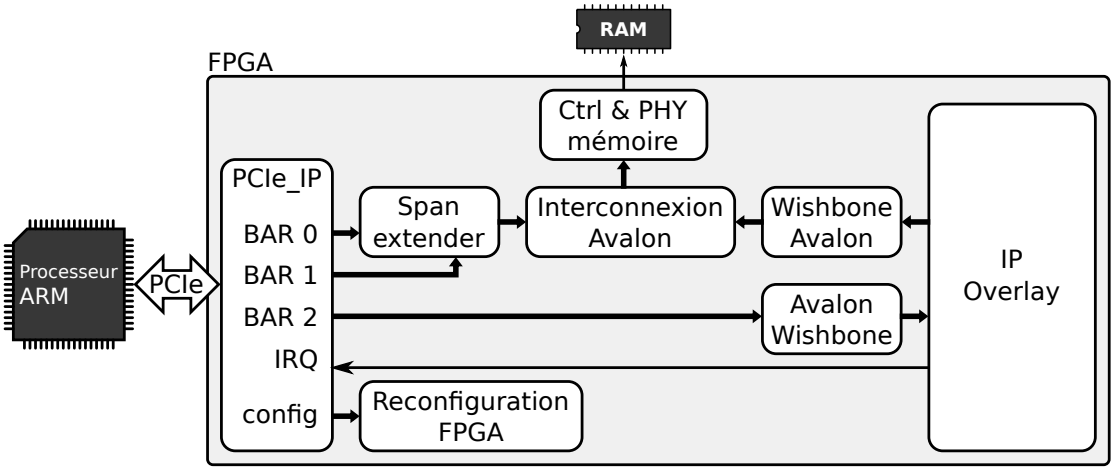


FIGURE 4.11 – Implémentation de nœud de calcul sur la plateforme APF6_SP.

normalisée VHPI [74]) et permet d'intégrer le point d'entrée "main" du programme de simulation créé dans un programme tiers lors de la phase d'édition de lien de celui-ci. Ainsi, dans le nœud de calcul virtuel, l'hyperviseur lance le "main" du simulateur généré par GHDL dans un fil d'exécution séparé et communique avec lui via des files et des sémaphores. Le VHDL simulé est la description RTL de l'IP overlay ; les échanges C $\leftrightarrow$ VHDL sont réalisés au niveau des interfaces Wishbone maître et esclave de l'IP simulée ainsi que son signal d'interruption. La simulation étant interactive, l'hyperviseur accède à l'IP overlay simulée et la contrôle exactement comme s'il s'agissait d'une IP overlay implémentée physiquement. L'abstraction en nœud de calcul offerte par l'hyperviseur rend transparent le fait que l'overlay soit implémenté physiquement sur un FPGA ou simulé de manière logicielle, et un circuit applicatif peut par exemple être déployé sur un nœud de calcul physique puis migré et restauré sur un nœud de calcul virtuel.

Évidemment, la vitesse de simulation de l'IP overlay est bien moindre que sa vitesse de fonctionnement sur un FPGA. Par exemple, la vitesse de simulation

Composant	ALM	FF	bits mémoire
PCIe_IP	2222	2158	24432
Ctrl & PHY mémoire	2267	1941	199744
Interconnexions Avalon	1187	1767	–
Span extender	2	18	–
Wishbone to Avalon	6	2	–
Avalon to Wishbone	6	35	–
Reconfiguration FPGA	631	381	4096
Total	6321	6302	228272

TABLE 4.4 – Utilisation des ressources FPGA par l'infrastructure d'intégration de l'IP overlay sur le FPGA de l'APF6_SP.

d'un vFPGA-flexible de 256 LUT-4 sur un processeur AMD Opteron 6274 à 2.2 GHz correspond à une fréquence $f_{Clk_{VTPR}}$ moyenne de 1.8 kHz, soit 10^5 fois moins rapide que sa vitesse de fonctionnement (200 MHz) sur un FPGA Artix7. De plus, la vitesse de simulation décroit avec la taille de l'overlay (c'est-à-dire le nombre de ressources à simuler).

4.5 Résumé

Dans ce chapitre ont été présentés l'intégration d'un overlay sur des plateformes FPGA permettant leur déploiement, et l'abstraction de ces plateformes en nœuds de calcul distribuables à des utilisateurs.

Pour supporter l'intégration de différentes matrices d'overlays dans différentes plateformes FPGA, en dépit de l'évolution des unes indépendamment des autres, la matrice reconfigurable de l'overlay est dans un premier temps intégrée en une IP comprenant l'overlay ainsi qu'un ensemble de contrôleur qui lui sont propres. Cette IP – l'IP overlay – a une interface fixe quels que soient les paramètres de son overlay.

Dans un deuxième temps, l'IP overlay est intégrée dans un système comprenant un processeur capable d'exécuter l'hyperviseur, c'est-à-dire un logiciel contrôlant et gérant l'exécution de l'overlay auquel il est attaché. Si la plateforme hôte possède un processeur physique, celui-ci peut être utilisé pour exécuter l'hyperviseur ; sinon, un système de type SoC avec un processeur softcore est implémenté avec l'overlay sur le FPGA.

L'utilisateur communique avec la plateforme via le réseau. L'hyperviseur joue le rôle d'interface entre l'overlay et l'utilisateur. Ainsi, l'hyperviseur banalise l'accès à l'overlay et rend transparentes les spécificités de la plateforme matérielle (par exemple que le processeur soit physique ou non). Nous appelons *nœud de calcul*

l'ensemble formé par l'overlay et son hyperviseur. Cette approche permet de supporter à la fois la variabilité des overlays intégrés et la variabilité des plateformes hôtes utilisées, tout en en donnant une vue standardisée à l'utilisateur.

Chapitre 5

Programmation des overlays : synthèse applicative

Nous avons vu dans le chapitre 3 comment définir l'architecture fonctionnelle d'un overlay et comment en générer une implémentation synthétisable. Au chapitre 4, nous avons vu comment intégrer un overlay dans une plateforme FPGA physique pour pouvoir l'utiliser (le configurer et l'alimenter en données). Maintenant que nous avons un overlay prêt à être utilisé, il nous faut être capable de le programmer, c'est-à-dire :

- à partir d'une description textuelle d'une application, produire le fichier binaire de configuration permettant de configurer l'overlay pour qu'il implémente l'application ;
- il faut que l'outil de programmation cible le modèle d'architecture de l'overlay, et s'adapte aux changements de paramètres de ce modèle ;
- il faut être capable d'évaluer les performances des circuits applicatifs implémentés, c'est-à-dire leur fréquence maximale de fonctionnement sur la plateforme physique d'exécution ;
- il faut pouvoir accompagner le concepteur pendant le développement applicatif, notamment en lui permettant le débogage de son application à différents stades de son implémentation.

Dans ces travaux le flot de synthèse ciblant l'overlay est appelé *flot de synthèse virtuelle*, par opposition au *flot de synthèse physique* qui cible le FPGA hôte. Le flot de synthèse virtuelle est utilisé pour synthétiser une application sur l'overlay, tandis que le flot de synthèse physique est lui utilisé pour synthétiser l'overlay sur le FPGA hôte. La différence entre les deux est donc la cible. L'architecture fonctionnelle de l'overlay étant indépendante de l'architecture du FPGA hôte, les flots de synthèse virtuelle et physique sont indépendants. Le flot physique est fourni par les vendeurs de FPGAs (par exemple ISE ou Vivado pour les FPGAs Xilinx, ou Quartus pour les FPGAs Altera). Le flot physique ne cible que les FPGAs du vendeur et ne peut donc pas être utilisé pour effectuer la synthèse virtuelle qui cible l'overlay. Ainsi, pour être en mesure de synthétiser des applications sur un overlay, un flot de synthèse virtuelle doit donc être mis en place pour cibler celui-ci.

Le flot de synthèse virtuelle comporte plusieurs étapes, chacune d'elles doit pouvoir être vérifiée lors de la mise en place du flot. Aussi, une fois le flot de synthèse virtuelle opérationnel, l'utilisateur du flot doit pouvoir être en mesure de vérifier et déboguer les applications qu'il développe. Ainsi, des moyens de vérification et de débogage doivent être mis en place avec le flot de synthèse virtuelle.

De même que pour un flot de synthèse physique ciblant de l'ASIC du FPGA, pour qu'une application virtuelle soit utilisable sur l'overlay ciblé, le flot de synthèse virtuelle doit être en mesure de fournir la fréquence d'horloge maximale de fonctionnement jusqu'à laquelle l'application est garantie de fonctionner suivant le comportement selon lequel celle-ci a été conçue. L'extraction de cette fréquence maximale de fonctionnement est appelée *analyse de timing*. Elle repose sur l'analyse du délai de chaque signal applicatif parcourant les ressources du plan de calcul de l'overlay, de sa source à sa destination. Ainsi, dans le cas des overlays, l'analyse de timing doit non seulement prendre en compte l'architecture du plan de calcul de l'overlay ciblé, mais aussi les délais physiques des ressources atomiques de l'overlay synthétisé sur son FPGA hôte. La fréquence maximale de fonctionnement d'une application virtuelle ne dépend donc pas seulement de l'application elle-même et de l'overlay qu'elle cible, mais aussi de l'instance de l'overlay sur lequel elle est exécutée (c'est-à-dire du FPGA hôte donné et de la qualité de la synthèse physique réalisée pour cet overlay sur le FPGA). Cependant, pour garder le flot de synthèse virtuelle indépendant du flot de synthèse physique, et surtout pour maintenir l'overlay en tant que couche d'abstraction isolant complètement l'application du FPGA sous-jacent, permettant ainsi d'établir pour les overlays un slogan similaire à celui de la machine virtuelle Java "application synthétisée une fois, exécutable sur n'importe quelle instance de l'overlay ciblé", il est désirable que l'analyse de timing ne soit réalisée qu'une fois, lors de la synthèse virtuelle de l'application, et ne dépende donc que de l'application et de l'overlay ciblé, et non de l'instance de l'overlay sur laquelle l'application est ensuite exécutée.

Les trois contributions présentées dans ce chapitre sont les suivantes :

(i) la mise en place d'un flot de synthèse virtuelle capable de générer des binaires de configuration ciblant les architectures overlay présentées dans le chapitre 3 ;

(ii) la mise en place d'outils permettant de vérifier chaque étape du flot et de déboguer une application, de sa description structurelle à son exécution sur carte ;

(iii) une méthode d'analyse de timing sure pour les overlays, permettant d'isoler la fréquence de fonctionnement maximale virtuelle f_{virt} (propre seulement à l'application synthétisée et à l'architecture du plan de calcul de l'overlay ciblé) de la fréquence de fonctionnement maximale physique f_{reelle} (qui dépend en plus de la synthèse physique de l'overlay et du FPGA hôte). La fréquence de fonctionnement réelle est déduite simplement de la fréquence virtuelle selon $f_{reelle} = f_{virt} \times f_{VTPR}$. Le scalaire f_{VTPR} est propre à chaque instance d'overlay (l'overlay ciblé synthétisé sur un FPGA donné) et est déduite de l'analyse de timing réalisée par l'outil constructeur lors de la synthèse physique de l'overlay sur son FPGA hôte.

5.1 Flot de synthèse virtuelle

5.1.1 Flot de synthèse modulaire

Il est possible de concevoir une infinité d'overlays différents, qu'il s'agisse de matrices de processeurs, de réseaux d'opérateurs gros grain, de matrices grain fin ou encore d'un mixte de différentes granularité. Les architectures possibles et leurs modèles d'exécution sont tellement variés qu'il n'est pas possible d'avoir un outil de synthèse virtuelle universel générique à tous les overlays. Cependant, parmi les flots de synthèses virtuelles, certaines parties peuvent être similaires. Lors de la mise en place d'un flot de synthèse virtuelle ciblant un overlay donné, pour diminuer le temps de développement et de vérification du flot, il est donc avantageux de mettre en place un flot de synthèse virtuelle modulaire qui permette de réutiliser différentes parties d'autres flots.

Le flot de synthèse virtuelle est une *toolchain*, c'est-à-dire une chaîne d'outils : de manière similaire à une toolchain de compilation (par exemple de sources C vers un exécutable), pour partir d'une description textuelle de l'application pour arriver à un binaire de configuration ciblant un overlay, le flot de synthèse virtuelle met en œuvre plusieurs outils, chacun effectuant, à la suite de ses prédécesseurs, une tâche particulière. Aussi, un des intérêts des overlays est de pouvoir mener des expérimentations sur les outils de synthèse eux-mêmes, c'est-à-dire des composants du flot. Dans ce cas, pour ne pas avoir à implémenter tous les outils du flot, il est intéressant de pouvoir partir d'un flot fonctionnel pour ensuite remplacer un outil du flot par l'outil expérimental en développement. Pour permettre un tel flot modulaire, il est nécessaire que les fichiers de sorties des outils amonts et les fichiers d'entrées des outils avals partagent le même format et soient compatibles.

5.1.2 Exploration architecturale : généricité et spécialisation du flot

Certaines parties du flot de synthèse virtuelle sont intimement liées à l'architecture cible, et synthétiser une application de façon à exploiter au mieux une cible particulière demande la spécialisation de l'outil de synthèse pour celle-ci. Or, comme il a été vu au chapitre 3, lors de la conception d'un overlay, il est nécessaire d'évaluer celui-ci suivant chaque paramètre de l'espace de conception de son architecture fonctionnelle i) par rapport à son coût physique, et ii) par rapport à sa qualité fonctionnelle. La qualité fonctionnelle d'un overlay se mesure en synthétisant un ensemble de benchmarks sur l'overlay, puis et en évaluant la fréquence de fonctionnement de ces benchmarks et leur taux d'occupation de l'overlay. Pour effectuer une comparaison correcte de la qualité fonctionnelle des architectures issues de l'espace de conception d'un overlay, il est donc nécessaire que le même outil soit utilisé pour synthétiser les benchmarks sur toutes les architectures afin d'éviter un biais dû à l'outil. Il est aussi nécessaire que l'outil soit en mesure d'exploiter

au mieux les spécificités de chaque architecture (suivant leur paramètres). L'exploration de l'espace de conception d'un overlay demande donc de mettre en place un flot de synthèse virtuelle qui soit générique aux architectures de l'espace de conception, et qui puisse être spécialisé suivant chacun des paramètres de l'espace de conception exploré.

Dans ces travaux, nous avons choisi d'implémenter des architectures reconfigurables grain fin comme preuve de concept des overlays, car nous pensons que les overlays grain fin soulèvent le plus de problèmes techniques à l'implémentation, c'est-à-dire que si l'on peut utiliser un overlay grain fin, alors monter en granularité n'est pas un problème. De plus, les overlays grain fin sont les architectures reconfigurables les plus génériques en termes du spectre d'applications qu'ils permettent d'implémenter, l'augmentation du grain ce faisant en spécialisant les opérateurs de l'architecture pour un domaine applicatif donné. La suite de ce chapitre présente donc la synthèse virtuelle ciblant des overlays grain fin, similaire au flot de synthèse sur les FPGAs du commerce.

5.1.3 Les étapes du flot de synthèse virtuelle

Le flot de synthèse virtuelle pour une cible grain fin est réalisée en différentes étapes :

Synthèse haut niveau La description textuelle algorithmique de l'application est transformée en une description textuelle structurelle ou comportementale de l'application.

Synthèse RTL La description structurelle ou comportementale du circuit applicatif est synthétisée en une netlist RTL.

Synthèse logique Cette netlist est ensuite optimisée pour réduire son nombre de composants (optimisation en surface) ainsi que son chemin critique (optimisation des performances).

Mapping technologique La netlist optimisée est synthétisée en transformant ses composants en composants atomiques – par exemple des LUTs – disponibles dans l'architecture cible.

Packing Les composants connexes peuvent éventuellement être regroupés de façon à former des composants de hiérarchie plus haute, comme le regroupement d'une LUT et d'un registre applicatif pour former un BLE, ou le regroupement de plusieurs BLEs pour former un CLB.

Placement Ces composants sont ensuite placés sur l'architecture cible.

Routage Puis les nets les reliant sont routés suivant les canaux de routage de l'architecture.

Analyse de timing Le placement et routage du circuit applicatif est analysé pour calculer la fréquence de fonctionnement maximale de fonctionnement en deçà de laquelle le circuit virtuel synthétisé est assuré de se comporter suivant sa description textuelle (code source).

Extraction de bitstream Finalement, les configurations de chaque ressource configurable de l'architecture sont extraites et concaténées pour assembler le bitstream virtuel implémentant le circuit applicatif sur la cible.

La première étape de synthèse haut niveau est optionnelle : le concepteur de l'application peut entrer une description structurelle ou comportementale directement dans le flot de synthèse.

5.1.4 Les outils existants

La figure 5.1 montre différents outils existants (les arcs sur la figure) qui réalisent les différentes étapes de synthèse présentées ci-dessus. Les nœuds du graphe sont les langages d'entrées et de sorties de ces outils.

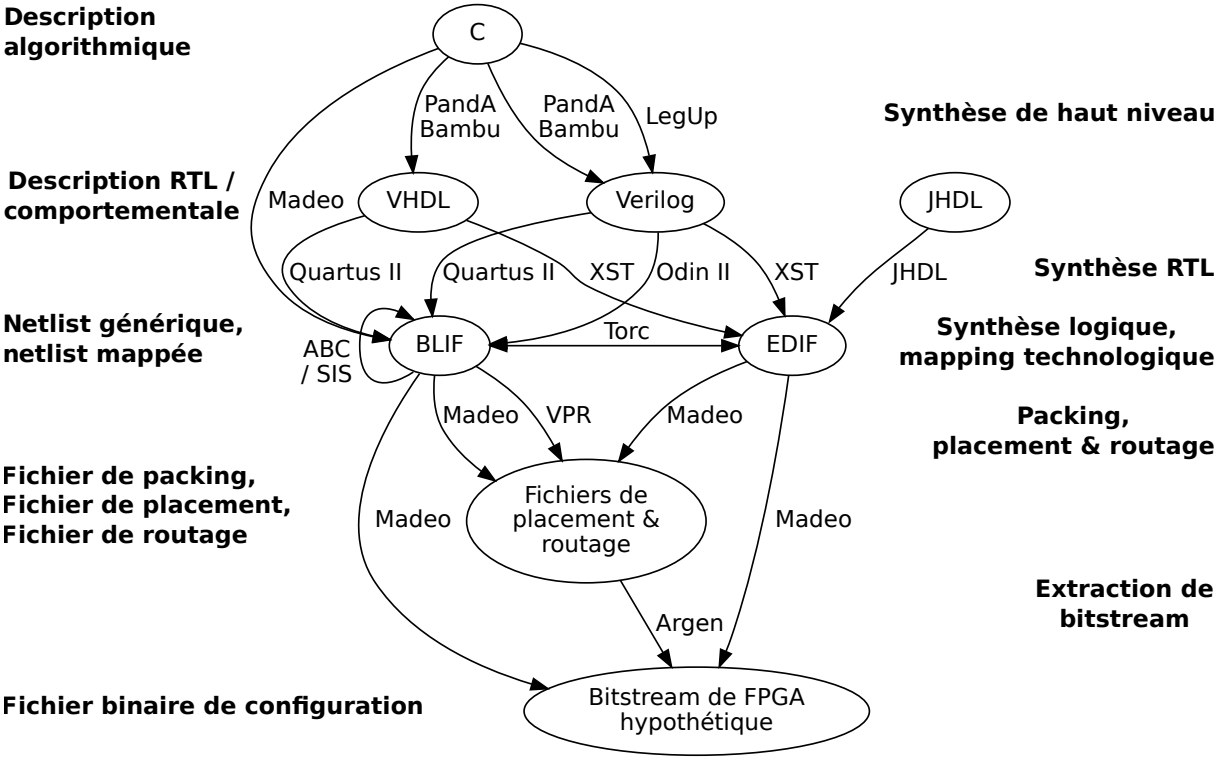


FIGURE 5.1 – Différents formats de fichier (nœuds) et des outils qui permettent les transformations de l'un à l'autre (arcs)

Bambu [75] et LegUp [76] sont des outils de synthèse haut niveau, c'est-à-dire qu'ils partent d'une description algorithmique de l'application, écrite dans le langage C, qu'ils transforment en la description structurelle et/ou comportementale d'un circuit logique, décrite dans le langage VHDL ou Verilog.

Quartus et XST sont des synthétiseurs RTL commerciaux fournis par les vendeurs de FPGAs Altera et Xilinx. Ils transforment la description textuelle structurelle et comportementale du circuit applicatif (écrite en VHDL ou Verilog) en une netlist RTL, c'est-à-dire en un ensemble de composants logiques inter connectés. Le

format de la netlist produit est propriétaire et propre à chacun de ces deux outils, et n'est pas fait pour être utilisé en dehors de l'outillage fournit par leurs vendeurs. Cependant, il est possible d'utiliser ces outils pour produire des netlists dans les formats ouverts BLIF (Berkeley Logic Interchange Format) [77] et EDIF (Electronic Design Interchange Format) [78]. Outre ces deux outils RTL propriétaires, Odin II [79] est un synthétiseur RTL open-source qui permet la synthèse du Verilog vers le BLIF. Un autre exemple académique (mais ancien) est l'environnement de développement JHDL [80], qui comporte entre autre un synthétiseur RTL prenant en entrée la description structurelle d'un circuit écrite dans un DSL (Design Specific Language) reposant sur le langage Java, et permet d'aboutir à une netlist EDIF. Le framework Madeo [48] permet de synthétiser la description algorithmique d'une application en Smalltalk directement en une netlist RTL BLIF.

Torc [81] est un ensemble d'outils développé dans le but d'aider la création d'outils de synthèse pour FPGA. Notamment, Torc peut être utilisé pour manipuler et convertir des netlists au format BLIF et EDIF, placer et router ces netlists sur des FPGAs Xilinx, et s'interfacer avec les outils Xilinx (par exemple de génération de bitstream) via le langage textuel XDL [82].

ABC [83] est un outil de synthèse [84] et de vérification [85] de circuits logiques séquentiels. ABC est le successeur de SIS [86]. Ces deux outils permettent la synthèse logique et le mapping technologique de netlists depuis et vers le format BLIF. La synthèse logique correspond à la minimisation logique (c'est-à-dire la réduction du nombre de composants utilisés) et à la minimisation du nombre de niveaux dans la netlist (c'est-à-dire à réduction du nombre maximum de composants combinatoires séparant deux éléments séquentiels, corrélée au chemin critique du circuit). Ces outils permettent aussi la synthèse séquentielle des circuits en modifiant conjointement les éléments logiques et séquentiels du circuit tout en conservant l'équivalence séquentielle entre le circuit de départ et celui obtenu. Un exemple est le retiming [87] qui consiste à modifier la position des éléments séquentiels de la netlist par rapport aux éléments combinatoires de façon à diminuer le chemin critique du circuit. Le mapping technologique est le procédé qui consiste à transformer les éléments de la netlist en éléments disponibles dans l'architecture ciblée. Il peut s'agir d'une bibliothèque de cellules standards dans le cadre d'une cible ASIC, ou de flip-flops et de LUTs à K entrées dans le cas d'une cible FPGA ou d'un overlay grain fin.

VPR [43] est un outil de packing, placement et routage de netlists BLIF technologiquement mappées, pour une cible FPGA hypothétique. Il est largement utilisé dans la recherche, et rassemble une communauté importante de contributeurs qui le développent activement. Outre son efficacité, VPR est un outil bien documenté, de même que les formats de ses fichiers d'entrée/sortie. Aussi, VPR permet de générer des modèles d'architectures grain fin détaillés et complètement spécifiés à partir de description haut niveau [45] (comme le nombre de pistes par canal de routage), ce qui en fait un outil idéal pour l'exploration fonctionnelle des architectures qu'il supporte. Les fichiers de sortie produits par VPR indiquent le packing des composants de la netlist d'entrée, le placement dans l'architecture de ces éléments packés ainsi que le routage emprunté par les nets ; cependant, VPR ne

produit pas de binaire de configuration pour l'architecture cible. En effet, VPR ne modélise pas le plan de configuration des architectures et ne peut donc pas générer de bitstream. VPR permet aussi d'effectuer l'analyse de timing des circuits placé et routés. Cependant, cette analyse de timing vise des architectures FPGA implémentées sur silicium et se base sur des propriétés physiques telles que la résistance et la capacité des pistes de routage (indiquées dans la description haut niveau de l'architecture). Ainsi, l'analyse de timing effectuée par VPR ne peut pas être utilisée dans les cadre des overlays.

En plus de la synthèse applicative vers une netlist RTL, le framework Madeo permet aussi le placement et le routage de netlists (BLIF ou EDIF) mappées sur une architecture reconfigurable. L'architecture cible est modélisée à partir d'une description fine, et Madeo génère les outils de synthèse physique (placement et routage) en fonction de celle-ci. Ainsi, l'espace de conception des architectures utilisables avec Madeo est plus large que celui de VPR, mais la description bas niveau de l'architecture et le fait qu'il est parfois nécessaire d'ajouter des modifications à l'outil pour prendre en compte des détails de l'architecture cible rendent l'exploration plus laborieuse. Cependant, comme le plan de configuration de la cible est modélisé en même temps que son plan de calcul, Madeo est capable de produire un binaire de configuration.

5.2 Conception d'un flot de synthèse virtuelle

Contrairement à ce que pourrait laisser penser la figure 5.1, assembler différents outils pour former un flot de synthèse virtuelle n'est pas aussi simple que de partir d'un langage de départ (tel que C, VHDL ou Verilog) et de choisir les outils et les langages intermédiaires qui permettent d'arriver à un bitstream virtuel. En effet, le choix des outils n'est pas seulement motivé par le fait que le langage du fichier de sortie d'un outil doive correspondre à celui du fichier d'entrée de l'outil suivant : chaque outil du flot doit être en adéquation avec l'architecture cible.

5.2.1 Adéquation outil/architecture

Chaque outil du flot doit avoir connaissance des spécificités de l'architecture cible. L'adéquation outil/architecture cible est évidente pour les outils de packing, de placement et de routage, qui viennent mapper la netlist applicative sur les éléments du plan de calcul de l'architecture. En aval du placement et routage, l'analyse de timing nécessite de connaitre les délais de chaque élément du plan de calcul, tandis que l'extraction de bitstream demande de connaitre la correspondance configuration/bitstream partiel de chaque élément reconfigurable ainsi que le chainage des bitstreams partiels de tous les éléments de l'architecture pour former le bitstream final. En amont du packing, placement et routage, le mapping technologique de la netlist applicative générique demande de connaitre les composants ato-

miques disponibles dans le plan de calcul de l'architecture cible (comme le nombre d'entrées des LUTs).

Les phases amonts du mapping technologique (c'est-à-dire la synthèse haut niveau, la synthèse RTL et la minimisation logique) ne demandent pas a priori de connaitre les spécificités de la cible, étant donné que la netlist produite avant le mapping technologique est générique. Cependant, une architecture reconfigurable peut contenir des macros blocs dans son plan de calcul, c'est-à-dire des éléments de granularité plus élevée que les LUTs et les registres applicatifs. Dans le cadre des FPGAs commerciaux, les mémoires BRAM et les blocs DSP sont des exemples de macros blocs intégrés dans la matrice de LUTs et de registres. Ces macros blocs sont implémentés "en dur" directement sur le silicium, c'est pourquoi ils réalisent leurs fonctions en occupant moins de surface et en présentant des délais moindres que si leurs fonctions étaient réalisées sur les ressources grain fin (LUTs et registres applicatifs) de l'architecture. De manière similaire, dans le cadre des overlays, les macros blocs sont implémentés directement sur les ressources configurables du FPGA hôte, et présentent donc une occupation en surface et des délais moindre que si leurs fonctions étaient implémentées sur les ressources grain fin virtuelles de l'overlay. Les macros blocs sont donc des éléments pré synthétisés, et, bien qu'ils puissent être configurables (sous forme de "mode", par exemple pour un bloc mémoire, différents rapports de tailles adresse/donnée), ils le sont dans une moindre mesure que les ressources grain fin de l'architecture, et ne sont donc pas manipulés de la même manière par les outils. De plus, dans le cadre des overlays, les macros blocs étant – comme le reste de l'overlay lui-même – indépendants du FPGA hôte, tout type de macro bloc peut être implémenté dans l'architecture suivant les besoins du domaine applicatif, qu'il s'agisse de blocs mémoire, de blocs DSP classique ou encore de bloc de multiplication en corps de Galois ou de blocs cryptographiques.

Or, les fonctions des macros blocs étant plus haut niveau que celles des ressources grain fin, il est nécessaire d'identifier les structures applicatives qui peuvent être implémentées sur les macros blocs de la cible le plus tôt possible dans le flot de synthèse, c'est-à-dire avant que ces structures ne soient cassées en structures plus petites pour une implémentation sur les ressources grain fin. En effet, une fois une structure applicative décomposée en structures grain fin, reconnaitre une fonctionnalité parmi un sous-ensemble des structures grain fin d'une netlist pour remplacer ce sous-ensemble par un macro bloc est trop complexe pour pouvoir être mis en pratique. Par exemple, une fois une multiplication décomposée en LUTs dans une netlist, il est compliqué d'inférer la fonction de multiplication depuis cette netlist. Ainsi, les outils de synthèse amonts (c'est-à-dire synthèse haut niveau et synthèse RTL, qui ont une vue des structures applicative) doivent avoir connaissance des macros blocs disponibles sur la cible afin de pouvoir les instancier pour remplacer les structures applicatives qui le permettent, plutôt que de synthétiser ces structures par des ressources génériques grain fin.

Ainsi, outre les entrées sorties de chaque outil qui correspondent à l'application en cours de synthèse, chaque outil du flot doit aussi prendre en entrée une description de l'architecture cible, qu'il s'agisse des macros blocs disponibles pour

la synthèse amont, du nombre d'entrées par LUT pour le mapping technologique, du plan de calcul détaillé pour le packing, le placement et le routage, les délais atomiques pour l'analyse de timing, et la description du plan de configuration pour l'extraction de bitstream. Lors du choix des outils du flot, il faut donc s'assurer que chaque outil permet de couvrir l'espace de conception de l'architecture cible ; et si aucun outil n'existe qui permet de couvrir cet espace de conception pour une phase du flot, il faut donc le développer ou adapter l'outil existant le plus proche. L'adéquation entre l'outillage logiciel et l'architecture ciblée peut donc demander d'important développement logiciel, et en pratique, pour limiter ces développements, il est avantageux de limiter l'espace de conception de l'architecture ciblée en fonction de ce que permettent les outils existants.

5.2.2 Tester et vérifier le flot de synthèse et les applications

Le flot de synthèse virtuelle est un flot complexe à mettre en place, et qui met en œuvre plusieurs outils, langages et formats de fichier différents. Lors de l'assemblage d'un flot de synthèse virtuelle, partant de la description d'une application pour aboutir à un bitstream virtuel, puis à l'exécution sur carte (après intégration de l'overlay ciblé sur FPGA), il y a peu de chances que le flot soit fonctionnel de bout en bout dès la première exécution. Il est donc nécessaire de pouvoir tester et vérifier le fonctionnement du flot à chacune de ses différentes étapes, comme illustré figure 5.2.

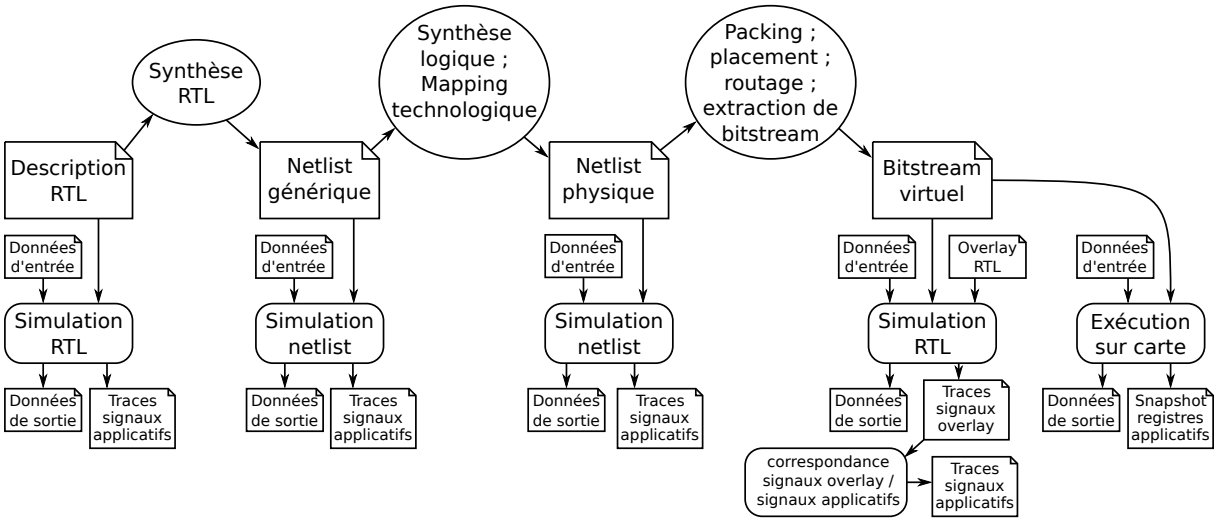


FIGURE 5.2 – Vérifications à chaque étape du flot de synthèse.

La vérification intervient avant même d'entrer dans le flot de synthèse virtuelle, c'est-à-dire lors de l'écriture de l'application, pour s'assurer que l'application écrite a bien le comportement désiré. Par exemple, une application écrite en VHDL ou en Verilog doit être simulée avec un testbench en utilisant un outil de simulation RTL tel que ModelSim [88], GHDL [73] ou Icarus Verilog [89]. Cette première simulation de l'application, dans le langage dans lequel elle est écrite, permet dans un premier temps de la déboguer et d'assurer que la description de l'application

dans son langage d'entrée est correcte et réalise bien la fonctionnalité désirée par son concepteur. En effet, un simulateur RTL donne au concepteur la visibilité sur chacun des signaux de l'application, ce qui facilite le pistage des erreurs.

Dans un deuxième temps, le testbench peut être utilisé pour produire un fichier de données de sorties à partir d'un fichier de données d'entrées. Il est nécessaire que le jeu de données d'entrée couvre le plus largement possible les différents comportements attendus de l'application. En effet, ces jeux de données d'entrées et de sorties constituent le "golden model", et seront utilisés par la suite, lors de la vérification des étapes suivantes, pour s'assurer qu'à chaque étape la simulation prenant en entrée le jeu de données d'entrées produit bien le même jeux de données de sorties que la simulation RTL.

Une fois l'application synthétisée en une netlist, il est nécessaire de simuler cette netlist pour vérifier que la netlist a bien le même comportement que l'application décrite en RTL. Après la synthèse/minimisation logique et le mapping technologique de la netlist en une nouvelle netlist, la nouvelle netlist doit à son tour être simulée. Il n'est pas toujours possible de comparer les traces des signaux des netlists avec les traces des signaux RTL. En effet, bien que la simulation d'une netlist donne la visibilité sur chacun de ses signaux, ceux-ci ne correspondent pas nécessairement aux signaux décrits dans l'application. Par exemple, après des opérations de synthèse séquentielle ou de retiming lors de la synthèse logique, les flip-flops n'ont plus la même place par rapport à la logique, et ne sont plus au même nombre, ce qui rend impossible la comparaison signal par signal de l'application RTL et de sa netlist synthétisé et éventuellement mappée. Cependant, au niveau des signaux d'entrées/sorties de l'application (c'est-à-dire les signaux d'interface de l'application avec l'extérieur), le comportement doit être le même, qu'il s'agisse de la simulation RTL de l'application ou la simulation de la netlist. La vérification de la netlist est donc réalisée en effectuant sa simulation avec les mêmes données d'entrées que la simulation RTL, puis en comparant la sortie des deux simulations.

Dans ces travaux, le format BLIF a été utilisé pour manipuler les netlists. Pour permettre la simulation des netlist avec le même testbench que la simulation RTL, un petit outil a été développé pour traduire les netlists BLIF en VHDL, et ainsi pouvoir simuler la netlist avec un outil classique de simulation RTL (tel que Modelsim ou GHDL).

Après l'obtention d'une netlist mappée, il n'est malheureusement pas possible de réaliser des simulations intermédiaires pendant les phases de packing, placement et routage, jusqu'à l'obtention du bitstream virtuel. Une fois le bitstream virtuel obtenu, celui-ci ne peut pas être simulé tel quel. La dernière étape de simulation consiste donc à effectuer la simulation RTL de l'architecture cible, comprenant le plan de calcul et de configuration. La description RTL de l'architecture cible étant disponible dans un langage HDL (pour la synthèse physique de l'overlay, cf 3.4), cette simulation peut être effectuée avec le même outil que celui utilisé pour la simulation RTL de l'application, comme ModelSim. Lors de cette simulation, le testbench charge le bitstream virtuel dans le plan de configuration de l'architecture simulée, puis les entrées/sorties de l'architecture sont alimentées par le jeu de

données d'entrées, et les sorties sont sauvegardées dans un fichier, qui est ensuite comparé avec la sortie de la simulation RTL.

La dernière étape de vérification consiste à vérifier le comportement du bitstream virtuel sur le matériel physique. Pour cela, il est nécessaire d'avoir à disposition une plateforme d'intégration physique de l'architecture cible (cf chapitre 4) qui permette de configurer l'overlay avec le bitstream virtuel, de l'alimenter avec le jeu de données d'entrées de la simulation RTL, et de récupérer les données de sorties pour la comparer au golden model.

5.2.3 Débogage du flot de synthèse virtuelle et des applications

Les différentes étapes de vérification présentées ci-dessus permettent de déterminer à quelle étape du flot de synthèse virtuelle est apparu un dysfonctionnement éventuel. La granularité de la vérification au niveau du flot correspond donc à :

— la synthèse RTL ;
— la synthèse/minimisation logique ;
— le mapping technologique ;
— le packing, placement, routage et extraction de bitstream ;
— l'exécution sur carte.

Une fois l'étape causant le dysfonctionnement déterminée, il faut alors chercher plus finement la cause du problème parmi les outils et les fichiers d'échange mis en œuvre dans l'étape incriminée. Pour pouvoir remonter depuis le décalage entre la sortie d'une simulation et le golden model, jusqu'à la cause du problème, il est souvent nécessaire de suivre l'état des différents signaux dans la simulation.

Lors d'un problème lors de la synthèse RTL, il est possible de comparer les signaux de la netlist avec les signaux applicatifs. En effet, bien que la netlist produite en sortie de la synthèse RTL présente plus de signaux qu'il n'y a de signaux déclarés dans l'application, les signaux de l'application sont tous présents dans la netlist. Par exemple, si une addition est déclarée dans l'application, l'additionneur sera surement cassé en plusieurs nets et composants lors de la synthèse RTL, mais les signaux (les nets) des opérandes et du résultats seront équivalents entre l'application et la netlist. Aussi, les flip-flops présentes dans la netlist sont équivalentes aux flip-flops déclarées dans l'application.

Lors de la synthèse/minimisation logique et du mapping technologique, la correspondance entre les signaux et des flip-flops de la netlist produite avec ceux de la netlist d'entrée est perdue, du fait du remaniement de la netlist par l'outil de synthèse. Une solution est de synthétiser et simuler les sous modules de l'application séparément tout en comparant les entrées sorties afin de cerner le module dans lequel apparait le problème. Une autre solution est de comparer les comportements combinatoires de la netlist est de l'application, en ne réaliser pas d'opérations modifiant les flip-flops lors de la synthèse logique afin de garder l'équivalence entre les flip-flops de la netlist et celles de l'application.

Après packing, placement, routage et extraction du bitstream, la simulation de l'architecture exécutant le bitstream virtuel couvre non seulement l'exécution applicative, mais aussi l'architecture de l'overlay lui-même. Ainsi, le désaccord entre le résultat de la simulation et le golden model peut indiquer plusieurs choses :

- une erreur de transformation de la netlist mappée en un bitstream virtuel lors des phases de packing, placement et routage ;
- la non-correspondance de l'interprétation du bitstream virtuel par l'architecture cible et l'outil en charge de l'extraction du bitstream (par exemple, pour un multiplexeur d'une Switch Box, la configuration partielle "`01`" signifie sélectionne l'entrée 1, alors que pour l'extracteur de bitstream ça signifie sélectionne l'entrée 2) ;
- une erreur lors de la conception de l'architecture cible ou de la génération de sa description RTL.

Cette simulation ne présente pas directement les signaux applicatifs (c'est-à-dire les signaux de la netlist mappée), mais l'intégralité des signaux implémentant l'overlay. Pour retrouver les traces des signaux applicatifs, il est donc nécessaire de se référer aux fichiers de packing, placement et routage généré la synthèse, pour déterminer quels éléments de l'architecture (et donc la trace de la simulation) qui correspond à quels éléments de la netlist mappée.

Aussi, le nombre important de signaux présents dans les traces de la simulation de l'architecture exécutant le bitstream virtuel rend la lecture et l'analyse de ces traces plus pénibles. En effet, lors de la simulation, la sélection et la lecture des signaux sous forme de chronogramme n'est pas pratique, par exemple lors de l'analyse à un instant donné de la configuration d'une LUT de l'architecture, de ses entrées et de sa sortie. L'utilisateur peut aussi instrumenter son testbench et la description de l'overlay par des assertions. Cependant ces assertions sont vérifiées lors de la simulation et doivent être écrites avant la simulation, or l'utilisateur ne sait pas nécessairement quelles assertions placer avant qu'il n'ait analysé les traces, et l'addition d'une nouvelle assertion demande de relancer toute la simulation.

Pour simplifier cette tâche, un petit outil nommé TraceNav a été développé en Ruby. Il lit les traces VCD [90] (pour Value Change Dump) issues d'un outil de simulation tel que Modelsim ou GHDL, et permet de visualiser les signaux et de vérifier des assertions sans avoir à exécuter à nouveaux la simulation. Pour la visualisation des signaux, TraceNav dispose d'une interface en ligne de commande inspirée d'un shell linux, qui permet de naviguer dans le temps et dans la hiérarchie des modules, via différentes commandes telles que :

cd : modifie le module courant. Par exemple `cd mult_4x4` permet d'entrer dans le module "mult_4x4", `cd ../` retourne au module parent, ou encore `cd /` retourne au top-level de la hiérarchie.

ls : affiche les noms des sous modules et les signaux du module courant.

signal : affiche la valeur des signaux du module courant, ou du signal nommé dans la commande, selon le format souhaité (binaire, hexadécimal, décimal).

go : modifie le temps courant. Permet d'aller à un temps absolu (ex. `go 260ns`), relatif (ex `go +/- 50ns`), ou de se positionner par rapport à un évènement

d'un signal, comme un front montant, descendant, ou un changement de valeur (`go <next|prev> <re|fe|change> <signal_path>`).

Comparée à l'affichage d'un chronogramme, l'interface en ligne de commande permet à l'utilisateur de naviguer intuitivement et rapidement parmi les modules, et de faire abstraction des signaux et des temps qui ne lui sont pas pertinents pour son analyse.

Les fonctions accessibles depuis l'interface en ligne de commande sont aussi accessibles sous forme d'une API (interface de programmation applicative), ce qui permet de tirer parti des fonctionnalités d'un langage de programmation comme Ruby pour mettre en place des assertions complexes, et ce sans avoir à exécuter à nouveau la simulation. Via cette API, TraceNav peut être augmenté pour lire les fichiers de packing, placement et routage issus de la synthèse virtuelle de l'application, de façon à lier les noms des signaux applicatifs de la netlist mappée aux éléments de l'architecture qu'ils empruntent, et ainsi retrouver les valeurs des signaux applicatifs de la netlist applicative dans la simulation de l'architecture chargée par le bitstream virtuel.

Finalement, lors de l'exécution sur carte, le débogage est rendu plus difficile par le fait que l'utilisateur n'a pas de visibilité sur l'évolution des signaux internes à l'overlay (exécution dans le FPGA). Cependant, si le mécanisme de snapshot (cf) a été intégré à l'overlay synthétisé, il est toujours possible d'exécuter l'application en pas à pas (un pas par cycle d'horloge applicative), et d'extraire du snapshot la valeur de chaque registre applicatif affin de comparer ces valeurs avec les simulations précédentes.

Cette méthode et les outils mis en œuvre pour vérifier et déboguer le flot peuvent aussi être utilisés pour déboguer les applications. Cependant, si une application passe le premier test, c'est-à-dire la simulation RTL des sources RTL de l'application, mais ne passe pas les tests avals, cela peut indiquer que le flot de synthèse est en cause plutôt que l'application elle-même, puisqu'une fois le flot opérationnel, le comportement de l'application en cours de synthèse est censé être préservé après chaque étape du flot.

5.3 Réalisations pratique de flots de synthèse

Cette section présente le flot de synthèse virtuelle réalisé pour ces travaux. Celui-ci permet d'utiliser VPR et Madeo.

Comme présenté en 5.1.4, VPR est un outil académique open-source de packing, placement et routage, efficace, largement utilisé dans la recherche, bien documenté, et idéal pour l'exploration fonctionnelle des architectures grain fin de par sa capacité à générer des modèles d'architectures complètement spécifiés à partir d'une description haut niveau.

Le framework académique Madeo permet aussi d'effectuer le placement et rou-

tage de netlists mappées. Contrairement à VPR, Madeo modélise le plan de configuration en plus du plan de calcul, ce qui lui permet de générer le bitstream virtuel de l'application placée et routée pour l'architecture ciblée. Madeo modélise l'architecture cible à partir d'une description. Cette description n'est pas haut niveau (comme celle de VPR), mais est entièrement spécifiée. Ainsi, pour effectuer une exploration rapide de l'espace de conception fonctionnel d'architectures, il est nécessaire d'utiliser un outil externe qui génère une description entièrement spécifiée à partir d'une description haut niveau. En revanche, comme la description d'architecture fournie à Madeo est entièrement spécifiée, l'espace de conception des architectures que Madeo peut manipuler est plus large que celui de VPR. Par exemple, là où le format de description VPR ne permet de spécifier que le nombre de pistes par canal de routage et laisse le générateur de VPR instancier chaque piste, la description Madeo permet de spécifier chaque piste séparément, et ainsi utiliser plusieurs canaux de routage de différentes largeurs dans l'architecture. Ainsi, Madeo peut cibler des architectures irrégulières, grain fin, gros grain ou encore hétérogènes. Madeo doit être vu comme une bibliothèque extensible permettant de réaliser les outils d'un flot de synthèse. Utiliser Madeo pour réaliser un flot ciblant un overlay donné demande donc des développements pour réaliser l'outil désiré et éventuellement adapter la bibliothèque à l'architecture ciblée et aux fonctionnalités désirées. Par exemple, lors des premiers tests avec Madeo, il n'était pas possible de placer et router un circuit qui n'avait que des sorties et pas d'entrées, tel qu'un compteur ou un générateur de nombre pseudos aléatoires. Ainsi, la largeur de l'espace de conception des architectures que peut cibler Madeo demande souvent des développements additionnels. VPR quant à lui n'est pas une bibliothèque mais un outil utilisable directement "out-of-the-box", mais l'espace de conception des architectures qu'il peut cibler en est plus restreint.

Au vu des différents avantages de ces deux outils, nous avons voulu assembler un flot de synthèse virtuelle qui, pour une même application, puisse mettre en œuvre de manière interchangeable VPR ou Madeo. Pour cela, le flot VTR (Verilog To Routing) [91] a été utilisé comme base. Les trois composants principaux de VTR sont :

Odin II : synthèse RTL, du Verilog vers le BLIF,

ABC : synthèse logique et mapping technologique, du BLIF vers du BLIF,

VPR : packing, placement, routage, et analyse de timing, du BLIF vers un fichier de packing, un de placement et un de routage.

Le flot VTR est illustré figure 5.3. Il est utilisable pour réaliser la synthèse virtuelle d'une application écrite en Verilog et ciblant une architecture décrite dans le format d'architecture VPR, pour aboutir au placement et routage. Cependant, il manque à ce flot l'extraction de bitstream. De plus, l'analyse de timing réalisée par VPR concerne des cibles ASICs, elle prend en compte des données physiques comme la résistance et la capacité des pistes selon la technologie ciblée. Dans le cas des overlays, une telle analyse de timing n'est donc pas pertinente. Afin d'obtenir un bitstream virtuel et une analyse de timing propre à l'overlay ciblé, le flot a été complété par deux outils : VirtBitgen et VirtSTA.

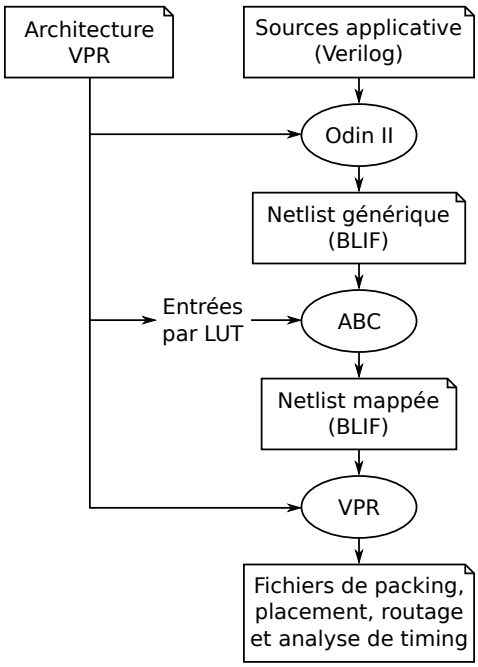


FIGURE 5.3 – Le flot VTR, du Verilog au placement et routage.

5.3.1 Architecture overlay et l'architecture fonctionnelle ciblée par le flot

Les outils de génération de bitstream et d'analyse de timing réalisés dans ces travaux, VirtBitgen et VirtSTA, prennent en entrée les fichiers de packing, placement et routage produits par VPR, ainsi qu'une description de l'architecture ciblée. Cependant, outre la description du plan de calcul de la cible, ces deux outils ont besoin d'informations additionnelles qui ne sont pas comprises dans la description VPR, telles que la relation entre le plan de calcul et le plan de configuration, ou encore les informations sur les timings virtuels des ressources. C'est pourquoi la description de l'architecture fournie à ces deux outils n'est pas la description VPR, mais la description utilisée pour générer l'architecture via l'outil ArGen (cf 3.4).

Bien entendu, il est nécessaire que la description utilisée au début du flot (l'architecture VPR) et la description utilisée à la fin du flot (l'architecture ArGen) correspondent au même overlay. Les deux descriptions de l'architecture doivent donc représenter le même plan de calcul, et il est donc nécessaire que l'une dérive de l'autre. Or, la description ArGen de l'architecture est complètement spécifiée, donc pour conserver l'avantage de VPR concernant l'exploration architecturale par sa capacité à modéliser une architecture entièrement spécifiée à partir d'une description haut niveau, il est donc nécessaire que la description ArGen dérive de la description VPR.

Les descriptions ArGen et VPR étant dans des formats différents, elles peuvent exprimer des architectures différentes. Par exemple, la description ArGen étant entièrement spécifiée, il est possible de décrire des architectures avec des canaux

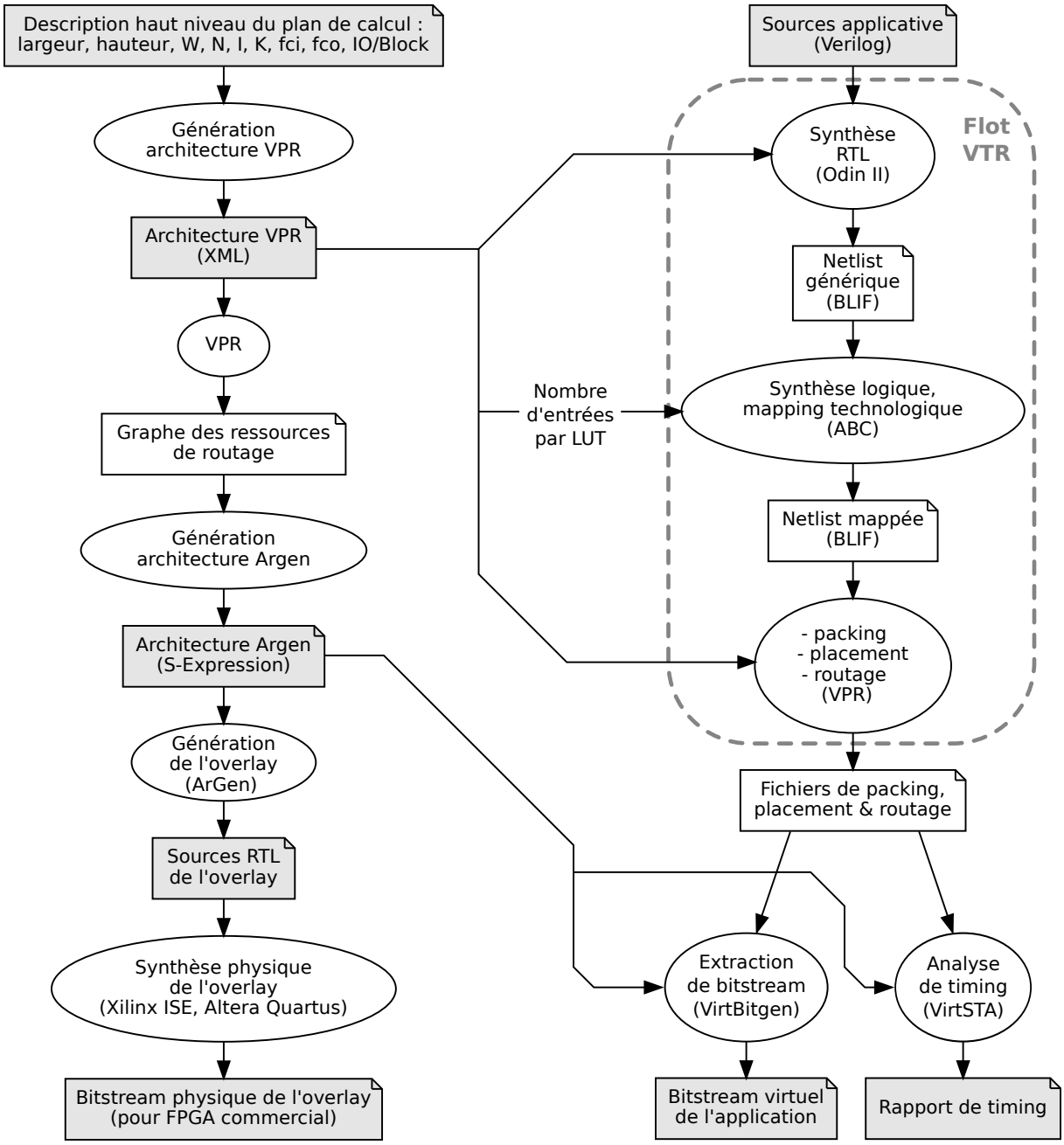


FIGURE 5.4 – Le flot de synthèse virtuelle (droite) couplé avec le flot de génération d'architecture (gauche). La partie entourée de pointillés est le flot VTR original.

de routage de différentes tailles, alors qu'une description VPR n'offre pas cette possibilité. À l'inverse, une description VPR permet de décrire des éléments qui ne sont pas encore pris en compte par ArGen, comme des LUTs fracturables ou des pistes de routage de longueur supérieure à un CLB. L'ensemble des architectures utilisables par ce flot est donc l'intersection de l'ensemble des architectures VPR et de l'ensemble des architectures ArGen. Dans notre flot, pour assurer que l'architecture manipulée peut être modélisée par VPR, la description ArGen est générée à partir de la description VPR ; et pour assurer qu'elle peut aussi être modélisée par ArGen, la description VPR est générée à partir de paramètres haut niveaux, qui

ne permettent que de générer des descriptions VPR modélisables aussi par ArGen.

La figure 5.4 montre le flot de synthèse virtuelle (à droite) ainsi que le flot de génération et de synthèse d'overlay (à gauche). Les paramètres hauts niveaux du plan de calcul (comme le nombre de pistes par canal de routage ou la flexibilité de l'interconnexion des CLB) sont fournis à un script qui les met en forme dans un fichier au format XML attendu par VPR. Ce fichier XML est la description VPR de l'architecture qui sera ensuite utilisée pour la synthèse virtuelle. VPR est ensuite exécuté avec cette description de l'architecture, non pas pour synthétiser un circuit, mais pour produire le graphe de routage entièrement spécifié de l'architecture qu'il a modélisée à partir des paramètres haut niveau compris dans la description XML d'entrée. Le graphe de routage est ensuite lu par un autre script afin de produire la description de l'architecture au format S-expression attendu par les outils ArGen, VirtBitgen et VirtSTA. Cette description est alors fournie à l'outil ArGen pour générer le code RTL implémentant l'overlay (cf 3.4.2). Ces sources RTL peuvent alors être simulées et être synthétisée sur FPGA (cf 4.1). Le flot de génération n'a besoin d'être exécuté qu'une fois, mais il est nécessaire de conserver le couple de descriptions d'architectures aux formats VPR et ArGen pour pouvoir les fournir ensuite au flot de synthèse virtuelle.

Le listing A.1 montre un exemple de description d'architecture au format Argen, la figure A.1 montre une représentation graphique de l'architecture modélisée par ArGen, et la figure A.2 montre le placement et routage d'un multiplieur $5 \times 5 \rightarrow 6$ bits sur cette architecture.

5.3.2 Utilisation de Madeo dans le flot de synthèse virtuelle.

Comme nous l'avons vu dans cette section, VPR est un outil efficace pour l'exploration architecturale et qui fonctionne "out-of-the-box". Cependant, le fait que VPR génère le plan de calcul de l'architecture à partir de quelques paramètres hauts niveaux limite l'espace de conception qu'il permet d'explorer à ces paramètres. Pour explorer un espace plus large, Madeo peut être utilisé dans le flot de synthèse virtuelle comme outil de packing, placement, routage ainsi que pour l'extraction de bitstream et l'analyse de timing.

La figure 5.5 illustre l'utilisation de Madeo dans le flot de synthèse virtuelle. Madeo utilise SIS [86] pour la synthèse logique et le mapping technologique. ABC est le successeur de SIS et produit des netlist dans le même format (BLIF). Ainsi, la partie du flot en amont du packing, placement et routage peut être conservée. Le synthétiseur haut niveau de Madeo peut aussi éventuellement être utilisé pour réaliser la synthèse haut niveau et la synthèse RTL d'une application écrite en Smalltalk vers une netlist BLIF.

Le but de ce flot étant d'explorer un espace de conception architectural plus large qu'avec VPR, c'est-à-dire des architectures qui ne sont pas modélisables par VPR, VPR ne peut donc pas être utilisé pour générer le modèle entièrement spécifié

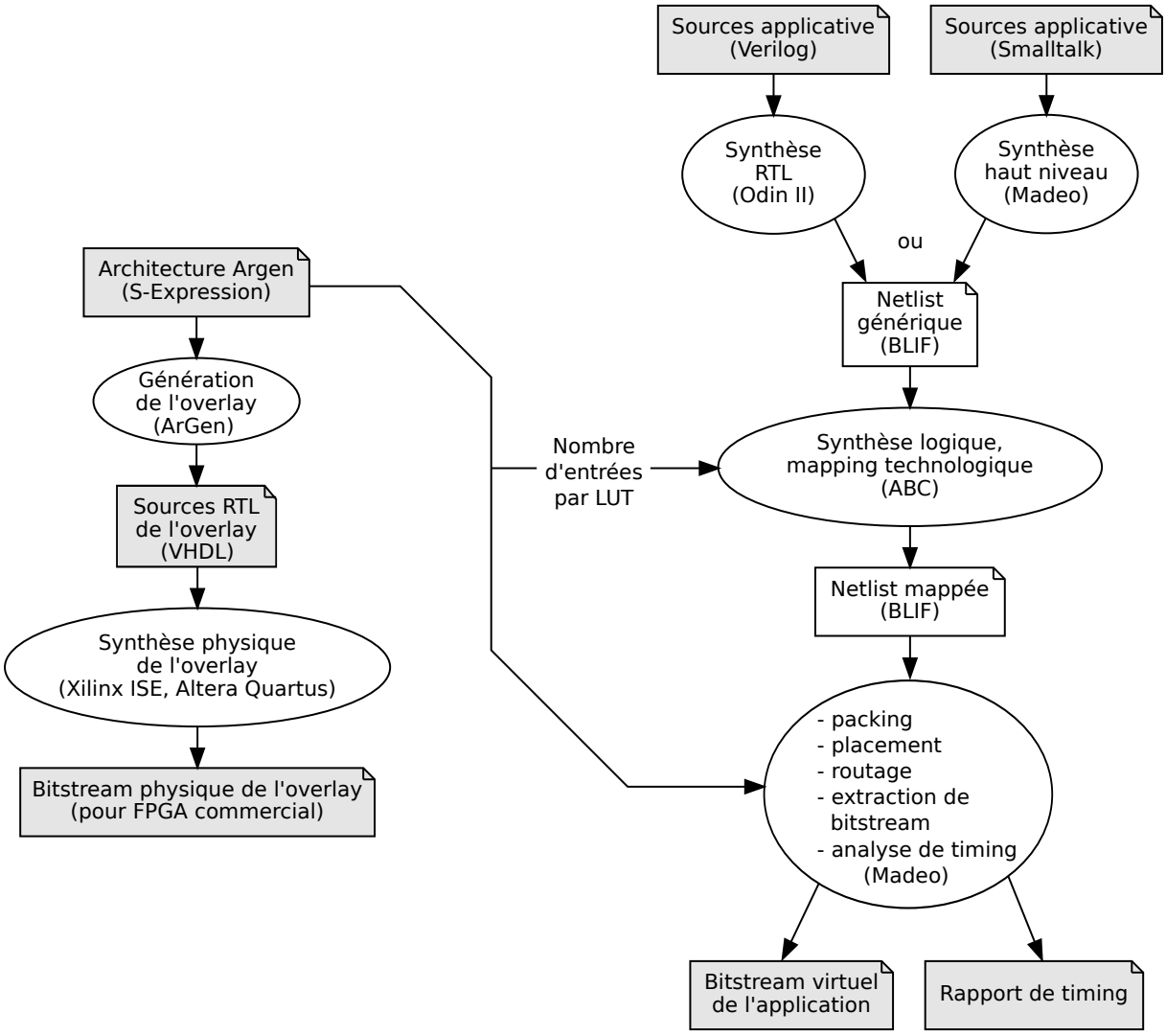


FIGURE 5.5 – Le flot de synthèse virtuelle utilisant Madeo (droite) et flot de génération d'architecture (gauche).

de l'architecture Il est donc nécessaire d'écrire la description de l'architecture ciblée à la main, ou pour faciliter l'exploration, de développer un nouvel outil à cet effet. Afin de conserver la correspondance entre l'architecture générée (description RTL de l'overlay synthétisable) par ArGen et l'architecture ciblée par le flot de synthèse virtuelle, Madeo a été adapté pour lire le format de description d'architecture ArGen. Cette tâche était simple de fait que la description ArGen spécifie entièrement l'architecture, tout comme le format natif de description d'architecture de Madeo.

Dans ce flot utilisant Madeo, il n'y a plus de description d'architecture au format VPR. Ainsi, lorsque Odin II est utilisé comme synthétiseur RTL, il n'est plus possible de lui fournir une description de l'architecture lui spécifiant les macros blocs présents dans l'architecture. Cela n'empêche pas Odin II de mener à bien la synthèse RTL du circuit applicatif en ciblant des LUT et des flip-flops génériques, mais sans description de l'architecture, Odin II ne peut pas inférer des macros blocs à partir de la description RTL. Par exemple, l'outil ne peut pas instancier des macros

blocs DSP à partir d'une multiplication apparaissant dans le code source, étant donné qu'il n'a pas connaissance des macros blocs présents dans l'architecture. Cependant, il est tout de même possible d'instancier explicitement les macros blocs de l'architecture, par exemple pour utiliser des blocs DSP ou des mémoires. Cela demande alors au concepteur de l'application d'avoir à disposition les déclarations des macros blocs présents dans l'architecture.

5.4 L'analyse de timing sur overlay : différentes solutions

5.4.1 Analyse de timing

Lors de la synthèse d'une application, que la cible soit directement le silicium (ASIC), ou une architecture reconfigurable de type FPGA ou encore un overlay, il est nécessaire d'effectuer une analyse de timing du circuit produit pour en obtenir les performances, c'est-à-dire la fréquence maximale de fonctionnement au-delà de laquelle le circuit ne suit plus le comportement selon lequel l'application a été conçue. Cette fréquence maximale de fonctionnement correspond à l'inverse du délai du chemin critique, c'est-à-dire le délai combinatoire maximum parmi tous les chemins combinatoires possibles dans le circuit, partant d'une entrée primaire ou d'un registre et arrivant sur une sortie primaire ou un registre. Ce délai prend en compte les temps de setup et de hold des registres, le délai à travers chaque élément logique et le délai à travers les éléments de routage.

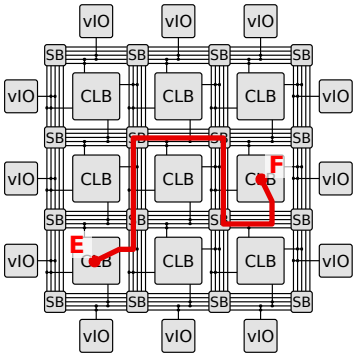
Réaliser l'analyse de timing d'un circuit demande donc d'avoir accès aux délais de chaque ressource du circuit, ou du moins d'avoir les données physiques qui permettent de les calculer. De plus, lors de la synthèse d'une application, l'analyse de timing permet de guider les outils de placement et de routage dans le compromis performance/surface afin de respecter les contraintes de timing indiquées par le concepteur de l'application (par exemple une fréquence minimum de fonctionnement). Un exemple de méthode utilisée pour obtenir le délai d'une piste de routage est l'approximation d'Elmore [92] reposant sur la modélisation de la résistance et de la capacité électrique des pistes. Dans le cas des overlays, une telle modélisation n'est pas possible, il faut donc trouver une méthode d'analyse de timing adaptée aux cibles overlays.

5.4.2 Spécificités de l'analyse de timing sur overlay

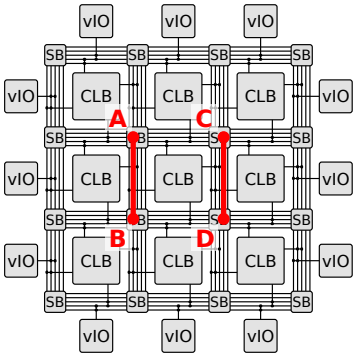
L'usage d'un overlay comme architecture cible pose des difficultés intéressantes quant à l'analyse de timing. En effet, l'overlay étant lui-même synthétisé sur une architecture reconfigurable de type FPGA, les délais des ressources atomiques de l'overlay dépendent à la fois de la qualité de la synthèse physique de l'overlay et des

performances de l'architecture sous-jacente. Après la synthèse physique de l'overlay, il est donc nécessaire de caractériser le délai de chaque ressource atomique de l'overlay pour permettre à l'outil de synthèse virtuelle de calculer le délai des chemins combinatoires du circuit virtuel empruntant ces ressources.

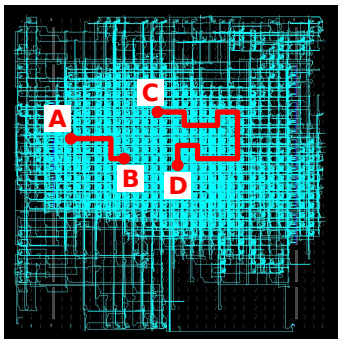
Cependant, contrairement à un FPGA physique dont le layout est régulier, deux ressources atomiques de l'overlay qui sont conceptuellement équivalentes (par exemple deux pistes de routage ayant la même longueur) peuvent avoir des délais différents dans l'implémentation physique de l'overlay sur son hôte. Ce problème est illustré figure 5.6 : pour déterminer le délai de chemin applicatif allant du point E au point F, l'outil doit entre autres connaitre le délai des pistes de routage (A, B) et (C, D) de l'overlay. Bien que ces deux pistes virtuelles aient la même longueur conceptuelle dans le plan de calcul de l'overlay, il peut résulter du placement et routage physique de l'overlay sur le FPGA hôte que ces deux pistes aient des délais différents. La taille de la base de données associant un délai à chaque ressource de l'overlay est donc proportionnelle à la taille de celui-ci et non au nombre de ressources différentes trouvées dans son plan de calcul ; ceci complexifie l'outil de synthèse virtuelle et ralentit l'analyse de timing lors de la synthèse d'applications.



Routage du chemin applicatif (E, F).
Ce chemin passe par les pistes
de routage (A, B) et (C, D) de l'overlay.



Pistes de routage (A, B) et (C, D)
du plan de calcul de l'overlay.
Ces deux pistes ont la même longueur.



Placement et routage physique de
l'overlay sur son FPGA hôte.
Les pistes (A, B) et (C, D) du plan de calcul
de l'overlay n'ont pas la même longueur
ni le même délai physique.

FIGURE 5.6 – L'analyse de timing de circuits applicatifs sur overlay doit prendre en compte les délais physiques des éléments de l'overlay résultant de la synthèse de celui-ci sur le FPGA hôte.

De plus, cette base de données nécessaire à la synthèse virtuelle sur un overlay est dépendante de l'instance de l'overlay synthétisé : les caractérisations des délais d'un même overlay synthétisé sur différents FPGA seront différentes. Pire encore, à cause des heuristiques utilisées lors de la synthèse physique sur FPGA, la caractérisation des délais d'un overlay donné sur un même FPGA peut varier d'une synthèse à l'autre. Ainsi, pour une application donnée, ciblant un overlay donné, l'analyse de timing de l'application doit être effectuée pour chaque instance de l'overlay synthétisé qui l'exécutera. Aussi, les optimisations effectuées lors de la synthèse virtuelle d'une application qui reposent sur l'analyse de timing (placement et routage "timing driven") ne sont donc valides que pour une instance d'overlay.

Finalement, il est difficile d'extraire les délais des ressources atomiques de l'overlay lors de la synthèse physique de l'overlay sur FPGA : du fait des transformations et des optimisations effectuées sur la netlist de l'overlay par l'outil de synthèse physique, des éléments sont fusionnés et leur nom disparaissent de la netlist, ce qui rend impossible d'obtenir certains délais de signaux apparaissant dans le code source RTL de l'overlay. De plus, contrairement aux outils ASICs, les outils FPGAs fournis par leurs vendeurs ne permettent pas aisément d'obtenir un délai combinatoire ne correspondant à un "timing path", c'est-à-dire ne partant et n'aboutissant pas d'un élément séquentiel à un autre, ce qui est le cas de toutes les ressources atomiques du plan de calcul d'un overlay, qui partent d'un multiplexeur et aboutissent à un autre. Par exemple, dans l'illustration figure 5.6, il est difficile d'obtenir les délais du point A au point B et du point C au point D.

Aussi, dans la littérature [51, 93], cette méthode (extraire les délais des ressources atomiques lors de la synthèse physique, pour les remonter à l'outil de synthèse virtuelle) est toujours considérée pour des travaux futurs, mais n'a à ce jour pas encore été mise en œuvre. Les deux sous-sections suivantes présentent d'autres solutions qui ont été utilisées dans la littérature.

5.4.3 Approches top-down : du circuit applicatif au délais physiques

Utiliser l'outil de synthèse physique pour l'analyse de timing virtuelle

Comme présenté ci-dessus, il est difficile d'extraire les délais des ressources atomiques de l'overlay lors de la synthèse physique de celui-ci. Cependant, l'outil de synthèse physique a cette information. Dans [93, 94], les auteurs synthétisent des architectures reconfigurables grain fin sur ASIC, ce qui présente les mêmes difficultés que pour les overlays sur FPGA quant à l'analyse de timing des circuits mappés sur l'architecture synthétisée. Les auteurs réalisent l'analyse de timing du circuit virtuel en utilisant l'outil de synthèse physique (faisant partie du flot ASIC). Pour ce faire, ils placent des exceptions sur touts les chemins de l'architecture inutilisées par le circuit virtuel (qu'ils taguent comme “false path”). Ces exceptions indiquent à l'outil d'analyse de timing les chemins de l'architecture à ignorer. Si le circuit virtuel est correctement conçu, il ne présente pas de boucles combinatoires, et donc l'ensemble des chemins analysés par l'outil (les chemins n'ayant pas d'exception) ne présente pas non plus de boucles combinatoires, et peut être analysé correctement.

Cette méthode peut être utilisée pour les overlays pour réaliser une analyse de timing précise des circuits virtuels (étant réalisée par une partie du flot physique). Cependant, elle nécessite, pour chaque circuit virtuel, d'utiliser le flot physique. Ce va-et-vient entre le flot virtuel et le flot physique, nécessaire lors de la synthèse de chaque circuit virtuel, peut ne pas être désirable lors de l'utilisation d'un overlay. Notamment, cette méthode ne va pas dans le sens de l'abstraction et de l'isolation

de l'overlay de son support physique.

Utiliser une fréquence pessimiste

Dans [51], les auteurs étendent l'implémentation de référence de ZUMA [23] pour y intégrer des registres applicatifs et ainsi permettre la synthèse de circuits applicatifs séquentiels. Pour assurer le fonctionnement correct du circuit applicatif, ils utilisent comme fréquence maximale de fonctionnement du circuit applicatif la fréquence maximale de fonctionnement de l'overlay, indiquée lors de la synthèse physique de l'overlay sur FPGA. Cette fréquence ne prend donc pas en compte une quelconque configuration de l'overlay, et est calculée par l'outil de synthèse physique comme le plus long chemin combinatoire possible dans l'overlay. Ainsi, cette fréquence, bien qu'elle assure le fonctionnement correct de tout circuit applicatif synthétisé sur l'instance de l'overlay, est extrêmement pessimiste.

5.4.4 Approches bottom-up : des délais physiques à l'analyse du circuit applicatif

Mesurer les délais des ressources atomiques de l'overlay

Le seul exemple d'analyse de timing sur un overlay grain fin trouvé dans la littérature est donné dans [30]. Les auteurs ont réalisé un overlay grain fin qu'ils synthétisent sur un FPGA. Ensuite, plutôt que d'extraire les délais des ressources atomiques via l'outil de synthèse physique, ils mesurent expérimentalement ces délais en utilisant différentes configurations virtuelles.

Cette méthode pose plusieurs problèmes. Premièrement, le nombre de configurations virtuelles et de mesures nécessaires pour estimer le délai de chaque ressource atomique croit avec la taille de l'overlay. Ensuite, ces mesures varient en fonction de paramètres tels que la température, et du vieillissement du FPGA sur lesquelles sont réalisées les mesures. Aussi, les délais mesurés sont liés au FPGA utilisé (non seulement le modèle de FPGA, mais le FPGA physique lui-même), et aussi à une synthèse particulière de l'overlay sur ce FPGA. C'est-à-dire que pour deux synthèses du même overlay sur le même FPGA, les délais de chaque ressource atomique peut varier. Ainsi, il n'est pas possible d'utiliser deux synthèses de l'overlay : une pour l'exploitation et une pour la caractérisation, qui inclue le mécanisme de mesure des délais. L'overlay est donc inséparable de son mécanisme de caractérisation, même si ce mécanisme est les ressources qu'il consomme sont inutilisés lors de son exploitation, une fois l'overlay caractérisé.

Contraindre la synthèse physique de l'overlay

Pour abstraire les caractéristiques des délais dans le plan de calcul d'un overlay et les décorréler d'une synthèse physique particulière et de la plateforme sous-jacente, une solution est de contraindre le délai maximum de chaque type de ressource atomique de l'overlay lors de sa synthèse physique, par exemple contraindre toutes les pistes de routage de même longueur dans le plan de calcul à avoir le même délai. Ces contraintes peuvent être générées en même temps que le code source RTL de l'overlay, et par le même outil. Si l'outil de synthèse physique n'arrive pas à respecter les contraintes, une solution est de multiplier les délais de toutes les contraintes par un même facteur k jusqu'à ce que la synthèse aboutisse en respectant les contraintes. Ainsi, l'analyse de timing effectuée lors de la synthèse virtuelle est plus simple et plus rapide, car les délais des ressources atomiques de l'overlay ne sont plus uniques pour chaque ressource, mais unique par type de ressource, ce qui diminue grandement la base de données de délai à manipuler. À titre d'illustration, le plan de calcul de l'architecture minimale vFPGA-flexible de 3×3 CLBs de 4 LUT-4 par CLB et de 8 pistes par canal de routage, représentée figure A.1 en annexe A, comporte 1260 ressources atomiques. C'est autant de délais à caractériser par instances de cet overlay et à manipuler par l'outil de synthèse virtuelle. Si ces ressources sont contraintes par types, l'outil n'a plus à manipuler que 7 délais : les pistes de routage (noir), les entrées des IOs et des CLBs (bleu), leurs sorties (rouge), les SBs (vert), les crossbars des CLBs et les LUTs.

Cette démarche va dans le sens de la virtualisation et de l'isolation de l'overlay de son support physique. En effet, lors de la synthèse virtuelle, les optimisations de placement et routage reposants sur l'analyse de timing de l'application virtuelle seront valides sur toutes les instances de l'overlay ciblé, car les proportions entre les délais des ressources atomiques de l'overlay seront les même d'une instance à une autre (le facteur k mentionné ci-dessus). Les performances d'une application virtuelle sont alors prédictibles d'une instance d'overlay à une autre, qu'elles soient synthétisées avec différentes options de synthèse et/ou sur différents FPGAs. Lors de la synthèse virtuelle, pour mener à bien son analyse de timing, l'outil n'a besoin que des délais de base de chaque type de ressource (ce qui est propre à un overlay et non à une instance d'overlay), et le facteur k (multipliant ces délais de base) propre à chaque instance. Les caractéristiques physiques d'une instance d'overlay nécessaire à la synthèse virtuelle sont abstraites en un seul scalaire : le facteur k.

Cependant, même si contraindre les délais des chemins combinatoires est une pratique courante dans le design d'ASIC, contraindre finement chaque ressource de l'overlay n'est pas aisé avec les outils de synthèse FPGA, pour les même raisons citées plus haut pour l'extraction des délais : il est difficile de cibler et contraindre un segment combinatoire à partir des noms des signaux apparaissant dans le code source RTL de l'overlay lorsque ce segment ne part pas et n'aboutit pas à un élément séquentiel (i.e. ce segment n'est pas un timing path). Et bien que l'outil de synthèse physique permette d'ajouter des contraintes pour empêcher les optimisations sur les éléments ciblés et ainsi les garder dans la netlist après synthèse logique (flot constructeur) tout en conservant leurs noms et permettant ainsi de les

cibler par des contraintes, il est difficile de mettre en place toutes les contraintes qui couvrent chaque ressource de l'overlay sans qu'une contrainte n'en casse une autre. Dans ces travaux, nous ne sommes pas parvenu à contraindre de manière satisfaisante toutes les ressources d'un petit overlay de test avec l'outil ISE de Xilinx, et nous n'avons pas essayé avec les outils Altera. Bien qu'un expert en synthèse FPGA avec le flot Xilinx serait surement parvenu à réaliser cette tâche, notre échec montre que l'utilisation des contraintes est une solution qui risque d'être très liée à chaque outil constructeur et de ne pas être une solution simple et portable.

Notre solution, présentée dans la section suivante, va également dans le sens de la démarche de virtualisation : nous cherchons à virtualiser les temps combinatoires en donnant une vue logique de ce qui est analogique.

5.5 Notre solution pour l'analyse de timing : les VTPRs

La solution qui a été retenue dans ces travaux permet de virtualiser la propagation des signaux applicatifs dans l'overlay via la "quantification" les chemins combinatoires de l'hôte. Il s'agit d'injecter des registres additionnels, au niveau de l'implémentation RTL du plan de calcul de l'overlay, de manière à isoler les ressources atomiques les unes des autres par des éléments séquentiels.

En pratique, ces registres sont placés à la sortie de chaque multiplexeur implémentant une ressource configurable telle qu'une piste de routage ou une entrée d'un CLB. Ces registres additionnels sont les VTPRs (Virtual Time Propagation Registers) qui ont déjà été mentionnés en 3.3 à propos de la synthèse physique de l'overlay. En plus de casser les boucles combinatoires et de diminuer le nombre de timings paths qui seraient présents dans l'implémentation de l'overlay si les VTPR étaient absents et ainsi de diminuer le temps de synthèse physique de l'overlay et d'augmenter la qualité du circuit produit, en délimitant chaque ressource atomique du plan de calcul de l'overlay, les VTPRs permettent aussi à l'outil de synthèse physique d'optimiser sa synthèse de façon à ce que chaque ressource atomique ait le même délai minimum.

Concernant l'analyse de timing des circuits virtuels, les VTPRs jouent ainsi le même rôle que des contraintes sur le délai des ressources atomiques de l'overlay, en attribuant un même délai pour chaque type de ressource (cf approche 5.4.4). Cependant, les délais considérés ne sont plus analogiques, mais logiques, et correspondent à des cycles d'une horloge physique. La fréquence maximale de cette horloge physique dépend des performances du FPGA hôte, mais l'analyse de timing virtuelle réalisée en considérant ces délais logiques reste valide quel que soit l'hôte et la qualité de la synthèse physique de l'overlay. Cette l'analyse de timing d'un circuit virtuel n'a donc besoin d'être réalisée qu'une seule fois. La "quantification" des chemins du plan de calcul par les VTPRs permet donc d'isoler complètement l'overlay de son hôte, et d'abstraire les performances des circuits applicatifs de celles du

FPGA hôte.

De plus, les VTPRs sont beaucoup plus simples à mettre en place que les contraintes de délai (cf 5.4.4), car il s'agit de règles RTL traditionnelles afin de maîtriser un design. Contrairement aux contraintes sur les délais combinatoires, les VTPRs apparaissent directement au niveau des sources RTL de l'overlay et sont interprétés de la même façon et sans ambiguïtés par les différents outils de synthèse physique des vendeurs FPGA, rendant ainsi le design de l'overlay complètement portable. L'outil de synthèse physique se charge lui-même de minimiser le délai de toutes les ressources atomiques de l'overlay et fait au mieux, contrairement à l'utilisation des contraintes qui demanderait des interventions manuelles pour ajuster celles-ci afin d'obtenir une implémentation de l'overlay satisfaisante.

Dans la sous-section 5.6.1, l'avantage d'utiliser les VTPRs pour l'analyse de timing par rapport à l'utilisation d'une fréquence pessimiste (cf 5.4.3) pour les circuits applicatifs est évalué. Les VTPRs garantissent une fréquence maximale de fonctionnement applicatif en moyenne $27\times$ supérieure à la fréquence pessimiste retournée lors de la synthèse de l'overlay sur le FPGA.

5.5.1 L'analyse de timing avec les VTPRs

Lors de la synthèse physique de l'overlay, l'outil indique la fréquence maximum de l'horloge physique de l'instance de l'overlay synthétisée qui garanti le bon fonctionnement de celui-ci. Nous appelons cette fréquence f_{VTPR}. L'outil garanti ainsi que le délai maximum parmi toutes les ressources atomiques de l'overlay est $d_{max} = \frac{1}{f_{VTPR}}$. Lors de la synthèse virtuelle, l'outil calcule le délai d'un chemin combinatoire (i.e. combinatoire au niveau de l'application, partant et aboutissant d'un registre applicatif à un autre) comme le nombre de ressources atomiques du plan de calcul de l'overlay empruntées par ce chemin. Nous appelons ce nombre la *longueur* du chemin. Tout comme pour une synthèse classique, le chemin critique applicatif est le chemin virtuel allant d'un registre applicatif à un autre et ayant la plus grande longueur, celle-ci étant liée au délai du chemin. Les performances du circuit applicatif synthétisé sont représentée par la grandeur $f_{virt} = \frac{1}{longueur_{critique}}$, qui est liée à l'architecture de l'overlay ciblé mais est indépendante d'une instance donnée de cet overlay. Étant donné une instance de l'overlay ciblé, ayant une fréquence maximale physique f_{VTPR} et donc un délai par ressource atomique de $d_{max} = \frac{1}{f_{VTPR}}$, le délai réel du chemin critique applicatif est $d_{reel_{critique}} = longueur_{critique} \times d_{max}$, et la fréquence d'exécution réelle de ce circuit sur cette instance est $f_{reelle} = f_{VTPR} \times f_{virt}$. Les performances d'un circuit applicatif sont ainsi abstraites des instances physiques de l'overlay qu'il cible, et l'outil de synthèse virtuelle n'a pas à manipuler de délais physiques. Aussi, optimiser les performances abstraites f_{virt} d'un circuit augmente ses performances réelles f_{reelle}, et augmenter les performances physiques d'une instance d'overlay augmentera de même les fréquences réelles f_{reelle} des circuits applicatifs qu'il exécutera.

Notons ici que les VTPRs ne sont pas vus par l'outil de synthèse virtuelle comme

des éléments séquentiels du plan de calcul de l'overlay, et n'ont pas du tout le même rôle que les registres applicatifs. Les VTPRs ne sont vus par l'outil que comme des points de passage des signaux virtuels lui permettant de déterminer la longueur d'un chemin. La figure 5.7 met en parallèle un élément de circuit implémenté sur un FPGA classique (en haut) et sur un overlay utilisant des VTPRs (en bas). Ce circuit met en œuvre des registres (bleu), une LUT (jaune) et du routage (nuages gris). Dans le cas d'un FPGA classique, le délai du chemin critique du circuit illustré figure 5.7 est la somme des délais de routage d_r et des délais logiques d_l, soit $d_{critique} = \max(d_{r_1}, d_{r_2}) + d_{l_1} + d_{r_3}$. Dans le cas de la synthèse virtuelle, le chemin critique traverse trois VTPRs, soit quatre ressources atomiques, la longueur critique est donc de quatre. Il faut ainsi que le signal d'horloge virtuelle soit activé au minimum tous les quatre cycles d'horloge physique (l'horloge des VTPR) pour permettre à tous les signaux applicatifs du circuit de se propager jusqu'à leur registre applicatif de destination et ainsi assurer le fonctionnement correct du circuit.

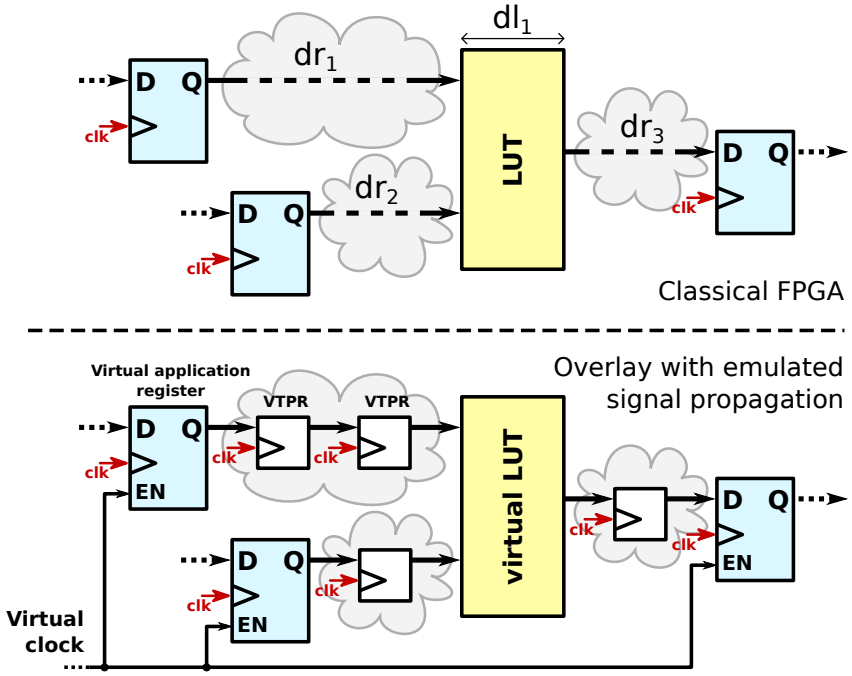


FIGURE 5.7 – Analogies entre le calcul des délais pour un FPGA classique et pour un overlay avec VTPRs

La figure A.2 montre le placement et routage d'un multiplieur $5 \times 5 \rightarrow 6$ bits sur une architecture. L'outil d'analyse de timing VirtSTA indique que le chemin critique applicatif traverse 19 VTPRs. Si la fréquence de fonctionnement f_{VTPR} de l'overlay sur le FPGA est de 200 MHz par exemple, alors la fréquence réelle maximale de l'application sera de 10.5 MHz.

5.5.2 Limitations des VTPRs

Les VTPRs sont des registres qui émulent la propagation des signaux dans le plan de calcul de l'overlay, dont le but est de faciliter la synthèse physique de l'over-

lay sur FPGA et de permettre une analyse de timing simple et indépendante d'une instance d'overlay synthétisé. Bien que la présence des VTPRs parmi les ressources de routage de l'overlay puisse faire penser à un pipeline, les VTPRs ne peuvent pas être utilisés pour pipeliner les signaux virtuels. De même, les VTPRs ne peuvent pas être utilisés non plus pour mettre en place la technique de C-slowing [95], qui utilisée conjointement au retiming permet d'augmenter le débit de fonctionnement d'un circuit. Le pipelining et le C-slowing ne peuvent se faire simplement qu'au niveau applicatif, via les registres applicatifs, pour lesquels l'outil de synthèse virtuelle à la main.

Ceci peut se comprendre en observant la partie basse de la figure 5.7 : pour pouvoir pipeliner le circuit, il faudrait qu'il y ait le même nombre de VTPRs dans les deux nuages à gauche de la LUT virtuelle. Il faudrait donc contraindre l'outil de synthèse virtuelle afin d'égaliser le nombre de VTPRs dans les deux sections de routage. Or, l'outil de synthèse n'a pas la main sur les VTPRs, c'est-à-dire que les VTPRs ne peuvent pas être contournés, à l'inverse des registres applicatifs dans les BLEs qui peuvent être contournés via un multiplexeur contrôlé par un bit de configuration. Si l'architecture permettait de contourner de manière configurable les VTPRs comme c'est le cas de l'architecture HyperFlex d'Altera [96] (c'est-à-dire en permettant un chemin combinatoire entre les ressources atomiques), les VTPRs n'isoleraient plus les ressources atomiques du plan de calcul, et cela annulerait donc les deux raisons d'être des VTPRs : faciliter la synthèse et garantir les délais des ressources atomiques. Ainsi, la seule façon de contraindre l'outil de synthèse virtuelle pour égaliser le nombre de VTPRs traversés par deux chemins applicatifs et de jouer sur la longueur de ces chemins lors du routage. Le problème du routeur n'est alors plus seulement de trouver une route d'une source A à une destination B pour chaque net tout en évitant les congestions et en minimisant le chemin critique, mais aussi de faire en sorte que chaque route ait une longueur précise (ie traverse un nombre précis de registres). Ce problème, connu comme le "N-Delay Routing problem", est expliqué dans [97]. Il s'agit d'un problème NP-complet, et les heuristiques qui permettent de le résoudre sont plus coûteuses en termes de calcul que l'algorithme classique PathFinder [98] de routage "timing driven".

Les overlays gros grain sont faits pour accélérer des applications en flux de données. Ils sont ainsi moins génériques que les architectures grain fin, mais les applications qu'ils ciblent ne présentent pas de boucles de rétroactions et à ce titre peuvent toutes bénéficier de pipelining. Ces architectures présentent des registres similaires aux VTPRs, placés le long des ressources de routage et ne pouvant pas être contournés par un chemin combinatoire, et dont le rôle et de permettre la synthèse physique de l'overlay tout en augmentant sa fréquence de fonctionnement. Pour mettre à profit ces registres dans le pipelining des applications et ainsi faire corréler l'horloge applicative avec l'horloge physique de l'overlay, et ce sans avoir à utiliser un routeur complexe prenant en compte le "N-Delay Routing problem", les architectures telles que [34, 24] intègrent à chaque entrée de leurs unités fonctionnelles des registres à décalage de longueur reconfigurables. Après un routage classique de l'application, l'outil de synthèse virtuelle peut alors jouer sur la longueur de chacun de ses registres à décalage pour égaliser la latence de chaque chemin sans avoir à jouer sur le routage des signaux applicatifs pour y parvenir.

Ce mécanisme de registre à décalage à longueur reconfigurable pourrait être utilisé sur les overlays grain fin. Cependant, les overlays grain fin ont des unités fonctionnelles (les LUTs) en bien plus grand nombre que pour les overlays gros grain. Ajouter des registres à décalages à longueur configurable à chaque entrée de chaque LUT de l'architecture aurait donc un impact très important sur la surface physique occupée par l'overlay. Or, ces registres à décalages additionnels ne seraient profitables (et leur surcoût ne serait justifiable) que pour une sous partie des applications synthétisables sur un overlay grain fin, c'est-à-dire les applications ne présentant pas de boucle de rétro action et qui peuvent donc être pipelinées.

5.5.3 L'analyse de timing dans le flot de synthèse virtuelle

De manière générale, le placement et routage d'un circuit sur une architecture reconfigurable se fait suivant un compromis performance/surface. Suivant le choix du concepteur du circuit, l'outil essaye soit de diminuer l'utilisation des ressources de la cible au prix d'un chemin critique plus long, ou à l'inverse de diminuer le chemin critique au prix d'une occupation plus importante en ressources (on parle alors d'un placement et routage "timing driven"). Ainsi, lors de la synthèse, l'outil de placement et routage a besoin de réaliser régulièrement des analyses de timing sur le circuit applicatif en cours de synthèse pour estimer le délai du chemin critique et pour le guider dans sa tâche. Le compromis performance/surface est aussi pertinent dans le cas d'un circuit virtuel synthétisé sur un overlay. L'analyse de timing virtuelle peut donc être réalisée dans l'outil de placement et routage pour guider la synthèse dans le cas d'un routage timing driven. Dans ces travaux, l'outil Madeo a été adapté pour prendre en compte les VTPRs pour l'analyse de timing.

Cependant, comme il a été vu précédemment, dans le cas de l'outil VPR, l'analyse de timing réalisée vise des cibles ASIC est n'est donc pas pertinente pour un overlay avec VTPRs. Le problème qui se pose est donc que lors du placement et routage, le chemin que VPR essaye de minimiser (du fait que son analyse de timing lui indique qu'il s'agit du chemin critique), n'est en réalité pas nécessairement le vrai chemin critique (en considérant les VTPRs). VPR est ainsi mal guidé et le circuit synthétisé est sous optimal (ie n'a pas le rapport performance/surface) qui aurait pu être obtenu si l'analyse de timing de VPR était en mesure d'obtenir le vrai chemin critique.

La solution la plus efficace pour résoudre ce problème est de modifier VPR pour qu'il prenne en compte les VTPRs dans son analyse de timing, comme cela a été fait avec Madeo. Cependant, pour éviter de rentrer dans des développements importants pour modifier VPR, un petit outil indépendant de VPR a été réalisé pour effectuer l'analyse de timing avec VTPRs après placement et routage. Il s'agit de l'outil VirtSTA, visible en bas de la figure 5.4. VirtSTA lit la description d'architecture au format ArGen ainsi que les fichiers de packing, placement et routage produits par VPR. Il détermine le chemin critique du circuit applicatif qui présente le plus grand nombre de VTPRs traversés (N_{VTPR}) et en déduit la fréquence virtuelle $f_{virt} = \frac{1}{N_{VTPR}}$. L'outil VirtSTA permet d'avoir un rapport de timing exact

du circuit synthétisé après placement et routage, mais ne permet pas de guider VPR lors de son placement et routage.

Pour augmenter la qualité du placement et routage de VPR (rapport fréquence/surface du circuit), VPR doit pouvoir estimer au mieux le chemin critique du circuit. Pour cela, la description d'architecture fournie à VPR indique des délais (là où VPR le permet, comme le délai des LUTs, du crossbar des CLBs ou encore des multiplexeurs des SBs) de façon à approcher le plus possible la séparation des ressources atomiques du plan de calcul par les VTPRs. Ces délais ne correspondent pas à une instance d'overlay particulière, et leur valeur n'a pas d'importance : c'est leurs proportions entre eux qui est importante et qui doit respecter la réalité de l'architecture de l'overlay (par exemple, un chemin qui traverse deux VTPRs doit être vu par VPR comme ayant un délai deux fois plus important que celui d'un chemin qui ne traverse qu'un VTPR).

Cependant, l'indication de ces délais est un "hack" pour utiliser l'analyseur de timing de VPR pour un overlay avec VTPRs, et n'est pas suffisante pour permettre à VPR de déterminer exactement le chemin critique. En effet, VPR calcul le délai des nets suivant la modélisation RC des ressources de routage, ce qui n'est pas approprié pour une architecture avec VTPRs. Prenons l'exemple des nets représentés figure 5.8. Le net de gauche part d'une source S_1 et aboutit à deux destinations A et B. Les chemins $S_1 \rightarrow A$, $S_1 \rightarrow B$ et $S_2 \rightarrow C$ traversent tous trois VTPRs. Ils ont donc le même délai abstrait, qui vaut 3 (soit un délai physique de 3 cycles d'horloge Clk_{VTPR}). Cependant, le premier net utilise sept pistes de routage, alors que le premier n'en comporte que quatre. Ainsi, la modélisation RC réalisée par VPR attribue au premier net une capacité supérieure à celle du second, et détermine ainsi un délai $S_2 \rightarrow C$ inférieur aux délais $S_1 \rightarrow A$ et $S_1 \rightarrow B$. Cette analyse de timing non appropriée peut conduire VPR à faire des choix de placement et routage qui mènent à un circuit moins optimal que si les VTPRs étaient correctement pris en compte.

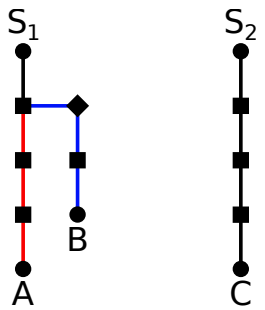


FIGURE 5.8 – Deux nets. Les segments (S_1, A), (S_1, B) et (S_2, C) ont la même longueur.

5.6 Évaluation du surcoût de la couche de virtualisation

Maintenant que nous avons un outillage permettant de synthétiser des applications sur un overlay, nous sommes en mesure d'évaluer le surcoût de la couche de virtualisation apportée par l'overlay, au niveau des ressources additionnelles du FPGA hôte, ainsi qu'au niveau de la perte de performances. Pour cela, cinq applications sont utilisées comme benchmarks. Elles sont décrites dans le tableau 5.1.

Application	Description
cordic	Fonction trigonométrique, phase et résultat sur 16 bits.
cmult	Multiplication combinatoire, opérandes 16 bits, résultat 32 bits.
pmult	Multiplication pipelinée, opérandes 8 bits, résultat 16 bits.
cdivmod	Division et modulo combinatoires, opérandes et résultats sur 16 bits.
filtre RII	Filtre RII d'ordre 4, multiplication et accumulation séquentielle sur 20 bits.

TABLE 5.1 – Descriptions des applications utilisées comme benchmarks.

Dans un premier temps, ces applications ont été synthétisées sur un overlay vFPGA-flexible (cf 3.1.2) de 14×13 CLBs comprenant chacun 4 BLEs avec des LUTs à 4 entrées, et avec 16 pistes par canal de routage. Le tableau 5.2 liste le résultat de ces synthèses virtuelles dans la partie *Synthèse sur Overlay*. La colonne *BLEs* indique le nombre de BLEs de l'overlay utilisés par chacun des circuits applicatifs. La colonne *Délai* N_{VTPR} indique le nombre maximum de VTPRs traversés par un chemin applicatif, c'est-à-dire la longueur du chemin critique en VTPRs. La fréquence virtuelle des circuits applicatifs est $f_{virt} = \frac{1}{N_{VTPR}}$.

Dans un deuxième temps, l'overlay ciblé par ces applications est synthétisé physiquement sur un FPGA Artix 7 de Xilinx, avec VTPRs. L'outil ISE rapporte alors une fréquence $f_{Clk_{VTPR}}$ maximum de 170.56 MHz. La colonne F_{reelle} du tableau 5.2 indique la fréquence réelle de fonctionnement des applications sur cette instance d'overlay lorsque la fréquence de l'horloge des VTPRs est au maximum permis, d'après la formule $f_{reelle} = \frac{f_{Clk_{VTPR}}}{N_{VTPR}}$.

Circuit applicatif	Synthèse sur Overlay			$\frac{f_{reelle}}{f_{pessimiste}}$	Synthèse sur FPGA (netlist)				Synthèse sur FPGA (RTL)			
	BLEs	Délai N_{VTPR}	f_{reelle} (MHz)		LUTs FFs	f_{max} (MHz)	Freq ratio	Area ratio	LUTs FFs	f_{max} (MHz)	Freq ratio	Area ratio
cordic	611	59	2.89	22.6	598	167.8	58.04	77.65	264	207.4	71.75	175.90
cmult	635	132	1.29	10.1	853	35.9	27.78	56.58	21	135.1	104.59	2298.21
Pmult	412	21	8.12	63.4	259	321.8	39.62	120.90	145	298.4	36.74	215.96
cdivmod	579	166	1.02	8.1	913	74.5	72.53	48.20	730	22.2	21.65	60.28
filtre RII	443	40	4.26	33.3	281	225.9	52.98	119.80	137	174.4	40.90	245.76
Moyenne	536	83.6	3.52	27.5	581	165.2	50.19	84.63	260	167.5	55.13	599.22

TABLE 5.2 – Résultats de synthèse de cinq applications sur overlay, sur FPGA à partir de netlists, et sur FPGA à partir des sources RTL.

5.6.1 Avantage des VTPRs par rapport à l'utilisation d'une fréquence pessimiste

La fréquence réelle des benchmarks s'étend de 1 à 8 MHz, ce qui est peu pour de petites applications. Cependant, si les VTPRs n'était pas présents dans l'overlay, et que pour assurer le fonctionnement des applications la fréquence pessimiste (cf 5.4.3) rapportée lors de la synthèse physique de l'overlay (et qui prend en compte le plus long chemin combinatoire possible dans l'overlay) était utilisée comme fréquence applicative, ces applications auraient toutes pour fréquence maximum 0.128 MHz. Cette fréquence a été obtenue en synthétisant le même overlay sur le même FPGA, mais sans les VTPRs, et en se basant sur la fréquence rapportée par l'outil de synthèse physique (ISE).

La cinquième colonne du tableau 5.2 présente le ratio entre la fréquence réelle de chaque application sur la fréquence pessimiste. Ainsi, l'utilisation de VTPRs permet de garantir le fonctionnement des applications à des fréquences $10\times$ plus rapide pour des applications combinatoires, $33\times$ pour des applications séquentielles, et jusqu'à $63\times$ plus rapide pour des applications pipelinées, que si les VTPRs n'étaient pas utilisés et que la fréquence pessimiste était utilisée [51].

5.6.2 surcoût en performances

Dans un troisième temps, pour comparer à une implémentation native, les cinq benchmarks ont été synthétisés directement sur le FPGA, c'est-à-dire sans l'overlay. Cependant, l'architecture de l'overlay utilisée précédemment ne comporte comme ressources logiques reconfigurables que des LUTs et des registres applicatifs, et ne présente pas de macros blocks tels que des blocks DSP ou encore de chaînes de retenue accélérant les additionneurs. Pour comparer ce qui est comparable, c'est-à-dire les LUTs et flip-flops physiques et virtuelles, les benchmarks ont d'abord étés pré synthétisés en suivant le début du flot virtuel, c'est-à-dire la synthèse RTL et la minimisation logique. Les netlists BLIF ont ensuite été traduites en VHDL pour finalement être synthétisées via l'outil de synthèse constructeur (ISE). Le VHDL issu du BLIF ne présente pas de symboles tels que des additions ou des multiplications, et n'est ainsi synthétisé que sur les LUTs et flip-flops du FPGA, sans faire usage de macros blocks ni de chaînes de retenue. La colonne *Synthèse sur FPGA (netlist)* du tableau 5.2 présente le résultat de ses synthèses.

La colonne *LUTs-FFs* indique le nombre de paires LUT-flip-flop occupées par les benchmarks. Une paire LUT-flip-flop est une LUT suivie d'une flip-flop, et est l'équivalent de ce que l'on appelle un BLE dans le cas de notre overlay. Une paire LUT-flip-flop est considérée occupée lorsqu'au moins sa LUT ou sa flip-flop est utilisée. L'occupation des benchmarks en ressources physiques est du même ordre de grandeur que l'occupation en ressources virtuelles (colonne *BLEs*). Cependant, il faut noter que les LUTs virtuelles de l'overlay sont des LUT-4 alors que les LUTs physiques du FPGA sont des LUT-6 fracturables.

La colonne f_{max} indique la fréquence maximale de fonctionnement des benchmarks sur le FPGA, et la colonne *Freq ratio* présente le rapport entre cette fréquence maximale native et la fréquence de fonctionnement maximale réelle sur l'overlay. Le surcoût de l'overlay en performances est important et s'étend de $28\times$ à $73\times$.

5.6.3 surcoût en ressources

La huitième colonne (*Area ratio*) du tableau 5.2 montre le surcoût en ressources FPGA apporté par l'overlay. Ce surcoût est calculé comme le rapport de l'occupation en ressources physiques des benchmarks synthétisés sur l'overlay par l'occupation en ressources physiques des benchmarks synthétisés directement sur le FPGA. L'occupation en ressources physiques des benchmarks synthétisés sur l'overlay est calculée comme les ressources FPGA occupées par l'overlay multiplié par son taux d'occupation par les benchmarks. L'overlay synthétisé comporte 728 BLEs et occupe 55328 paires LUT-flip-flops de son hôte, ce qui donne un rapport de 76 paires LUT-flip-flops physique par BLE virtuel. Il suffit alors de multiplier le nombre de BLEs occupées par un benchmarks pour estimer son occupation en ressources physiques.

Tout comme le surcoût en fréquence, le surcoût en ressource est important, et s'étend de $49\times$ à $121\times$ pour les benchmarks utilisés.

5.6.4 Comparaison avec une implémentation native depuis les sources

La troisième partie (*Synthèse sur FPGA (RTL)*) du tableau 5.2 présente les même résultats que pour la partie *Synthèse sur FPGA (netlist)*, avec la différence que les benchmarks ne sont pas pré synthétisés via le début du flot virtuel, mais sont synthétisés sur le FPGA entièrement via l'outil de synthèse constructeur (ISE) depuis les sources RTL des benchmarks. L'outil de synthèse constructeur est alors capable d'instancier les spécificités du FPGA cible qui n'ont pas d'équivalent dans le plan de calcul de l'overlay, tels que des blocs DSP (pour le benchmark cmult) et les chaînes de retenues.

Le surcoût en fréquence est du même ordre de grandeur que les synthèses natives de netlists pré-synthétisées, et s'étend de $22\times$ à $105\times$. En revanche, le surcoût en ressources est plus élevé, et s'étend de $60\times$ à $2300\times$. Le surcoût de $2300\times$ est dû à l'inférence d'un block DSP48E1 pour le benchmark *cmult*, qui permet de ne pas implémenter le multiplieur via des LUTs et flip-flops.

On peut noter que pour les trois derniers benchmarks, la synthèse sur FPGA depuis la netlist issue de la synthèse RTL et la minimisation logique via Odin II et ABC produit des circuits plus rapide que pour la synthèse sur FPGA directement depuis les sources RTL des benchmarks et entièrement via le flot construc-

teur. Ceci peut s'expliquer par le fait que la synthèse logique n'est pas réalisée par le même outil, et que ces outils n'ont pas les même options de synthèse. Aussi, pour ces benchmarks, les circuits pré-synthétisés utilisent presque deux fois plus de ressources que pour l'implémentation depuis les sources RTL. Ces ressources additionnelles peuvent expliquer les fréquences de fonctionnement plus élevées.

5.7 Résumé

Dans ce chapitre ont été présentés les différentes étapes constituant le flot de synthèse virtuelle ciblant des overlays grain fin, et les contraintes à respecter pour assembler un tel flot. Une fois l'overlay synthétisé sur FPGA via les outils de synthèse constructeur, l'outillage constructeur n'a plus besoin d'être utilisé. Ensuite, la synthèse d'applications sur l'overlay passe uniquement par le flot de synthèse virtuelle, qui est indépendant de l'outillage constructeur.

Le flot réalisé pour ces travaux a été présenté, il permet la synthèse d'application sur overlay, et produit un binaire de configuration ciblant l'overlay ciblé. Ce flot supporte la variabilité de l'overlay ciblé et s'adapte à ses paramètres architecturaux. Une méthode et des outils permettant de vérifier le flot et de déboguer les circuits applicatifs ont été présentés, le débogage permet d'accompagner le concepteur applicatif à différents stades de son développement. Les spécificités de l'analyse de timing sur overlay ont été soulignées, différentes méthodes ont été présentées et leur possibilité de mise en œuvre analysées. La méthode que nous avons retenue – l'utilisation des VTPRs pour donner une vue logique de la propagation des signaux applicatifs combinatoires – a été expliquée, et l'avantage qu'elle offre a été évalué. Finalement, le flot de synthèse réalisé a été mis en œuvre pour évaluer de surcoût en performance et en surface de l'overlay par rapport à une implémentation native des circuits applicatifs sur le FPGA.

Les overlays (grain fin) induisent un surcoût important en ressources et en performances. Ce surcoût est similaire au surcoût d'une implémentation FPGA par rapport à une implémentation ASIC [99]. Le surcoût des FPGAs ne fait pourtant pas obstacle à leur utilisation, car ils offrent en contrepartie la capacité de reprogrammation et permettent le prototypage rapide de circuits matériels. Il en va de même pour les overlays, qui, malgré le surcoût qu'ils induisent, apportent un niveau d'abstraction par rapport au support d'exécution physique. Ils offrent ainsi de nouvelles possibilités qui ne sont pas disponibles via l'utilisation native d'un FPGA. Notamment, ils permettent de rendre homogène un ensemble de FPGAs hétérogène, et le même bitstream applicatif peut être exécuté de manière transparente sur différents FPGAs hôte, sans nécessiter de ré-effectuer la synthèse applicative. L'outillage de synthèse applicative est lui aussi indépendant du FPGA hôte et des outils de synthèse qui le ciblent. Plus encore, la capacité d'extraire et de restaurer l'état d'un circuit applicatif en cours d'exécution permet de partager l'overlay dans le temps entre différents circuits applicatifs et de migrer ceux-ci d'un hôte physique à un autre. Dans le chapitre suivant, ces possibilités sont exploitées

pour la mise en œuvre des overlays dans le cadre du Cloud.

Chapitre 6

Exploitation des overlays dans un cadre Cloud

Dans les chapitres précédents nous avons vu ce qu'était un overlay, comment le concevoir, le générer et le synthétiser sur un FPGA. Nous avons ensuite vu comment l'intégrer dans un système en tant que "nœud overlay" d'un réseau informatique classique, effaçant ainsi les spécificités matérielles de la plateforme physique qui l'héberge. Finalement, la synthèse applicative sur overlay a été présentée. Cette section traite de l'utilisation des overlays dans le cadre du Cloud. En effet, l'abstraction du support physique offerte par les overlays, leur capacité à partager leurs ressources dans le temps et de migrer un circuit applicatif en cours d'exécution d'un overlay à un autre répondent à des problématiques du Cloud, qui consistent à gérer un ensemble hétérogène de ressources de manière optimale tout en respectant certaines contraintes.

6.1 Le cadre Cloud

Selon la définition du National Institute of Standards and Technology (NIST), le Cloud computing est l'accès via un réseau de télécommunications, à la demande et en libre-service, à des ressources informatiques partagées configurables [100]. Ces ressources informatiques sont physiquement situées dans des fermes de serveurs appelées datacenters. Le fournisseur – l'administrateur du datacenter – assure le bon fonctionnement des infrastructures de son datacenter, et les clients louent les ressources. Le SLA (Service Level Agreement) [101] est un accord négocié entre le fournisseur et le client, qui formalise la qualité de service attendu par le client, comme le niveau de disponibilité ou la performance des ressources louées. Si le fournisseur ne parvient pas à fournir la qualité de service négociée dans le SLA, il doit payer des pénalités au client. Le fournisseur n'a pas à savoir ce que font les applications des clients, et les clients n'ont pas à connaître les détails de l'infrastructure du datacenter.

Les ressources de calcul les plus abondantes dans un datacenter sont des serveurs, c'est-à-dire des ordinateurs composés de processeurs, de mémoires RAM et de mémoires de stockage. Ces serveurs sont généralistes et permettent l'exécution de tâches de différentes natures. Cependant, de plus en plus de ressources spécialisées apparaissent dans les datacenters, telles que les GPUs, permettant aux clients du Cloud d'utiliser les ressources de calcul de ces circuits pour réaliser leurs traitements plus rapidement que sur les processeurs généralistes composants les serveurs [102]. Cependant, les GPUs ne sont pas adaptés à tous les types de tâches, car leur architecture est spécialisée pour le calcul massivement parallèle en flux de données. En effet, les GPUs tirent parti du parallélisme au niveau des données (SIMD) et des fils d'exécution [4]. Du fait du modèle d'exécution SPMD (pour Single Program, Multiple Data) des GPUs, le bénéfice de ces derniers s'atténue lorsque les fils d'exécution exécutés par une même unité prennent des branchements différents [103]. Les FPGAs commencent eux aussi à apparaitre dans les datacenters [104]. Microsoft a récemment présenté une étude des bénéfices de l'utilisation des FPGAs dans des datacenters pour accélérer son moteur de recherche Bing [105]. Aussi, Intel intègre un FPGA avec un processeur XEONs sur une seule puce, et son récent achat d'Altera souligne l'importance des FPGAs dans les datacenter comme ressource de calcul [13]. En effet, la flexibilité des FPGAs leur permet de réaliser un circuit matériel adapté et optimisé pour chaque application, et le nombre important de mémoires qu'ils intègrent dans leur matrice reconfigurable permet d'éviter les faiblesses du modèle d'exécution Von Neumann des processeurs classiques et des GPUs [106].

Cependant, l'intégration de FPGAs dans un datacenter comme ressources accessibles aux clients comme accélérateurs reconfigurables n'est pas triviale. En effet, pour être intéressante, l'intégration de ressources de calcul dans un datacenter doit permettre à la fois à l'administrateur du datacenter de gérer simultanément différents clients avec différentes demandes et différentes priorités tout en respectant leurs SLAs respectifs, mais aussi d'optimiser la gestion de l'infrastructure du datacenter pour en réduire le coût d'exploitation. Dans le cas des serveurs classiques, la solution qui a été adoptée est d'isoler les applications clientes des serveurs physiques par l'utilisation de machines virtuelles (VM). La virtualisation des serveurs permet à l'administrateur de contrôler dynamiquement l'exécution des charges de travail des clients dans les ressources du datacenter. Notamment, l'utilisation de machines virtuelles permet :

— De *partager* un serveur physique entre plusieurs clients, car un serveur physique peut héberger plusieurs machines virtuelles. L'administrateur a le contrôle de la quantité de ressources du serveur allouées à chaque VM ainsi que le contrôle sur le temps d'exécution de chaque VM.

— De *migrer* une VM d'un serveur physique à un autre. La migration peut servir à réaliser de l'*équilibrage de charge* dans l'infrastructure du datacenter, c'est-à-dire de répartir équitablement la charge de travail sur les différentes ressources de l'infrastructure pour maximiser les performances ; ou au contraire de *consolider* des serveurs, c'est-à-dire de concentrer la charge de travail dans un nombre minimum de serveurs pour permettre d'éteindre les ressources alors libérées et ainsi de diminuer la consommation énergé-

tique de l'infrastructure.
— De prendre des sauvegardes régulières des VMs afin de pouvoir les restaurer en cas de panne d'une ressource, et ainsi augmenter la résilience aux pannes.

6.2 Exigences

L'utilisation native des FPGAs se prête mal à une utilisation dans le Cloud. En effet, les FPGAs possèdent quelques faiblesses pour une telle utilisation : ils ne sont pas multi-utilisateur, ont des temps de reconfiguration non négligeables (de l'ordre de la centaine de millisecondes), et ne sont pas conçus pour permettre aisément d'extraire et de restaurer l'état d'exécution des circuits applicatifs qu'ils implémentent. Or, la capacité d'extraire et de restaurer un état d'exécution (c'est-à-dire la valeur de tous les éléments séquentiels du circuit applicatif) est indispensable pour pouvoir effectuer des commutations de contextes préemptives. La figure 6.1 présente le schéma classique d'ordonnancement des processus sur un processeur par un système d'exploitation. Dans ce schéma, l'utilisation du processeur est optimisée par le fait que l'ordonnanceur libère le processeur de tout processus qui tombe en attente d'un évènement pour laisser la place à un processus prêt. Aussi, l'ordonnanceur permet le partage du processeur suivant des priorités entre les différents processus en interrompant le processus en cours d'exécution (préemption) au terme d'un temps déterminé pour permettre à un autre de s'exécuter. L'optimisation de l'utilisation du processeur et l'équité de son partage entre les processus reposent sur la capacité du système d'exploitation d'interrompre et de reprendre l'exécution des processus sur le processeur.

La reconfiguration partielle dynamique (DPR) est une fonctionnalité offerte par certains FPGAs du commerce, elle permet de reconfigurer une région du FPGA sans modifier ni interrompre le fonctionnement des autres régions. La DPR peut donc être utilisée pour réaliser l'ordonnancement d'applications sur FPGA, et ainsi permettre un partage temporel de celui-ci. Cependant, sans accès à l'état d'exécution des applications, il n'est possible de restaurer une application que dans son état initial, et non son état précèdent sa suspension. L'ordonnanceur doit donc attendre la complétion de l'application en cours avant de pouvoir réaliser le prochain changement de contexte. Ainsi, il faut que la durée d'exécution des applications soient limités dans le temps pour permettre à plusieurs applications d'être exécutées sur le même FPGA dans un laps de temps donné. Aussi, lorsqu'une application est en attente d'un évènement extérieur comme l'arrivée de données d'entrées, elle utilise inutilement le FPGA, mais son exécution ne peut pas être suspendue au profit d'une autre application à moins de perdre son état actuel et de devoir reprendre son exécution depuis son état initial. Ainsi, sans accès à l'état d'exécution des applications, il est possible de partager temporellement un FPGA entre plusieurs applications mais il n'est pas possible de mettre en place le schéma d'ordonnancement présenté figure 6.1 ; l'utilisation du FPGA n'est pas optimale et il n'est pas possible d'assurer un partage équitable du FPGA entre les applications.

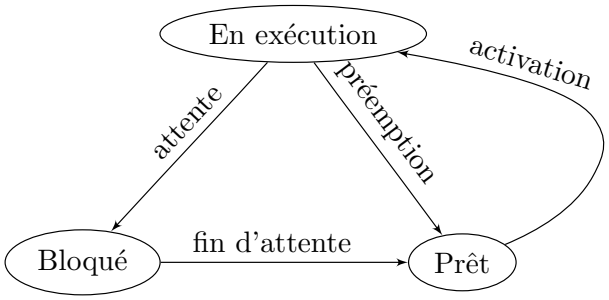


FIGURE 6.1 – Diagramme de transition d'états d'un processus

De plus, les bitstreams ne sont pas portables d'un modèle de FPGA à un autre, ni même entre les FPGAs d'un même vendeur et les variantes d'un même modèle. Cela signifie qu'un bitstream synthétisé pour un FPGA donné ne peut pas configurer un FPGA d'un autre modèle. Dans le meilleur des cas, il suffit de synthétiser le code source de l'application pour obtenir un bitstream compatible avec la nouvelle cible ; dans le pire des cas il peut être nécessaire de réécrire en partie l'application. Or, les ressources de l'infrastructure d'un datacenter sont hétérogènes. Par exemple, la durée de vie moyenne d'un serveur dans un datacenter est de trois ans [107]. Mais les équipements ne sont jamais remplacés simultanément, et afin de moderniser l'infrastructure et de suivre les évolutions technologiques et des ventes de matériels, les derniers équipements du commerce sont achetés et intégrés au datacenter. Cela résulte en différentes générations d'équipements étant utilisées à un instant donné dans la même infrastructure. Dans le cas des serveurs, cette hétérogénéité n'est pas un problème, car bien que les processeurs de différentes génération et de différents vendeurs aient des microarchitectures différentes, ils partagent le même jeu d'instructions (le x86-64), ce qui assure la portabilité du logiciel et des machines virtuelles. Cependant, les FPGAs ne bénéficient pas d'une telle inter-compatibilité. Ainsi, les FPGAs ne permettent pas nativement de mettre en place les mécanismes nécessaires pour la gestion dynamique de l'infrastructure d'un datacenter.

Ainsi, l'intégration de FPGAs dans l'infrastructure d'un datacenter comme ressource de calcul demande de répondre à plusieurs exigences. Dans un premier temps, les exigences posées par le cadre général du Cloud sont les suivantes :

Ea : *Optimiser l'exploitation des ressources*. En effet, le fournisseur cherche à diminuer le coût d'exploitation de son infrastructure.

Eb : *Répondre au SLAs des clients* lors de la gestion des ressources. C'est-à-dire offrir la qualité de service à laquelle les clients ont souscrit. Le fournisseur cherche à respecter les SLAs de ses clients pour éviter d'avoir à leur payer des pénalités.

Ec : *La gestion de l'infrastructure est entièrement à la charge du fournisseur*. Le client n'a pas à connaitre des détails de l'infrastructure, et la gestion efficace de l'infrastructure ne peut pas reposer sur des actions réalisées par les clients.

Ed : *Supporter un ensemble hétérogène de plateformes*, et sans contraintes sur les plateformes, pour supporter la mise à jour graduelle de l'infrastructure.

Ee : *Exploitation dynamique des ressources*, pour permettre d'adapter leur utilisation en fonction de la charge de travail globale qui varie dans le temps.

Ces exigences peuvent ensuite être raffinées :

E1 : *Multi-utilisateur* : une même ressource FPGA doit pouvoir être partagée entre plusieurs utilisateurs/applications.

- **E1.a** : *Partage spatial* : La ressource est partagée spatialement entre différentes applications.
- **E1.b** : *Partage temporel* : La ressource est partagée dans le temps entre différentes applications.

E2 : *Portabilité des binaires applicatifs* : un même binaire applicatif peut être exécuté sur les différentes ressources de l'infrastructure.

- **E2.a** : *Un bitstream par application* : le fournisseur n'a à manipuler qu'un unique binaire applicatif par application cliente.
- **E2.b** : *Une synthèse par application* : le client n'a à réaliser qu'une synthèse par application.

E3 : *Ordonnancement préemptif* : l'ordonnancement doit être préemptif pour permettre au fournisseur d'optimiser l'utilisation des ressources et gérer les priorités des applications en gérant le temps alloué à chacune.

- **E3.a** : *Extraction/chargement d'état d'exécution* : pour permettre la préemption, la ressource doit permettre d'extraire et de restaurer l'état d'exécution des applications.
- **E3.b** : *Extraction/chargement d'état d'exécution et configuration rapide* : pour minorer le surcout dû à l'ordonnancement, l'accès à l'état d'exécution et la configuration de la ressource doivent être rapides.

E4 : *Migration* : pour optimiser l'utilisation globale de l'infrastructure en permettant des mécanismes comme la consolidation [108] et l'équilibrage de charge [109].

- **E4.a** : *Migration à chaud* : l'application est reprise sur la plateforme d'arrivée sans avoir à être réinitialisée.
- **E4.b** : *État d'exécution indépendant de l'architecture cible* : la migration peut avoir lieu entre deux plateformes différentes.

6.3 Solutions existantes

Certains travaux répondent en partie à ces problèmes. Cette section présente différentes solutions existantes et les compare par rapport aux différentes exigences établies ci-dessus. Une synthèse est présentée dans le tableau 6.1.

Certains auteurs [57, 110, 55, 111] utilisent la reconfiguration dynamique partielle (DPR) pour rendre les FPGAs multi-utilisateurs. Le FPGA est divisé en plusieurs régions reconfigurables dynamiquement (c'est-à-dire qui sont reconfigurables sans modifier ni interrompre le fonctionnement du FPGA dans les autres

régions), et d'une partie statique. Les régions reconfigurables sont donc utilisables indépendamment les unes des autres par différents utilisateurs (E1.a), tandis que la partie statique est chargée de la reconfiguration de ces régions et de leur alimentation en données depuis l'extérieur (E1.b). Cependant, tous les FPGAs n'ont pas la fonctionnalité de reconfiguration dynamique partielle. De plus, cette solution seule ne permet pas de réaliser de l'ordonnancement préemptif (E3) ni de la migration transparente d'application (E4), et ne répond pas non plus au problème de non-portabilité des applications (E2). En particulier, comme souligné dans [55], un bitstream partiel ciblant une région reconfigurable ne peut pas configurer une autre région reconfigurable du même FPGA, il faut donc réaliser une synthèse par application, par FPGA, et en plus par région reconfigurable.

Dans ses travaux [112], Alban Bourge adresse l'extraction et la restauration du contexte d'exécution d'applications, ce qui permet la mise en place d'un ordonnancement préemptif des applications sur le FPGA (E3). Durant sa synthèse, l'application est analysée de façon à déterminer les points d'exécution – appelés points de contrôle – pour lesquels l'état d'exécution significatif est minimum (en termes de bits d'information) de manière à limiter le temps d'extraction/restauration de l'état et ainsi minimiser le coût d'un changement de contexte (E3.b). L'application est automatiquement instrumentée par un outil de synthèse HLS (en amont de la synthèse RTL) pour permettre l'extraction et la restauration rapide des éléments séquentiels significatifs [20]. En plus de permettre un ordonnancement efficace sur FPGA, cette solution permet la migration d'applications d'un FPGA à un autre (E4.a et E4.b). En effet, l'état d'exécution de l'application est indépendant de l'architecture de la cible physique, de même que le mécanisme d'extraction de l'état qui est fixé au niveau RTL (et donc avant spécialisation du flot de synthèse pour une cible FPGA particulière), donc l'état d'exécution n'est propre qu'à l'application et non à son implémentation sur un FPGA en particulier. En revanche, les bitstreams ne sont toujours pas portables d'un FPGA à un autre, et bien que la migration soit possible entre deux FPGAs différents, l'application doit être synthétisée pour chacun des deux FPGAs (E2).

Cependant, dans un cadre Cloud, le fournisseur n'a pas à connaître ni à manipuler les applications du client qui sont exécutées sur les ressources louées par celui-ci, et le client n'a pas à connaître les détails de l'infrastructure physique du fournisseur (Ec). Comme l'instrumentation de l'application se fait au niveau de sa synthèse, elle est donc à la charge du client, et non du fournisseur. Or, c'est cette instrumentation qui permet au fournisseur de gérer l'exécution des applications sur son infrastructure. Le fournisseur doit donc faire confiance aux applications fournies par les clients, et notamment que le temps maximum de sauvegarde/restauration de l'état d'exécution ainsi que le temps maximum entre deux points de contrôle correspondent bien aux contraintes du fournisseur. En effet, bien que l'ordonnancement soit à la charge de l'administrateur, la préemption ne peut avoir lieu qu'aux états d'exécution correspondant aux points de contrôle choisis lors de la synthèse de l'application. De plus, le client doit réaliser une synthèse de son application par modèle de FPGA utilisé par le fournisseur (E2.b). Cela l'oblige donc à connaître certains détails de l'infrastructure physique du fournisseur (Ec). Aussi, le fournisseur doit gérer plusieurs bitstreams pour une même application cliente

(E2.a).

Une alternative à l'instrumentation de l'application donnant accès à l'état d'exécution du circuit et permettant ainsi l'ordonnancement préemptif et la migration d'application (E3.a, E4.a) est la relecture de bitstream [16, 19]. Pour les FPGAs Xilinx, le bitstream relu (à l'aide d'une sonde JTAG ou d'un port interne ICAP) comprend les bits de configuration du plan de calcul du FPGA physique, mais aussi l'état des registres des CLBs. Cependant, cette solution ne permet pas une extraction et une restauration rapide de l'état (E3.b) du fait que les bits de configuration et d'état sont mélangés et ne sont pas accessibles séparément, tous les éléments séquentiels de l'architecture sont sauvés (pas seulement ceux qui sont pertinents pour une application particulière), et les bits de configuration sont sauvés et restaurés avec les bits d'état. De plus, la relecture du bitstream ne permet pas la migration de l'application entre deux FPGAs différents (E4.b). Il est possible d'extraire les éléments séquentiels du bitstream relu, mais cela demande un temps de traitement supplémentaire et rend la solution encore plus dépendante de la cible

Les overlays sont une solution qui répond aux problèmes posés par l'intégration de FPGAs dans une infrastructure Cloud comme ressources accessibles aux clients en tant qu'accélérateurs reconfigurables. Comme la virtualisation des serveurs classiques en machines virtuelles, les overlays abstraient les applications de leurs supports physiques d'exécution (les FPGAs de l'infrastructure), et permettent à l'administrateur de contrôler dynamiquement leurs exécutions. Notamment, en étant conçu pour permettre l'extraction et la restauration de l'état d'exécution des applications (E3.a) via l'intégration d'un plan de snaptshot, les overlays reprennent les avantages de [112] ; et en apportant leur propre mécanisme de reconfiguration, les overlays permettent le partage temporel des ressources FPGA (E1.b) avec ordonnancement préemptif (E3) d'applications même sur un FPGA ne proposant pas la DPR. Ce partage temporel n'est pas incompatible avec un partage spatial (E1.a) : en effet, un FPGA peut très bien accueillir plusieurs overlays, et un overlay lui-même peut aussi accueillir plusieurs applications indépendantes. Le changement de contexte est transparent pour l'application qui voie les ressources de l'overlay comme lui étant entièrement dédiées, et n'a aucune vision des autres applications avec qui elle partage l'overlay. L'administrateur peut gérer lui-même et dynamiquement le temps d'exécution qu'il alloue à chaque application et ainsi gérer leurs priorités. L'accès à l'état d'exécution permet aussi la migration d'applications d'un overlay à un autre (E4.a), donnant ainsi à l'administrateur le moyen de gérer dynamiquement la répartition des charges de travail dans son infrastructure, et ce toujours de façon transparente pour les applications. Aussi, l'état d'exécution peut être sauvegardé arbitrairement par l'administrateur pour permettre de reprendre l'exécution d'une application après une panne de son support d'exécution, et ainsi rendre l'infrastructure résiliente aux pannes.

Par rapport aux travaux [112], le changement de contexte est géré entièrement au niveau de l'overlay par l'administrateur et ne repose sur aucune opération à la charge du client et ne dépend d'aucune contrainte venant de l'application (Ec). Le changement de contexte peut être réalisé à tout moment car il ne dépend pas de points de contrôle préétablis. De plus, par nature, l'utilisation d'un overlay rend

les bitstreams virtuels applicatifs portables sur toutes les ressources FPGAs implémentant l'overlay ciblé (E2). Les clients n'ont donc à réaliser qu'une unique synthèse par application (E2.b), et l'administrateur n'a à manipuler qu'un unique bitstream virtuel par application cliente (E2.a). Aussi, la migration d'application peut se faire entre deux FPGAs différent tant qu'ils implémentent le même overlay (E4.b).

Cependant, du fait que l'accès à l'état d'exécution est pris en charge par l'overlay et non par l'application, l'extraction et la restauration de l'état d'exécution sont moins rapide que [112] ; en effet, tous les éléments séquentiels du plan de calcul sont sauvegardés de manière aveugle, qu'ils soient utilisés et significatifs ou non pour l'application. En revanche, par rapport à la solution de relecture du bitstream, l'état d'exécution étant accessible séparément de la configuration pour l'overlay, l'accès à l'état est plus rapide (E3.b). Dans le cas d'un changement de contexte, la nouvelle application exécutée est différente de la première, ou alors la même application est exécutée mais avec un état différent. Dans le deuxième cas, seul l'état doit être modifié, il n'y a donc pas d'avantage à ne pouvoir accéder qu'à l'état d'exécution ; mais dans le premier cas (le plus général), la configuration doit être changée en plus de l'état d'exécution, et il n'y a donc pas de pénalité par relecture du bitstream à ne pouvoir extraire et restaurer qu'ensemble la configuration et l'état. Cependant, comme il a été vu en 3.2.2, un overlay peut implémenter un mécanisme de pré-configuration qui, en permettant de pré-charger la configuration (étape longue) sans interrompre le fonctionnement du plan de calcul, minore le coût de configuration (par recouvrement) (E3.b). Pour la mise en place d'ordonnancement d'applications, les overlays ont donc un avantage par rapport à [relecture de bitstream].

Le tableau 6.1 récapitule et compare les fonctionnalités offertes par ces différentes solutions. Le partage spatial des ressources entre plusieurs applications est assuré par l'utilisation de la DPR simple, mais aussi par les méthodes de relecture de bitstream et [112] qui utilisent aussi les mécanismes de DPR des FPGAs. Ces solutions ne peuvent être mises en place que sur des FPGAs qui proposent la fonctionnalité de reconfiguration partielle dynamique. Les overlays quant à eux ne reposent pas sur la DPR du FPGA hôte car ils implémentent leur propre mécanisme de configuration, et peuvent donc être implémenté sur un ensemble plus large de FPGA du commerce. Le partage spatial du FPGA est réalisé en implémentant plusieurs overlays sur le même FPGA. Le partage temporel du FPGA est possible pour toutes les solutions donnant accès à l'état d'exécution, mais aussi pour la DPR simple qui permet d'exécuter les applications les unes à la suite des autres, chacune étant exécutée jusqu'à complétion. L'extraction/restauration de l'état d'exécution est la plus rapide pour [112] car seuls les éléments pertinents pour l'état de l'application sont accédés, contrairement à la solution overlay qui accède de manière aveugle à tous les éléments séquentiels de son plan de calcul via l'extraction du plan de snapshot. L'accès est encore plus long par relecture de bitstream car en plus de tous les éléments séquentiels, les éléments de configuration sont aussi accédés. Dans [112], la préemption d'une application ne peut avoir lieu que si cette application est dans un état qui correspond à un point de contrôle établi lors de la synthèse, alors qu'avec la relecture de bitstream et les overlays, la préemption

peut avoir lieu à tout moment. Pour [112], l'instrumentation des applications pour qu'elles puissent être gérées correctement par le fournisseur est à la charge du client. Aussi, toutes les solutions reposant sur la DPR (DPR simple, relecture de bitstream et [112]) nécessitent que le client synthétise son application contre le design statique réalisé par le fournisseur. La migration d'une application entre deux FPGAs identiques est possible avec la relecture de bitstream, car le bitstream qui est transféré au second FPGA (qui contient configuration et état) peut être utilisé par ce second FPGA puisqu'il est du même modèle que le premier. En revanche, la relecture de bitstream ne permet pas de migrer une application entre deux modèles de FPGAs différents, contrairement aux solutions [112] et des overlays. En effet, pour [112], l'accès à l'état d'exécution est remonté au niveau applicatif et est indépendant du FPGA hôte, tandis que pour l'overlay l'accès à l'état est fourni par son architecture, qui est portable sur différents FPGAs. Comme les FPGAs disponibles dans le commerce ne proposent pas de fonctionnalité de pré-configuration, les overlays, en implémentant leur propre mécanisme de configuration, sont la seule solution qui permette de minorer le temps de configuration. Finalement, les overlays sont la seule solution qui permette la portabilité des bitstreams applicatifs sur différents FPGAs.

Ainsi, la solution des overlays permet l'intégration des FPGAs dans l'infrastructure d'un datacenter et leur utilisation depuis le Cloud comme accélérateurs reconfigurables accessibles aux clients tout en répondant aux différentes exigences établies au début de cette section. En effet, les overlays permettent un usage multi-utilisateur (E1) des FPGA en permettant le partage spatial (E1.a) et temporel (E1.b) des ressources FPGA, et laissent au fournisseur le choix de la fragmentation spatiale et temporelle de ses ressources pour les utiliser au mieux (Ea). Par exemple, une fragmentation élevée d'un FPGA en plusieurs overlays permet de supporter simultanément plus d'applications/de clients par FPGA mais la taille des applications que peut supporter chaque overlay est plus limitée. Aussi, à une échelle de temps donnée, diminuer la tranche de temps d'exécution allouée à chaque application avant sa préemption permet de gérer plus finement le partage de l'overlay mais augmente le coût d'ordonnancement. Les overlays sont la seule solution qui supporte un ensemble hétérogène de plateformes FPGA (Ed) tout en assurant la portabilité des binaires applicatifs (E2). Ainsi, malgré la mise à jour graduelle des ressources de l'infrastructure, le fournisseur n'a à gérer qu'un binaire par application cliente (E2.a) et les clients n'ont à effectuer qu'une synthèse par application (E2.b). Les overlays permettent l'ordonnancement préemptif des applications (E3) en donnant accès à l'état d'exécution des applications (E3.a), ce qui permet d'augmenter leur taux d'utilisation et de gérer finement et dynamiquement (Ee) les différentes priorités de chaque application pour répondre au mieux aux SLAs des clients (Eb). Bien que l'accès à l'état d'exécution est plus lent pour les overlays que pour [112], les overlays peuvent implémenter le mécanisme de chargement et d'extraction des registres applicatifs via un double buffer (cf 5.2.3) qui, comme pour la pré-configuration, permet par recouvrement de minorer le coût de l'accès à l'état des registres lors d'un changement de contexte. De plus, l'implémentation des mémoires virtuelles utilisées avec les overlays (cf 3.2.5) permet aussi la mise en place d'un système de double buffers rendant possible de commuter le contenu des mé-

Solution / Exigence	Partage spatial (E1.a)	Partage temporel (E1.b)	Portabilité des binaires applicatifs (E2)	Préemption (E3)	Accès à l'état d'exécution (E3.a)	Reconfiguration (E3.b)	Migration (E4)	Migration à chaud (E4.a)	Migration entre FPGAs de modèles différents (E4.b)	Le client n'a pas à connaitre les détails de l'infrastrutcure, et la gestion de celle-ci ne dépend pas du client (Ec)	Ne nécessite pas DPR hôte (Ed)	Surcoût (surface et performances)
FPGA natif	non	non	non	non	non	très lente	non	non	non	oui	oui	nul
DPR simple	oui	oui	non	non	non	lente	non	non	non	non, le client à besoin du design statique	non	nul
Relecture de bitstream	oui	oui	non	n'importe quand	lent	lente	oui	oui	non	non, le client à besoin du design statique	non	nul
Bourge [112]	oui	oui	non	aux points de contrôles	très rapide	lente	oui	oui	oui	non, le client à besoin du design statique et instrumente ses applications	non	très faible
Overlay sans snapshot	oui	oui	oui	non	non	rapide	non	non	non	oui	oui	important
Overlay avec snapshot	oui	oui	oui	n'importe quand	rapide	rapide	oui	oui	oui	oui	oui	important

TABLE 6.1 – Les différentes solutions pour l'intégration de FPGAs dans un datacenter par rapport aux exigences du Cloud.

moires virtuelles sans avoir à déplacer des données dans les mémoires physiques les implémentant, et donc minore l'impact des mémoires dans le coût d'un changement de contexte. Les overlays offrent donc un changement de contexte rapide pour minorer le surcout dû à l'ordonnancement (E3.b). Aussi, les overlays permettent la migration des applications (E4) d'une plateforme à une autre, qu'elles soient différentes ou non sans nécessiter de re-synthétiser l'application (E4.b), tout en conservant l'état d'exécution lors de la migration (E4.b). Finalement, les overlays laissent les mécanismes de gestion de l'infrastructure entièrement à la charge du fournisseur (Ec) : les clients n'ont pas à connaitre les détails de l'infrastructure (comme les différents modèles de FPGA qu'elle contient ou la définition de zones DPR à disposition), et le client n'est pas non plus chargé d'instrumenter son application pour permettre sa préemption.

Pour résumer, les overlays permettent une gestion dynamique et transparente d'un ensemble hétérogène de ressources. Bien que les overlays grain fin aient un surcoût important en ce qui concerne les performance et la surface occupée par rapport à une utilisation native des FPGAs, ce surcoût peut être minoré en optimisant l'implémentation par rapport à l'architecture hôte [23], en grossissant le grain [39, 113], ou encore en spécialisant leurs architectures fonctionnelles par domaines applicatifs [24]. Ainsi, les overlays gros grain (avec mécanisme de snapshot) peuvent se présenter comme une solution intermédiaire entre les overlay grain fin présentés dans ces travaux et la solution d'Alban Bourge [112].

6.4 L'hyperviseur : contrôle local

Dans les chapitres précédents nous avons vu ce qu'était un overlay, comment le réaliser, comment compiler des applications dessus et comment l'intégrer dans un système. Nous venons de voir en quoi les overlays sont une solution attractive pour une intégration dans l'infrastructure d'un datacenter. Cette section présente l'exploitation d'un overlay dans un contexte Cloud.

Dans un datacenter, les ressources de l'infrastructure sont rassemblées en cluster, c'est-à-dire en grappes de ressources connectées entre elles par un réseau informatique, et contrôlées via ce réseau par un contrôleur global. Dans un cluster, les ressources sont appelées des nœuds. Les deux principales catégories de nœuds sont les nœuds de calcul, réalisant les traitements, et les nœuds de stockage, hébergeant les données. Comme il a été vu au début de ce chapitre, les nœuds de calcul peuvent être de différente nature, comme des serveurs généralistes, des GPUs ou des FPGAs. Les overlays doivent donc être intégrées à un datacenter en tant que nœuds de calcul d'un cluster, contrôlables par un contrôleur global.

Nous avons vu au chapitre 4 qu'un nœud de calcul pour overlay est une plateforme qui comporte une mémoire locale, une interface réseau, un overlay et son contrôleur local. Ce contrôleur local est un logiciel exécuté dans le nœud de calcul, c'est-à-dire sur la plateforme matérielle de déploiement de l'overlay. Nous appelons ce contrôleur l'*hyperviseur* de l'overlay, par analogie à un hyperviseur (ou moniteur de machine virtuelle), qui est la brique logicielle utilisée pour créer et exécuter des machines virtuelles sur un ordinateur. Sur ordinateur, dans le cadre de la virtualisation en machines virtuelles, l'hyperviseur alloue et contrôle les ressources physiques de l'ordinateur hôte pour les émuler en ressources abstraites capables d'exécuter différents systèmes d'exploitation invités indépendants du système hôte : les machines virtuelles. Les différentes instances de machines virtuelles s'exécutent indépendamment les unes des autres sans interférer entre elles. Chaque instance de machine virtuelle voit ses ressources comme lui étant entièrement dédiées et n'a pas connaissance ni de l'exécution ni de l'existence des autres instances.

Dans le cadre de ces travaux, l'émulation du FPGA hôte en FPGA abstrait est réalisée matériellement par l'overlay, tandis que l'hyperviseur gère de manière lo-

gicielle le partage des ressources du nœud de calcul – soit l'overlay, la mémoire locale et la communication avec l'extérieur – pour permettre l'exécution de différentes applications indépendantes sur l'overlay. La plateforme matérielle d'intégration et d'exploitation de l'overlay – vue au chapitre 4 et qui se présente au contrôleur global comme un nœud de calcul – est composée de l'overlay, de ses contrôleurs matériels et d'un processeur qui exécute l'hyperviseur. La partie bas niveau de l'hyperviseur a été présentée au chapitre 4, c'est-à-dire la partie comprenant les appels systèmes permettant la gestion des contrôleurs matériels de l'overlay. Ici nous allons nous intéresser à la partie haut niveau de l'hyperviseur, responsable de la gestion des applications.

L'hyperviseur gère localement (au niveau du nœud) l'exécution des applications dont il a la charge, c'est-à-dire qu'il est responsable de leur ordonnancement sur l'overlay suivant leurs priorités et la disponibilité de leurs entrées/sorties. Il reçoit ses ordres du contrôleur global, tels que l'ajout ou le retrait d'une application à sa liste d'applications à exécuter, ou une demande de rapport d'état (taux d'utilisation, consommation électrique) permettant au contrôleur global d'optimiser sa gestion des nœuds.

6.4.1 Applications virtuelles

Les applications sont donc transférées du contrôleur global vers le nœud de calcul et sont conservées sur celui-ci jusqu'à complétion ou migration sur un autre nœud. Pour être manipulées par le contrôleur global et l'hyperviseur des nœuds de calcul, ces applications doivent donc être sérialisées. Nous appelons *application virtuelle* la structure comprenant les différentes informations constituant une application. L'application virtuelle peut être vue comme l'équivalent d'un bloc de contrôle de processus (dans un système d'exploitation), qui en plus de contenir les informations relatives à l'identifiant, l'état et le contrôle du processus, contiendrait aussi le programme (les instructions) exécuté par le processus ainsi que le contenu de sa pile d'exécution (état). En effet, l'application virtuelle contient des métadonnées, des données statiques et des données dynamiques. Les données statiques sont composées du bitstream virtuel permettant de configurer l'overlay pour implémenter l'application. Les informations sur le plan de calcul de l'overlay ciblé par le bitstream virtuel y sont aussi incluses de façon à ce que l'hyperviseur puisse rejeter une application virtuelle dont le bitstream virtuel n'est pas compatible avec l'overlay qui lui est rattaché. Aussi, les performances temporelles de l'application sont incluses dans l'application virtuelle, il s'agit de la fréquence f_{virt} (cf 5.4), indiquant de combien l'horloge physique du FPGA doit être divisée pour obtenir l'horloge applicative contrôlant les registres applicatifs du plan de calcul de l'overlay. Les données dynamiques de l'application virtuelle comprennent l'état d'exécution de l'application. Celui-ci est composé de la valeur de tous les registres applicatifs du plan de calcul de l'overlay, et du contenu d'éventuelles mémoires virtuelles. Pour optimiser la gestion des mémoires virtuelles par l'hyperviseur, les données statiques de l'application virtuelle indiquent aussi si l'application écrit ou non dans les mémoires virtuelles. En effet, si l'application modifie le contenu de la mémoire

virtuelle, alors son contenu doit être sauvegardé dans l'état d'exécution de l'application virtuelle lors d'un changement de contexte ; mais si la mémoire n'est utilisée par l'application qu'en lecture seulement, cette sauvegarde est inutile. Finalement, les métadonnées de l'application virtuelle contiennent des informations telles que le client à qui appartient l'application, sa priorité, et aussi des informations pouvant aider à sa gestion telles que le nombre de cycles d'horloge applicative dont a bénéficié l'application virtuelle ou encore le nombre de changements de contextes qu'il a subi.

La création d'une application virtuelle par le client est réalisée après extraction du bitstream virtuel et analyse de timings (cf chapitre 5). L'état d'exécution initial (le snapshot des registres applicatifs du plan de calcul de l'overlay) est construit à partir de la valeur initiale des signaux dans le code source de l'application. Par exemple, dans le cas d'une application écrite en VHDL et où apparaitrait "`signal counter : std_logic_vector(3 downto 0) := "0110";`", le "`"0110"`" participerait à la création de l'état d'exécution initial pour initialiser les registres applicatifs de l'overlay implémentant le signal "`counter`" de l'application. Si la mémoire virtuelle est utilisée, son contenu est construit à partir du code RTL de l'application ou à partir d'un fichier spécifique au contenu de la mémoire virtuelle, et est intégré dans l'application virtuelle. Lorsque le client envoie son application virtuelle dans le Cloud, l'administrateur du datacenter complète les métadonnées de l'application virtuelle par l'identifiant du client et une priorité correspondant à la qualité de service du SLA du client, et initialise le nombre de cycles d'horloge exécutés et le nombre de changements de contextes subit par l'application virtuelle à zéro ; puis envoie l'application virtuelle à un nœud de calcul selon sa politique de gestion de son infrastructure.

Dans ces travaux, un prototype d'hyperviseur a été réalisé, implémentant les mécanismes de base nécessaires à l'ordonnancement d'application virtuelles sur un overlay. Pour simplifier son développement et permettre la réutilisation de certaines parties indépendamment des autres, il est divisé en trois composants : une couche d'appels systèmes, un ordonnanceur et un serveur. La couche d'appels systèmes a été vue en 4.3, elle permet d'accéder aux contrôleurs matériels de l'overlay. L'ordonnanceur gère l'ordonnancement des applications, il repose sur la couche d'appels systèmes pour orchestrer les changements de contextes sur l'overlay. Le serveur quant à lui écoute sur l'interface réseau du nœud et relaie les requêtes du contrôleur global à l'ordonnanceur. La sous-section suivante présente les opérations réalisées par l'ordonnanceur pour réaliser un changement de contexte sur l'overlay.

6.4.2 Changement de contexte sur l'overlay

Lors de son fonctionnement, l'hyperviseur orchestre différents composants : l'overlay, la mémoire locale, les applications sous formes d'application virtuelles stockés dans la mémoire locale du nœud, ainsi que les flux d'entrées/sorties des applications. Les flux d'entrées/sorties peuvent provenir/repartir de/vers l'Internet

(externe au datacenter), de nœuds de stockage ou d'autres nœuds de calcul (internes au datacenter). La figure 6.2 illustres les mouvements de données entre ces différents composants. Les flux d'entrées/sorties des applications sont tamponnées de façon à être disponibles lorsque leurs applications sont exécutées. Ces tampons sont nommés *données d'entrée* et *données de sortie* dans la figure 6.2. En plus du stockage des applications virtuelles et des tampons de leur entrées/sorties respectives, une partie de la mémoire locale est allouée à la gestion des buffers d'entrées/-sorties et à l'implémentation de la mémoire virtuelle.

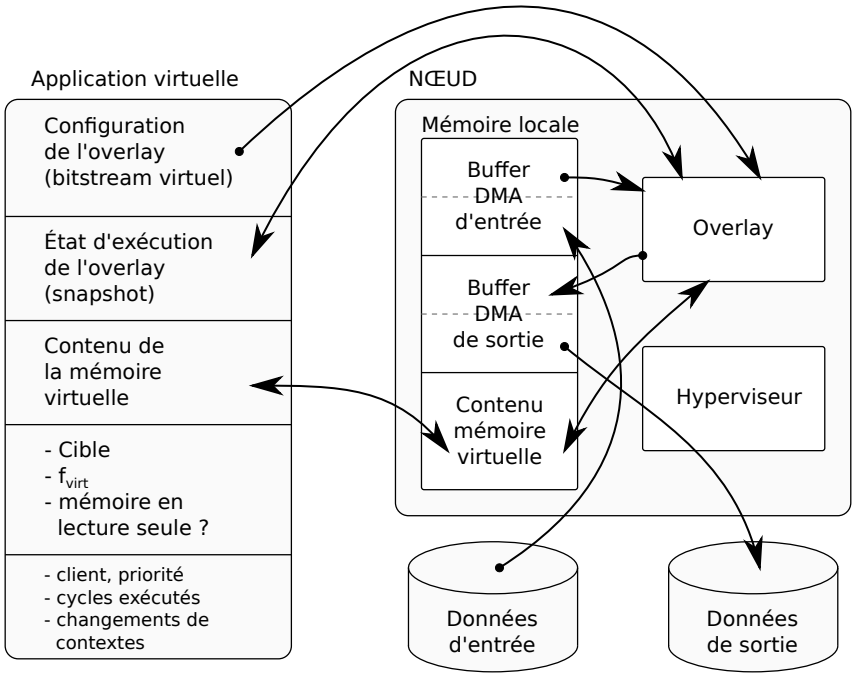


FIGURE 6.2 – Mouvements de données orchestrés par l'hyperviseur

La figure 6.3 montre le déroulement dans le temps des différentes actions et des mouvements de données entre les différents composants illustrés figure 6.2 lors d'un cycle d'exécution d'une application virtuelle, c'est-à-dire du chargement de l'application sur l'overlay, de son exécution, jusqu'à sa sauvegarde. Lors de la restauration d'une application, l'hyperviseur commence par pousser le bitstream virtuel contenu dans l'application virtuelle dans le plan de configuration de l'overlay. Ensuite l'hyperviseur pousse le snapshot des registres applicatifs depuis l'application virtuelle vers le plan de snapshot de l'overlay, puis réalise le transfert entre les registres de snapshot et les registres applicatifs. Le contenu de la mémoire virtuelle est lue depuis l'application virtuelle et écrit dans une zone de la mémoire locale que l'hyperviseur a alloué à cet effet, puis celui-ci configure le contrôleur de mémoire virtuelle (cf chapitre 4) est configuré avec l'adresse mémoire du début de cette zone. Finalement, le contrôleur d'horloge est configuré par la fréquence f_{virt} indiquée dans l'application virtuelle, est l'horloge applicative est activée pour un nombre de cycles choisi.

Lors de son exécution, l'application consomme et produit des données via le contrôleur DMA de l'overlay qui accède à son tour aux données dans la mémoire locale. De même, l'application a accès à la mémoire virtuelle via le contrôleur de

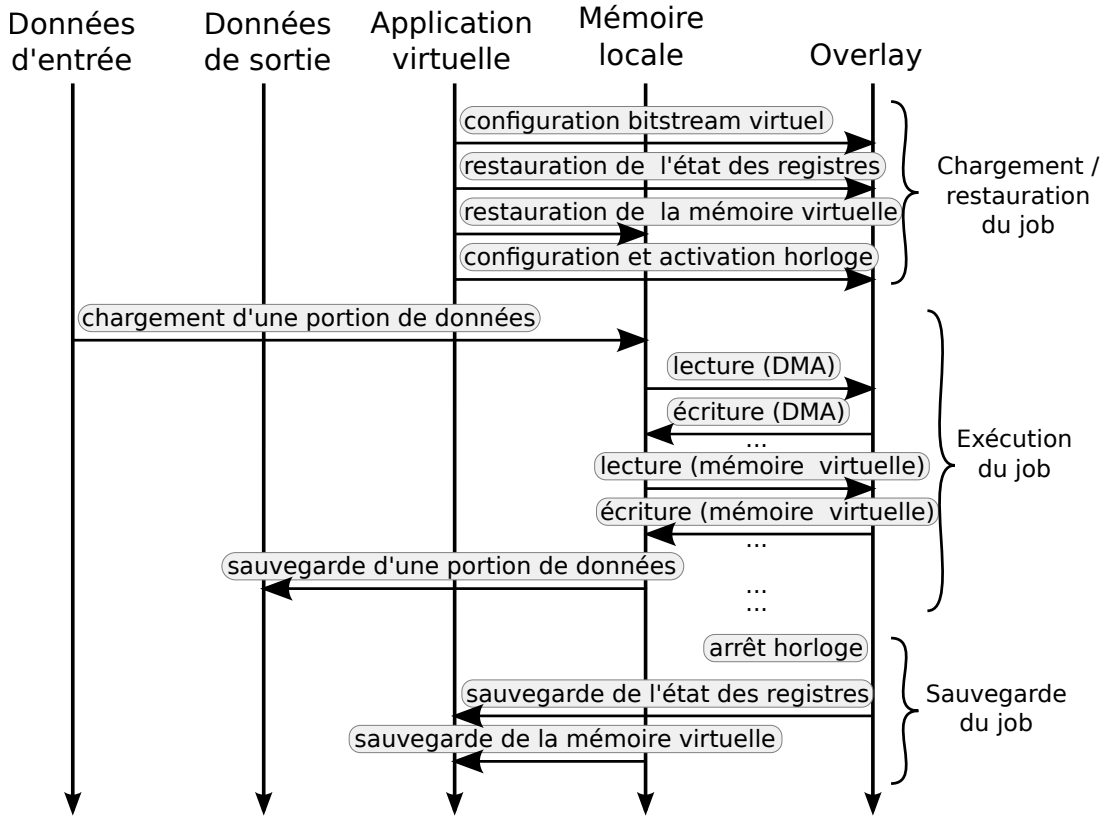


FIGURE 6.3 – Diagramme de séquence du cycle d'exécution d'une application virtuelle

mémoire virtuelle de l'overlay. Lorsque le nombre de cycles d'horloge choisi au chargement de l'application a été réalisé, le contrôleur d'horloge stoppe l'horloge applicative et génère une interruption pour en notifier l'hyperviseur. Celui-ci peut alors effectuer la sauvegarde de l'application virtuelle pour libérer l'overlay. L'hyperviseur peut aussi réaliser une préemption de l'application (figure 6.1) en effectuant une sauvegarde de l'application virtuelle avant complétion du nombre de cycles d'horloge voulu si les buffers d'entrées/sorties ne peuvent être alimentés/vidés au rythme de la consommation et de la production de données par l'application.

Lors de la sauvegarde de l'application, l'hyperviseur commence par arrêter l'horloge applicative si celle-ci n'est pas déjà stoppée. L'état des registres est copié dans le plan de snapshot de l'overlay, puis est extrait de celui-ci pour être écrit dans l'application virtuelle, écrasant l'ancienne version du snapshot des registres applicatifs. Si la mémoire est utilisée en écriture par l'application, le contenu de la mémoire virtuelle est lu depuis la mémoire locale et écrit dans l'application virtuelle, écrasant aussi son ancienne version. Le nombre de cycles d'horloge exécutés par l'application est mis à jour dans l'application virtuelle, et son nombre de changements de contextes est incrémenté.

Optimisations des flux DMA

Pour que le contrôleur DMA de l'overlay ne soit jamais en attente et ait toujours un buffer de données à disposition, les buffers d'entrées et de sorties sont chacun organisés en buffer ping-pong, c'est-à-dire qu'ils sont chacun divisés en deux zones. Le flux d'entrée est écrit depuis le tampon d'entrée dans le premier buffer tandis que le contrôleur DMA de l'overlay lit le deuxième buffer d'entrée qui a préalablement été rempli. Lorsque le contrôleur DMA arrive à la fin du deuxième buffer, il génère une interruption indiquant à l'hyperviseur qu'il peut commencer à remplir le deuxième buffer, et commence à lire le premier, et ainsi de suite. Le buffer de sortie fonctionne de la même façon. Si malgré la présence du tampon d'entrée, le flux d'entrée n'est pas assez rapide pour permettre de remplir un buffer à temps, l'hyperviseur réalise un changement de contexte pour permettre à une application dont le tampon d'entrée est suffisamment plein de s'exécuter.

6.4.3 Optimisations pour le changement de contexte

Le schéma de changement de contexte décrit ci-dessus est le schéma de base, et il est possible de l'optimiser en tirant parti de fonctionnalités de l'overlay vues au chapitre 3. En effet, dans ce schéma, comme le montre la figure 6.4, lors de la configuration par le bitstream virtuel et de l'extraction et de l'injection du snapshot des registres applicatifs, le plan de calcul de l'overlay n'est pas utilisé. Le temps de configuration et d'accès au snapshot étant constant pour un nœud donné, le rapport entre le temps où l'overlay est inactif et le temps où l'overlay est en exécution croît avec la fréquence des changements de contextes. Si l'overlay supporte la préconfiguration (vue au chapitre 3 section 3.2.2), il est alors possible à l'hyperviseur de pousser le bitstream virtuel de la prochaine application à exécuter vers le plan de configuration de l'overlay sans interrompre l'exécution de l'application courante sur le plan de calcul de l'overlay. Le changement de configuration effective se faisant en un cycle d'horloge, il est alors possible d'annuler le temps d'inactivité de l'overlay dû à sa configuration tant que l'hyperviseur a le temps de lire la configuration dans l'application virtuelle et de la pousser dans le plan de configuration de l'overlay avant le prochain changement de contexte.

De même, si le plan de snapshot de l'overlay est implémenté comme présenté en 3.2.4, c'est-à-dire que chaque registre applicatif est appairé avec un registre du plan de snapshot, il est alors possible comme pour la pré-configuration d'effectuer le transfert $plan\ de\ snapshot \leftrightarrow registre\ applicatif$ en un unique cycle d'horloge, et d'accéder au contenu du plan de snapshot de l'extérieur de l'overlay sans interférer avec le fonctionnement des registres applicatifs.

Cependant, si une application A est en cours d'exécution pendant que l'état d'une application B est chargé dans le plan de snapshot, alors l'état de B sera écrasé par l'état de A dans le plan de snapshot lors du transfert
$registres \rightarrow snopshot$

permettant de sauvegarder A, et le transfert
$snopshot \rightarrow registres$
suivant ne fera que restaurer le snapshot de A, et non celui de B. Le transfert
$snopshot \rightarrow registres$
permettant de restaurer l'état de B ne peut pas non plus être réalisé avant que l'état de A ne soit sauvegardé, sinon l'état de A est écrasé et perdu avant d'avoir pu être extrait de l'overlay.

Pour pouvoir pré-charger l'état de B dans le plan de snapshot pendant l'exécution de A et pouvoir récupérer l'état de A pendant l'exécution de B, et ainsi annuler le coût dû à l'accès au snapshot lors d'un changement de contexte, il est donc nécessaire réaliser les échanges
$registre\ applicatif \rightarrow snapshot$ et $snapshot \rightarrow registre\ applicatif$
simultanément dans le même cycle d'horloge. Ainsi, l'état de B prend la place de celui de A dans les registres applicatifs en même temps que l'état de A prend la place de celui de B dans le plan de snapshot, et aucun des deux n'est écrasé.

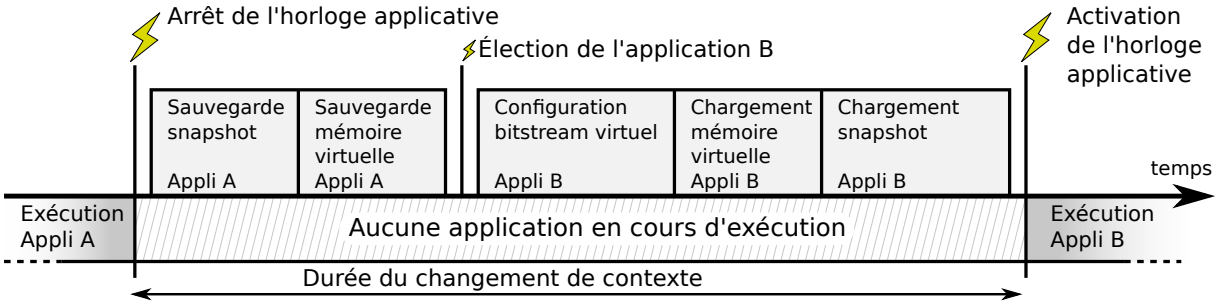


FIGURE 6.4 – Le plan de calcul est inactif sur la durée du changement de contexte.

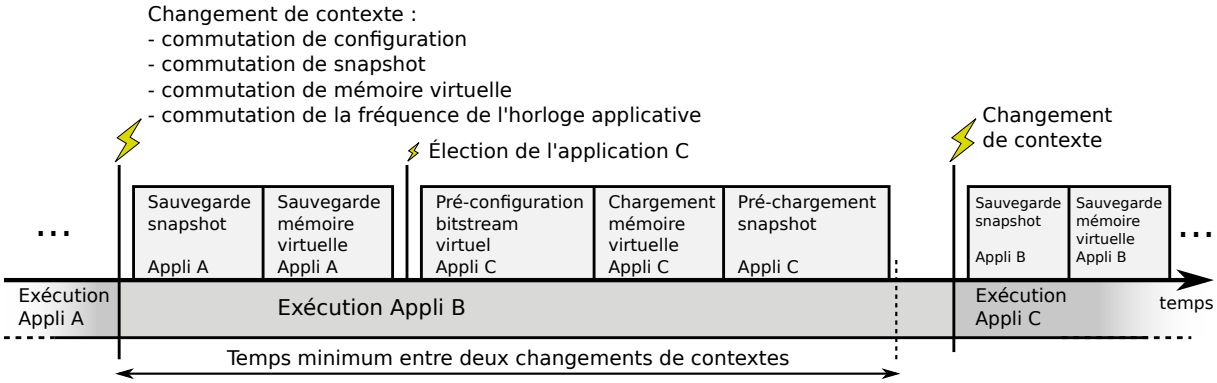


FIGURE 6.5 – Le changement de contexte est étalé en amont et en aval de la commutation de contexte, sans perturber l'exécution des applications.

Les mécanismes de pré-configuration de l'overlay et de pré-chargement et du snapshot des registres applicatifs permettent ainsi de réaliser les transferts
$configuration \rightarrow overlay$,
$snapshot \rightarrow overlay$ et
$overlay \rightarrow snapshot$
sans interrompre le fonctionnement de l'overlay. Cependant, pour pouvoir réduire à zéro le temps pendant lequel aucune application ne peut être active sur l'overlay

pendant un changement de contexte, il est encore nécessaire de pouvoir annuler le coût de sauvegarde et de restauration de la mémoire virtuelle. La solution a été vue en 3.2.5 sur l'implémentation des mémoires virtuelles : il s'agit d'allouer dans la mémoire locale deux fois la taille de la mémoire virtuelle, pour pouvoir utiliser cet espace en mode ping-pong comme pour les buffers DMA. Lorsque la mémoire virtuelle utilisée par l'overlay est mappée sur le premier espace mémoire, l'hyperviseur sauvegarde le contenu du deuxième espace mémoire dans l'application virtuelle de l'application précédente puis y écrit le contenu de la mémoire virtuelle depuis l'application virtuelle de l'application qui a été choisie pour succéder à la suivante. Lors du changement de contexte, l'offset de l'espace mémoire actif peut être configuré en un simple accès au registre de contrôle du contrôleur de mémoire virtuelle de l'overlay.

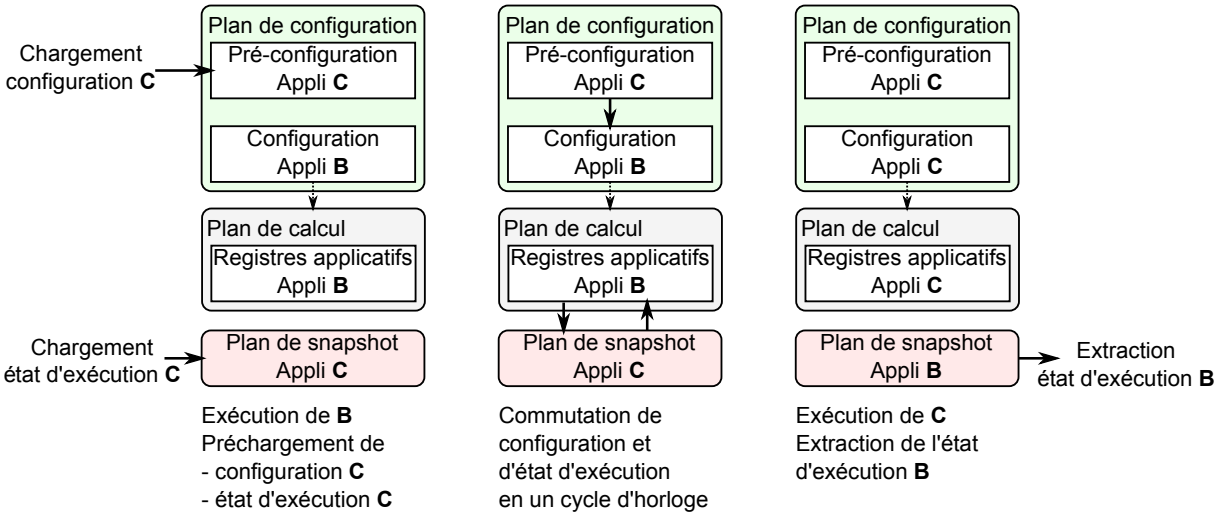


FIGURE 6.6 – Transferts de données aux niveaux des trois plans de l'overlay avant, pendant et après commutation de contexte.

Ces trois optimisations permettent donc d'annuler le temps pendant lequel l'overlay est inactif durant un changement de contexte. La figure 6.5 illustre le fait qu'avec ces mécanismes matériels de doubles buffers pour la configuration le snapshot et la mémoire virtuelle, le chargement de l'application suivante est effectué avant la commutation de contexte tandis que la sauvegarde de l'application précédente est effectuée après. La figure 6.6 détaille les transferts aux niveaux du plan de pré-configuration, de calcul et de snapshot de l'overlay avant, pendant et après la deuxième commutation de contexte de la figure 6.5. La commutation de contexte effective est réalisée instantanément (en un cycle d'horloge) ; ainsi le plan de calcul de l'overlay exécute en permanence une application.

Pour que ces optimisations soient effectives, il faut que le laps de temps alloué à chaque application soit supérieur au temps maximum nécessaire à l'hyperviseur pour sauvegarder la mémoire virtuelle et le snapshot de l'application précédente, d'élire l'application suivante et de configurer et restaurer la mémoire virtuelle et le snapshot de l'application suivante (voir figure 6.5). Cependant, l'application en cours d'exécution peut se trouver en attente de données avant que son laps de temps alloué ne soit entièrement écoulé. Pour éviter que l'overlay ne se trouve

inoccupé, l'application suivante doit systématiquement être élue et pré-chargée dans l'overlay le plus tôt possible (après sauvegarde de l'application précédente) de façon à ce que la commutation de contexte soit préparée et puisse avoir lieu directement après que l'application en cours tombe éventuellement en attente. Si l'application en cours d'exécution tombe en attente de données avant que l'application précédente n'ait finie d'être sauvegardée, que l'ordonnanceur n'ait pas encore élu la prochaine application, ou que la prochaine application n'ait pas fini d'être pré-chargée, alors le plan de calcul de l'overlay se retrouve inoccupé le temps que ces mécanismes puissent s'achever pour donner lieu au changement de contexte suivant.

6.5 Contrôle global, vue haut niveau

Dans la section précédente, il a été vu comment un nœud de calcul peut réaliser les changements de contextes sur l'overlay pour ordonnancer localement différentes applications. Cette section présente la gestion d'un cluster d'overlays. Comme énoncé en 6.2, l'exploitation efficace d'un cluster d'overlay doit permettre à l'administrateur du datacenter de maximiser sa vente de services tout en minimisant ses frais d'infrastructure (Ea), mais aussi en évitant de violer les SLAs des clients pour ne pas avoir à leur payer de pénalités (Eb), c'est-à-dire de respecter des contraintes comme des performances minimales ou un temps d'indisponibilité maximal. Pour ce faire, l'administrateur du cluster peut jouer sur deux éléments : le nombre de ressources allouées à chaque application virtuelle, et le placement des applications virtuelles sur les nœuds. Pour l'allocation, il s'agit d'exécuter plus ou moins d'instances d'application virtuelle d'une même application, et de gérer la priorité des applications virtuelles de façon à ce que chaque nœud les exécutant leur alloue un temps d'exécution adapté. Pour le placement des applications virtuelles sur les nœuds, il s'agit de choisir un nœud pour chaque application virtuelle, en fonction des contraintes de ces applications virtuelles (SLA client), de la configuration statique du cluster (les performances réelles de chaque nœud, qui dépendent des performances de chaque FPGA hôte physique), et de l'état dynamique du cluster (le taux d'utilisation de chaque nœud). Deux exemples de stratégies de placement sont l'équilibrage de charge [109] et la consolidation [108]. L'équilibrage de charge vise à utiliser le plus de nœuds d'exécution possible et de répartir la charge de travail entre tous les nœuds de façon à maximiser l'utilisation des ressources et offrir des performances maximales (débit, temps de réponse). À l'inverse, la consolidation vise à concentrer la charge de travail dans le moins de nœuds possible de façon à diminuer le nombre de nœuds en fonctionnement, dans le but par exemple de diminuer la consommation énergétique du cluster.

Cependant, dans un cadre Cloud, la charge de travail évolue en permanence en fonction de l'utilisation des services par les clients finaux. Ainsi, la gestion des ressources doit se faire dynamiquement (Ee). Notamment, le placement des applications virtuelles sur les nœuds doit pouvoir être remanié à tout moment. Les applications virtuelles placés doivent donc pouvoir être migrées d'un nœud à un

autre, à chaud, c'est-à-dire sans avoir à réinitialiser l'état de l'application virtuelle lors de son déplacement.

6.5.1 Migration à chaud

La migration à chaud d'une application virtuelle suit le même principe que le changement de contexte vu précédemment, à la différence que l'application virtuelle est restaurée sur un overlay différent de celui sur lequel il était exécuté précédemment. La migration à chaud est réalisée en différentes actions. Premièrement, l'administrateur choisi le nœud d'arrivée qui va accueillir l'application virtuelle. Il peut le faire en fonction de sa politique de placement (par exemple le nœud le moins utilisé dans le cas d'équilibrage de charge, ou un nœud dont le taux d'utilisation permet d'accueillir l'application virtuelle sans être surchargé, dans le cas de consolidation). Le nœud d'arrivée peut aussi être choisi pour accélérer l'exécution d'une application virtuelle ciblant un overlay fonctionnellement équivalent mais étant plus performant (grâce à un FPGA hôte plus performant), et/ou étant moins chargé. Ensuite, l'application virtuelle est stoppé et sauvegardé, sur le nœud de départ, puis est transféré via le réseau jusqu'au nœud d'arrivée. Il est aussi nécessaire d'acheminer les données manipulées par l'application virtuelle sur le nœud d'arrivée. Pour cela les flux d'I/O doivent être redirigés non plus vers l'ancien nœud mais vers celui d'arrivée. Il est aussi nécessaire de transférer les données non consommées des tampons d'entrée/sorties du nœud de départ pour cette application virtuelle vers celui d'arrivée. Une fois l'application virtuelle et ses données acheminées vers le nœud d'arrivée, celle-ci peut être restauré et reprendre son exécution sur l'overlay du nouveau nœud d'accueil.

Le temps entre l'arrêt de l'application virtuelle sur le nœud de départ et sa reprise sur celui d'arrivée est un temps pendant lequel l'application est indisponible. Pour répondre à une contrainte de temps d'indisponibilité maximale, l'administrateur doit donc prendre en compte le temps de la migration par rapport à l'avantage qu'apporte le déplacement de l'application virtuelle. Il a été vu dans la section précédente que des optimisations permettent d'annuler le coût du changement de contexte par rapport à l'utilisation du plan de calcul de l'overlay, cependant ces optimisations ne permettent pas d'annuler le temps minimal d'indisponibilité de l'application virtuelle lors d'une migration. Comme le montre la figure 6.7, ce temps d'indisponibilité minimal entre l'arrêt de l'application virtuelle et sa restauration sur un autre nœud correspond au temps nécessaire pour extraire le snapshot et la mémoire virtuelle pour mettre à jour l'application virtuelle, envoyer l'application virtuelle et le contenu non consommé des tampons d'IO via le réseau jusqu'au nœud de destination, puis de configurer, charger le snapshot et la mémoire virtuelle du nœud d'arrivée avant de pouvoir reprendre l'exécution de l'application virtuelle.

Dans le cas d'une forte sollicitation d'une application, l'administrateur peut avoir besoin de cloner une application virtuelle, par exemple pour augmenter spatialement le nombre de ressources allouées à l'exécution de l'application. Le procédé de migration vu ci-dessus peut aussi être utilisé pour cloner une application

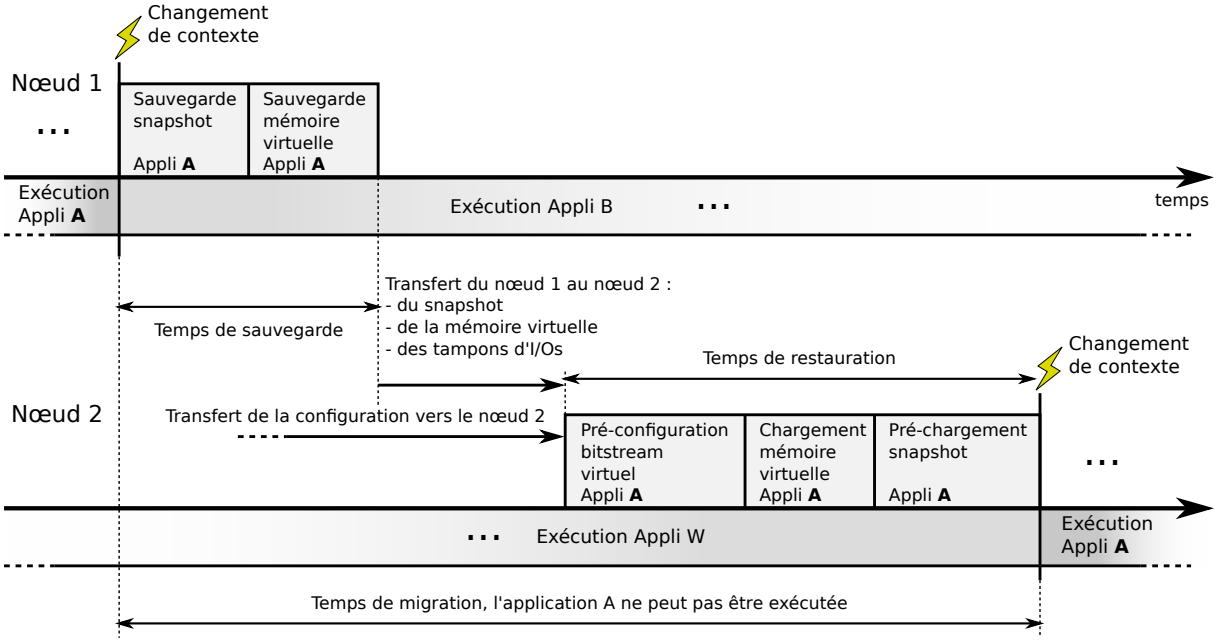


FIGURE 6.7 – Migration d'une application entre deux nœuds.

virtuelle : l'application virtuelle suit le mécanisme de migration et est copiée et restaurée sur le nœud d'arrivée avec un nouvel identifiant, mais l'application virtuelle originale n'est pas supprimée du nœud de départ, et continue son exécution sur celui-ci. Plutôt que d'aiguiller le flux réseau vers le deuxième nœud comme pour la migration, l'administrateur doit créer un nouveau flux.

6.5.2 Résilience aux pannes

Au regard du nombre important de ressources hébergées dans un datacenter, il arrive régulièrement que certaines d'entre elles tombent en panne. Pour limiter l'indisponibilité des applications lors de pannes des nœuds les hébergeant, l'administrateur doit régulièrement effectuer des sauvegardes des applications virtuelles exécutées dans l'infrastructure. Aussi, pour que l'administrateur puisse détecter les pannes des nœuds, il est nécessaire que ceux-ci envoient régulièrement un message de "battement de cœur", c'est-à-dire un message indiquant qu'il fonctionne correctement. Lorsque l'administrateur ne reçoit plus ce message d'un nœud donné, il distribue les applications virtuelles précédemment sauvegardées depuis le nœud silencieux pour les distribuer sur d'autres nœuds fonctionnels de l'infrastructure. Les applications reprennent alors leur exécution depuis leur dernier point de sauvegarde par l'administrateur.

6.5.3 Support de l'évolution des overlays

Comme il a été mentionné en 6.2, en abstrayant leurs architectures fonctionnelles des architectures physiques des FPGAs hôtes, les overlays offrent une stabilité au cours du temps des architectures fonctionnelles "vues" par les applications malgré la diversité et l'évolution des FPGAs physiques utilisés dans l'infrastructure. Cela ne doit pas nécessairement impliquer que le fournisseur ne puisse pas faire évoluer son offre d'overlays, c'est-à-dire l'offre d'architectures fonctionnelles présentées aux clients. Les questions qui se posent sont : quand l'offre d'overlay doit-elle évoluer ? et comment assurer que la solution overlay supporte l'évolution des architectures fonctionnelles ?

Le changement peut être motivé par l'apparition de nouveaux domaines applicatifs, dans le but d'offrir des architectures fonctionnelles optimisées et adaptées aux besoins de ces nouvelles applications. Dans ce cas, l'intégration de nouvelles architectures fonctionnelles dans l'infrastructure ne pose pas de problèmes de portabilité, puisque le domaine applicatif est nouveau et qu'il n'y a donc pas d'anciennes applications à porter.

Le changement peut aussi être motivé par l'évolution des plateformes hôtes, pour mieux tirer parti des améliorations offertes par les nouveaux FPGAs physiques utilisés. En effet, un FPGA physique présentant plus de ressources peut héberger plus d'un overlay. Dans un premier temps, l'overlay peut être instancié plusieurs fois dans le FPGA hôte sans que son architecture fonctionnelle ne soit modifiée, donc dans ce cas il n'y a pas de problème de portabilité. Cependant, l'hyperviseur doit être mis à jour pour gérer plus d'un overlay et les faire apparaitre au contrôleur global comme des nœuds distincts. Dans un second temps, le surplus de ressources du FPGA hôte peut être absorbé par un overlay offrant une architecture fonctionnelle plus riche. Dans ce cas, l'architecture fonctionnelle de l'overlay évolue mais pas les applications, la compatibilité binaire est donc perdue.

Néanmoins, comme la chaîne d'outils utilisée offre le contrôle de la phase de synthèse virtuelle à l'exploitation des overlay, elle permet d'assurer la compatibilité des nouveaux overlays avec des applications plus anciennes. Pour cela, la netlist de l'application doit être resynthétisée, placée et routée (cf chapitre 5) sur la nouvelle architecture fonctionnelle. Par exemple, une netlist mappée pour des LUTs à quatre entrées doit subir une nouvelle synthèse logique pour cibler les LUTs à six entrées d'une nouvelle architecture, avant d'être placée et routée sur celle-ci afin de produire le bitstream virtuel ciblant le nouvel overlay.

Cependant, le fournisseur n'a pas nécessairement accès à la netlist applicative, le client lui ayant transmis une application virtuelle contenant un bitstream virtuel déjà synthétisé pour un overlay donné. Or, le fournisseur doit assurer la transparence de l'exploitation de son infrastructure par rapport aux clients (exigence Ec), donc c'est à lui de gérer la rétrocompatibilité des nouveaux overlays qu'il intègre par rapport aux anciennes applications clientes. Le fournisseur doit donc effectuer une traduction de bitstream virtuel à bitstream virtuel (illustré fi-

gure 6.8), qui d'un point de vue fonctionnel, est une transformation de modèle à modèle. La première étape de cette transformation nécessite donc de reconstruire la netlist applicative à partir du bitstream virtuel de départ.

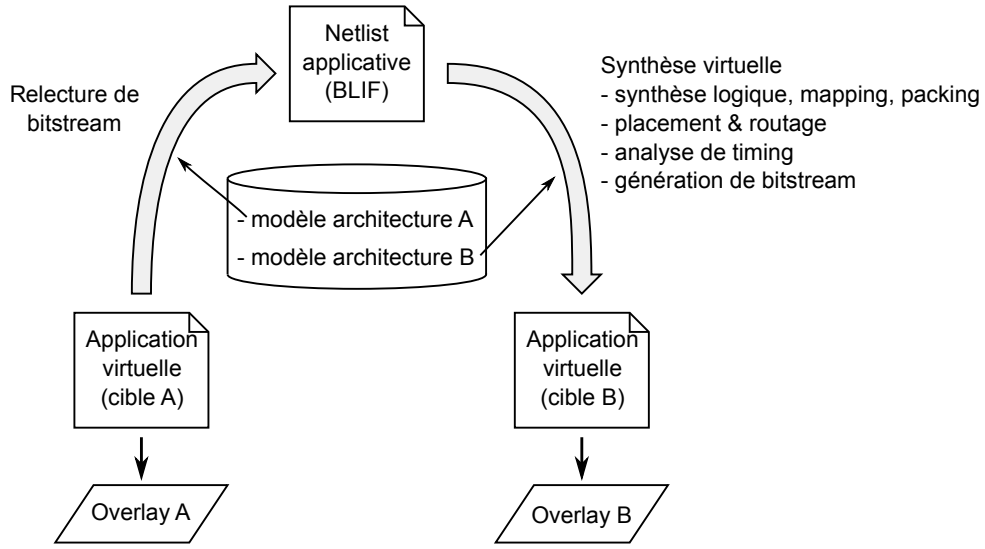


FIGURE 6.8 – La compatibilité d'une application avec un nouvel overlay est assurée par la capacité de traduire un bitstream virtuel pour une nouvelle cible.

La reconstruction de la netlist applicative se fait en relisant le bitstream virtuel de façon à mettre à jour la configuration de chaque élément du modèle du plan de calcul de l'architecture ciblée par l'application virtuelle de départ. Les nœuds de la netlist sont construits à partir de la configuration des LUTs, tandis que les nets reliant ces nœuds sont construits en suivant les chemins reliant les LUTs. Cependant, lors de la relecture du bitstream, chaque élément du modèle du plan de calcul se voit attribué une configuration issue de la partie du bitstream lui correspondant. Or, tous les éléments de l'architecture ne sont pas utilisés pour implémenter l'application. La relecture du bitstream génère donc des chemins et des nœuds issus de la configuration par défaut des éléments non utilisés lors de la première synthèse. Ces chemins et ces nœuds ne participent pas à l'implémentation de la netlist et doivent être filtrés. Pour ce faire, dans un premier temps, le contenu de chaque LUT est analysé de façon à déterminer les entrées de la LUT qui ont un impact sur sa sortie, et ainsi déterminer les entrées de la LUT utilisées par l'application. Ensuite, les chemins menant d'une entrée primaire du circuit ou de la sortie d'une LUT à une sortie primaire du circuit ou à une entrée utilisée d'une LUT sont conservés. Finalement, les LUTs qui ne sont reliées par aucun chemin et les sous graphes isolés de la netlist (c'est-à-dire qui ne sont reliés d'aucune façon aux sorties primaires du circuit) sont supprimés. Une fois la netlist applicative reconstruite, le fournisseur peut alors effectuer la synthèse logique, le placement, le routage et l'analyse de timing sur la nouvelle cible, et générer la nouvelle version de l'application virtuelle ciblant le nouvel overlay.

La reconstruction de la netlist applicative pour synthétiser un bitstream virtuel ciblant une autre architecture overlay demande de connaitre l'architecture ciblée par le bitstream virtuel de départ. Cette information est disponible dans l'applica-

tion virtuelle (cf 6.4.1) qui en plus du bitstream virtuel, intègre (entre autres) les informations identifiant le plan de calcul de l'architecture ciblée par le bitstream.

6.5.4 Infrastructure de déploiement des overlays

Nous avons vu au chapitre 4 comment intégrer différents overlays sur différentes plateformes du commerce, de façon à former des nœuds de calcul intégrables dans un réseau informatique classique. Dans la section 6.4, nous avons vu comment l'hyperviseur – exécuté sur chaque nœud de calcul – fournit des services de type système d'exploitation à l'overlay pour gérer l'exécution d'un ensemble d'applications virtuelles sur l'overlay qui lui est attaché. Nous venons de voir les mécanismes de gestion d'un ensemble de nœuds de calcul connectés à un même réseau (un cluster d'overlays). Cette sous-section traite d'un nœud particulier, unique dans le cluster, que nous appelons le *contrôleur global*, et qui réalise la gestion des nœuds de calcul du cluster.

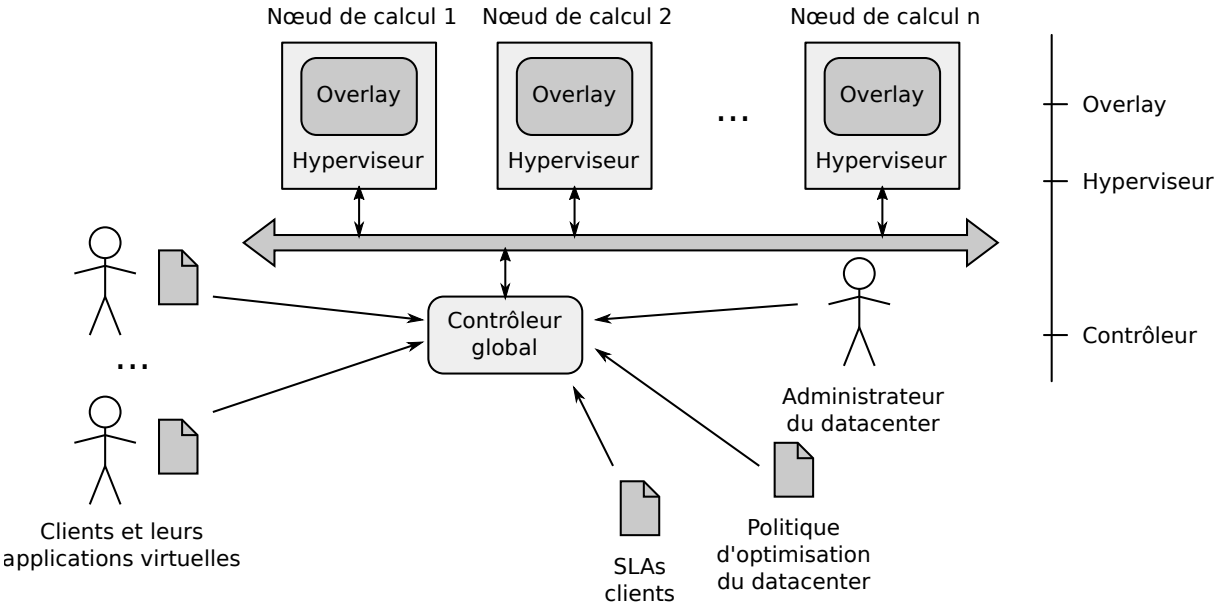


FIGURE 6.9 – Infrastructure de déploiement en trois niveaux : overlay, hyperviseur (local), contrôleur (global).

La figure 6.9 illustre l'infrastructure de déploiement des overlays en trois niveaux : les overlays sont gérés localement par l'hyperviseur au niveau du nœud de calcul, et l'ensemble des nœuds de calcul est géré par le contrôleur global. Le contrôleur global est le seul point d'entrée vu par les clients, qui lui soumettent leurs applications virtuelles. Le contrôleur global ventile ces applications virtuelles sur les différents nœuds de calcul du cluster suivant les directives de l'administrateur du datacenter, c'est-à-dire qu'il doit prendre en compte les SLAs des clients ainsi que la politique d'optimisation indiquée par l'administrateur (comme cibler la performance via l'équilibrage de charge, ou l'économie d'énergie via la consolidation) pour répartir les applications virtuelles sur les nœuds de calcul. Pour pouvoir gérer l'ensemble du cluster, le contrôleur global a en permanence besoin de

connaitre la topologie du réseau, c'est-à-dire de connaitre quels nœuds sont connectés, quels overlays ils intègrent, et l'état de chacun.

Un prototype de contrôleur global a été réalisé par Loïc Lagadec et intégré au framework Madeo. Les nœuds de calcul et le contrôleur global communiquent via une API réseau définie. Le contrôleur global intègre un serveur de noms. Lorsqu'un nœud de calcul se connecte sur le réseau, il s'identifie automatiquement auprès du serveur de nom. Aussi, lorsque le contrôleur global est connecté tardivement au cluster, le serveur de nom peut à tout moment envoyer une requête en broadcast afin de demander à tous les nœuds déjà connectés de s'identifier à nouveau.

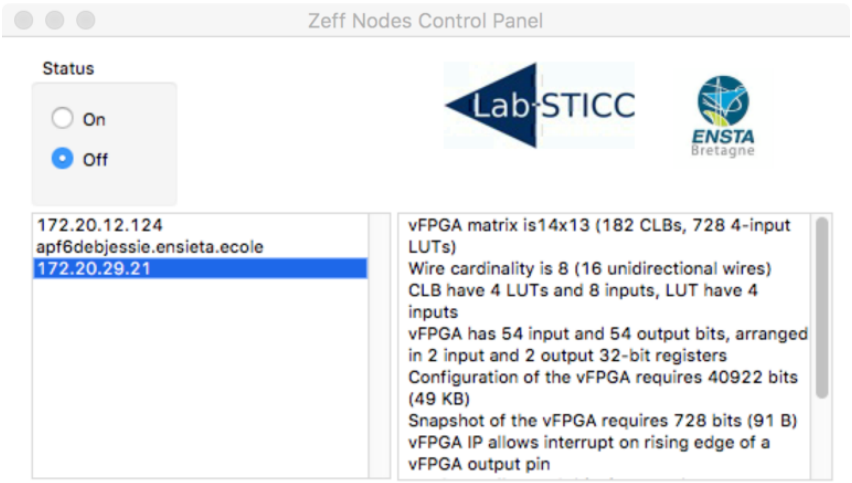


FIGURE 6.10 – Enregistrement des nœuds de calcul auprès du contrôleur global.

Lorsqu'un nœud de calcul s'identifie auprès du serveur de nom du contrôleur global, il lui fournit de manière structurée les informations concernant le plan de calcul de l'overlay qu'il intègre, ainsi que la fréquence physique du FPGA hôte (la fréquence f_{VTPR}, cf 5.5.1). La figure 6.10 illustre l'enregistrement de trois nœuds de calcul auprès du serveur de noms du prototype de contrôleur global réalisé. Le contrôleur global interroge régulièrement chaque nœud sur son état, notamment sur son taux d'utilisation et l'état d'avancement des applications qu'il exécute. Le contrôleur global modélise les nœuds de calcul comme des objets (logiciels) pour avoir une vue centralisée et haut niveau de la topologie du cluster. Cette modélisation permet au contrôleur global de prendre les décisions de migrations.

Lorsque la migration d'une application virtuelle d'un nœud A à un nœud B est décidée, le contrôleur global ordonne au nœud A de retirer l'application virtuelle de sa liste d'application ordonnancées, puis télécharge l'application depuis le nœud A. Si le nœud B comporte un overlay différent du nœud A, le contrôleur traduit le bitstream de l'application virtuelle pour cibler l'overlay du nœud B (cf 6.5.3). L'application virtuelle est ensuite envoyée au nœud B, pour être intégrée à la liste d'applications ordonnancées par ce nœud.

6.6 Modèle de coût

Les mécanismes de changement de contextes et de migration ont été vus dans les sections 6.4 et 6.5.1. La mise en place d'un ordonnancement efficace et la prédiction du coût d'une migration demande d'avoir préalablement caractérisé le temps de reconfiguration sur overlay sur chaque plateforme physique d'implémentation. Cette section présente un modèle de coût qui est fonction des temps utilisés par chacun des composants pour réaliser la sauvegarde et la restauration d'une application virtuelle. Les travaux présentés dans cette section ont été réalisés en collaboration avec M. Mohamad Najem [114, 115].

6.6.1 Modélisation des temps de sauvegarde et de restauration

Dans un premier temps, cinq variables sont définies :

— S_{snap} et S_{config} : la taille en octets du snapshot et du bitstream virtuel, qui dépendent des paramètres et de la granularité du plan de calcul de l'overlay ciblé.

— $S_{memvirt}$: la taille en octets de la mémoire virtuelle utilisée pas une application.

— S_{Buff} : la taille en octets des buffers DMA d'entrée et de sortie.

— $S_{d_{sortie}}$: la taille en octets des données produites par l'application, présentes dans le buffer DMA de sortie, et qui n'ont pas encore été lues par l'hyperviseur au moment de la préemption de l'application.

Sauvegarde d'une application virtuelle

Comme il a été vu dans la section 6.4, le temps pris pour sauvegarder une application virtuelle est la somme des temps demandés pour extraire le snapshot des registres applicatifs (T_{sauv_snap}), pour sauvegarder le contenu de la mémoire virtuelle ($T_{sauv\_memvirt}$). Le temps T_{sauv_Buff} pris pour sauvegarder les données présentes dans le buffer DMA de sortie qui n'ont pas encore été lues par l'hyperviseur doit aussi être pris en compte. Le temps de sauvegarde T_{sauv} est modélisé par l'équation 6.1, où T^0_{sauv} est une constante liée à la plateforme d'implémentation du nœud de calcul. Le temps T_{sauv_snap} peut être approximé par un modèle linéaire faisant intervenir la constante $L^l_{overlay}$ qui est la latence pour la lecture d'un octet depuis le contrôleur de snapshot de l'overlay (cf 4.1.1). De même, les temps $T_{sauv\_memvirt}$ et T_{sauv_Buff} sont approximés linéairement en définissant L^l_{mem}, la latence pour la lecture par l'hyperviseur d'un octet depuis la mémoire locale attachée à l'overlay. L'équation 6.1 peut ainsi être détaillée en l'équation 6.2.

$$T_{sauv} = T^0_{sauv} + T_{sauv\_snap} + T_{sauv\_memvirt} + T_{sauv\_BuffO} \quad (6.1)$$

$$T_{sauv} = T^0_{sauv} + L^l_{overlay} \cdot S_{snap} + L^l_{mem} \cdot (S_{memvirt} + S_{d_{sortie}}) \quad (6.2)$$

Restauration d'une application virtuelle

De manière similaire, le temps T_{rest} demandé pour accomplir le chargement d'une application virtuelle sur l'overlay et la somme des temps nécessaires pour charger le bitstream virtuel dans le plan de configuration (T_{config}), charger le snapshot dans le plan de snapshot (T_{rest_snap}), écrire le contenu de la mémoire virtuelle de l'application dans la mémoire locale dans l'espace qui lui est réservé ($T_{rest\_memvirt}$). Pour que l'application soit en mesure de traiter des données juste après le chargement, il est de plus nécessaire de remplir chacun des deux buffers DMA d'entrée (T_{Buff}). Le temps de restauration T_{rest} est modélisé par l'équation 6.3, où T^0_{rest} est une constante liée à la plateforme d'implémentation du nœud de calcul. En introduisant $L^e_{overlay}$ la latence pour l'écriture d'un octet vers le contrôleur de configuration de l'overlay, et L^e_{mem} la latence pour l'écriture d'un octet dans la mémoire locale attachée à l'overlay, l'équation 6.3 est détaillée en l'équation 6.4.

$$T_{rest} = T^0_{rest} + T_{config} + T_{rest\_snap} + T_{rest\_memvirt} + 2 \cdot T_{Buff} \quad (6.3)$$

$$T_{rest} = T^0_{rest} + L^e_{overlay} \cdot (S_{config} + S_{snap}) + L^e_{mem} \cdot (S_{memvirt} + 2 \cdot S_{Buff}) \quad (6.4)$$

6.6.2 Expérimentation

Le but de cette section est d'évaluer expérimentalement la précision du modèle de coût ci-dessus, et d'explorer et étudier le surcoût de l'ordonnancement avec ce modèle. Dans cette expérimentation, la plateforme utilisée est une carte APF6-SP d'Armadeus [62] qui comporte un processeur ARM i.MX6 Cortex-A9 et un FPGA Cyclone V GX C9 d'Altera. Le processeur communique avec le FPGA via un bus PCI-express Gen1, et le FPGA est connecté à une mémoire RAM DDR3 qui lui est dédiée. Le nœud de calcul implémenté sur cette plateforme utilise le processeur ARM pour exécuter l'hyperviseur (cf 4.2.1). Le FPGA implémente l'IP overlay, une interface PCI et un contrôleur mémoire pour accéder à la mémoire DDR. Ces trois composants sont connectés par une matrice d'interconnexion Avalon [72], qui est l'interface de connexion des IPs proposées par Altera, comme le contrôleur mémoire et l'interface PCI utilisés. Un module noyau (linux) a été développé pour permettre à l'hyperviseur d'accéder depuis l'espace utilisateur aux registres de l'IP overlay et à la mémoire DDR attachée au FPGA, via le bus PCI-express.

Estimation des paramètres et précision du modèle

Cette expérience vise à estimer les paramètres et la précision du modèle pour l'implémentation du nœud de calcul décrite ci-dessus. Pour mesurer les différents temps, l'hyperviseur a été instrumenté pour fournir la date précise de début et de fin de chaque action. Pour trouver les paramètres du modèle, plusieurs configurations du système ont été utilisée : S_{Buff} allant de 1kio à 1Mio, $S_{memvirt}$ allant de 64 octets à 124kio, S_{config} allant de 1kio à 18kio et S_{snap} de 20 octets à 340 octets. Le tableau 6.2 présente l'estimation des paramètres issue d'une régression linéaire réalisée sur Matlab sur plus de 10000 mesures. Les paramètres issus de ces mesures prennent en compte les temps de tous les éléments mis en œuvre pour l'accès aux registres de l'IP overlay et à la mémoire DDR depuis l'hyperviseur, c'est-à-dire : le mécanisme de mémoire partagée du noyau linux pour accéder au bus PCI, le bus PCI physique, l'interconnexion Avalon, l'IP overlay, le contrôleur mémoire implémenté sur le FPGA et la mémoire DDR. Les latences d'écriture sont plus faibles que les latences de lecture du fait que l'écriture PCI en rafale (write burst) été générée par le processeur tandis que la lecture en rafale n'était pas supportée par le système. Le modèle de coût a été comparé aux mesures expérimentales, l'erreur étant calculée comme la différence entre les mesures réalisées des temps de sauvegarde et de restauration et les valeurs estimées par le modèle. Le modèle de coût proposé (équations 6.2 et 6.4) présente une estimation fidèle des temps de sauvegarde et de restauration des applications virtuelles avec un coefficient de détermination R^2 égal à 0.99 et une erreur moyenne de 0.9 ms pour T_{sauv} et T_{rest}. L'erreur relative cumulée est de 8.26% pour T_{rest} et de 14.9% pour T_{sauv}.

Paramètre	**Valeur** [µsec / 32-bit]	**Constante**	**Valeur** [µsec]
$L^l_{overlay}$	2.37	T^0_{sauv}	1854
$L^e_{overlay}$	0.96	T^0_{rest}	4189
L^l_{mem}	2.34	–	–
L^e_{mem}	0.26	–	–

TABLE 6.2 – Estimation des paramètres du modèle de coût pour l'implémentation d'un nœud de calcul sur la plateforme APF6_SP

6.6.3 Scénarios d'ordonnancements

Le modèle étant établi, il est maintenant utilisé pour explorer le surcout de l'ordonnancement pour un ensemble d'application virtuelles. Pour cela, l'algorithme d'ordonnancement ETRR (Equal Time Round Robin) est utilisé. Cet ordonnancement alloue une même tranche de temps fixe appelée *quantum* (dénotée Q) à toutes les applications virtuelles. Dans cette expérimentation, plusieurs configurations du système sont utilisées en faisant varier le quantum Q et la taille des buffers DMA. Les changements de contextes sont réalisés suivant le mécanisme expliqué en 6.4.2, sans utiliser de mécanisme de pré-configuration. Le plan de calcul

de l'overlay est donc inactif pendant les temps T_{rest} et T_{sauv} (cf figure 6.4). L'implémentation du nœud de calcul sur la carte APF6_SP est utilisée, avec un overlay vFPGA-flexible (cf 3.1.2) de 14×13 CLBs comprenant chacun 4 BLEs avec des LUTs à 4 entrées. Cinq applications virtuelles ont été synthétisées pour cet overlay, est sont décrites dans le tableau 6.3. Ces applications ont des profils variés au niveau de leur rythme de production de données F_{prod}, qui dépend de leur implémentation, de leur fréquence applicative f_{virt} et de l'horloge physique de l'overlay f_{VTPR} qui est fixée à 65MHz dans cette expérimentation (cf 5.4). Deux applications utilisent la mémoire virtuelle

Nom	F_{prod} kio/s	**Mémoire virtuelle** octets	**Description**
cordic	15.2	–	Fonction trigonométrique, opération sur 16 bits
filtre RII	12.0	64	Filtre RII d'ordre 4, multiplication séquentielle sur 20 bits
cmult	120	–	Multiplication combinatoire, opérandes 16 bits, résultat 32 bits
pmult	1032	–	Multiplication pipelinée, opérandes 8 bits, résultat 16 bits
sobel	84.0	4096	Filtre de sobel, noyau de 3×3 pixels

TABLE 6.3 – Estimation des paramètres du modèle de coût pour l'implémentation d'un nœud de calcul sur la plateforme APF6_SP

L'ordonnancement ETRR a été réalisé pour les cinq applications jusqu'à ce que chacune ait fini de traiter 4 Gio de données. Pour évaluer le surcoût des changements de contextes, la même manipulation a été effectuée avec un ordonnancement FCFS (First Come First Serve), qui est un ordonnancement où les applications sont exécutées chacune jusqu'à complétion les unes à la suite des autres, et ne présente donc qu'un seul changement de contexte par application. La figure 6.11 présente le temps d'exécution total des cinq applications pour l'ordonnancement ETRR normalisé sur le temps d'exécution total pour l'ordonnancement FCFS, selon différents quantums et différentes tailles de buffer DMA. FCFS prend 499 secondes pour exécuter toutes les applications virtuelles tandis que ETRR demande entre 504 et 560 secondes suivant le quantum et la taille des buffers DMA utilisés, ce qui correspond à un surcoût d'ordonnancement entre 1% et 12%. Le quantum et la taille des buffers DMA ont donc un impact important sur le surcoût de l'ordonnancement.

6.6.4 Choix du quantum pour l'ordonnancement avec préconfiguration

Comme il a été vu en 6.4.3, le surcoût de l'ordonnancement peut être minoré en utilisant le système de double buffer des plans de configuration et de snapshot de l'overlay. Ce système de double buffer permet de charger et extraire la configuration et le snapshot tout en laissant le plan de calcul exécuter l'application en cours, et permet une commutation de contexte en un cycle d'horloge. Comme décrit en 6.4.3, pour que ce mécanisme soit efficace, il faut que le quantum choisi soit supérieur au temps de sauvegarde et de restauration : $Q_{min} = T_{sauv} + T_{rest}$. Pour

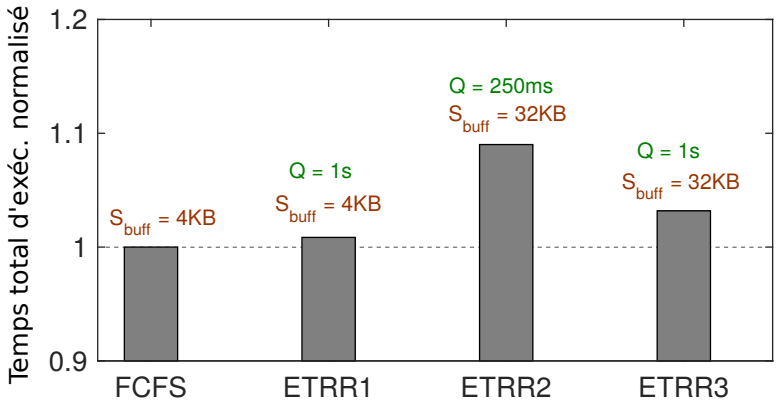


FIGURE 6.11 – Temps d'exécution total pour l'ordonnancement ETRR normalisé sur le temps d'exécution total pour FCFS, pour différentes configurations de Q et S_{Buff}

un overlay donné, la taille de la configuration et du snapshot S_{config} et S_{snap} sont fixés. La quantité de mémoire virtuelle utilisée est différente pour chaque application et peut être majorée par la taille de mémoire virtuelle maximum $S_{memvirt\_max}$ fixée pour l'implémentation du nœud de calcul par le fournisseur. La quantité de données $S_{d_{sortie}}$ produites par l'application, présentes dans le buffer DMA de sortie et qui n'ont pas encore été sauvegardées par l'hyperviseur au moment de la préemption de l'application dépend du rythme de production de données F_{prod} par l'application et du moment où a lieux la préemption. $S_{d_{sortie}}$ peut être majorée par la taille du buffer DMA de sortie S_{Buff}.

Ainsi, une fois les paramètres du modèle de coût caractérisés pour une implémentation d'un nœud de calcul, le quantum minimum de l'ordonnanceur à utiliser pour assurer l'efficacité du mécanisme de pré-configuration au regard du taux d'utilisation du plan de calcul de l'overlay peut être déterminé par :

$$\begin{aligned} Q_{min} =& T^0_{sauv} + L^l_{overlay} \cdot S_{snap} + L^l_{mem} \cdot (S_{memvirt\_max} + S_{Buff}) + \\ & T^0_{rest} + L^e_{overlay} \cdot (S_{config} + S_{snap}) + L^e_{mem} \cdot (S_{memvirt\_max} + 2 \cdot S_{Buff}) \end{aligned}$$

Pour éviter l'inactivité du plan de calcul dans le cas où une application tombe en attente de données avant l'écoulement de Q_{min} après sa restauration, l'ordonnanceur doit s'assurer que chaque application élue a dans les tampons d'entrées du nœud assez de données en attente d'être traitées (S_{tampon_in}) pour l'occuper sur au moins la durée de Q_{min}. C'est-à-dire :

$$S^{app\_elue}_{tampon\_in} \geqslant F^{app\_elue}_{prod} \cdot Q_{min}$$

Le tableau 6.4 présente le temps maximum de chargement pré-changement de contexte et de sauvegarde post-changement de contexte des applications d'exemple du tableau 6.3, selon les paramètres du modèle caractérisés pour la plateforme

APF6_SP, et un buffer DMA fixé à 4 kio. Le facteur variant entre ces applications est la quantité de mémoire virtuelle utilisée qui doit être sauvegardée et restaurée lors des changements de contextes. Les temps maximums de restauration et de sauvegarde correspondent donc à l'application *sobel* qui utilise 4 kio de mémoire virtuelle. Le quantum minimum pour l'ordonnancement de cet ensemble d'applications sur l'APF6_SP est donc $Q_{min} = 12.94ms$. Le tableau 6.4 est complété par la taille minimale $S_{tampon\_in\_min}$ des données d'entrées qui doivent être disponible lors de l'élection d'une application par l'ordonnanceur pour éviter que l'application élue ne tombe en attente de données avant l'écoulement de Q_{min} et ne fasse diminuer le taux d'utilisation de l'overlay.

Application	T_{rest} (ms)	T_{sauv} (ms)	$S_{tampon\_in\_min}$ (o)
cordic	5.97	4.30	202
filtre RII	5.98	4.34	195
cmult	5.97	4.30	1590
pmult	5.97	4.30	13673
sobel	6.24	6.70	1113

TABLE 6.4 – Temps maximums de sauvegarde et de restauration sur l'APF6_SP ($S_{Buff} = 8kio$) pour l'ensemble d'applications du tableau 6.3, et taille minimale du tampon d'entrée.

Dans le cadre d'une exploitation Cloud des overlays, la fréquence moyenne de production des données F_{prod} pour chaque application n'est pas indiquée à l'administrateur. En pratique, l'ordonnanceur doit donc profiler chacune des applications au cours de leur exécution pour pouvoir : i) ajuster dynamiquement le quantum, de façon à ce qu'il soit assez petit pour assurer un ordonnancement flexible, tout en restant supérieur à Q_{min} pour assurer le taux d'utilisation maximal de l'overlay ; et ii) prédire quantité minimale de données d'entrées $S_{tampon\_in\_min}$ qui doit être disponible à une application pour pouvoir l'élire sans risque.

6.7 Illustration

Une démonstration a été proposée [116] à la conférence internationale Design & Architectures for Signal & Image Processing (DASIP) 2016. Cette démonstration met en œuvre un contrôleur global et deux nœuds de calcul implémentant le même overlay sur des FPGAs de modèles et de vendeurs différents. L'installation est présentée figure 6.12. Le premier nœud de calcul est implémenté sur la plateforme APF6_SP présentée en 6.6.2 et implémente l'overlay sur un FPGA Cyclone V d'Altera ; le deuxième nœud de calcul est implémenté sur une carte Nexys4 comportant un FPGA Artix 7 de Xilinx, relié via un pont USB/UART à 4 Mb/s à une Raspberry Pi qui exécute l'hyperviseur. Les deux nœuds sont connectés via un commutateur réseau Ethernet à un ordinateur qui exécute le contrôleur global. Dans le cadre de cette démonstration, le contrôleur global est un script qui séquence au

cours du temps l'envoi de différentes commandes aux deux nœuds. L'ordinateur et les deux nœuds partagent un système de fichier NFS (Network File System) partagé via le réseau pour que chacun ait accès aux applications virtuelles, et les flux d'entrées des applications sont lus et depuis des fichiers. L'overlay mis en œuvre est un vFPGA-flexible de 14×13 CLBs de 4 BLEs comportant des LUTs à 4 entrées, avec 16 pistes par canal de routage. Le contrôleur DMA permet des flux d'entrées et de sortie de 16 bits et le contrôleur de mémoire virtuelle permet d'adresser au maximum 2^{16} mots de 16 bits.

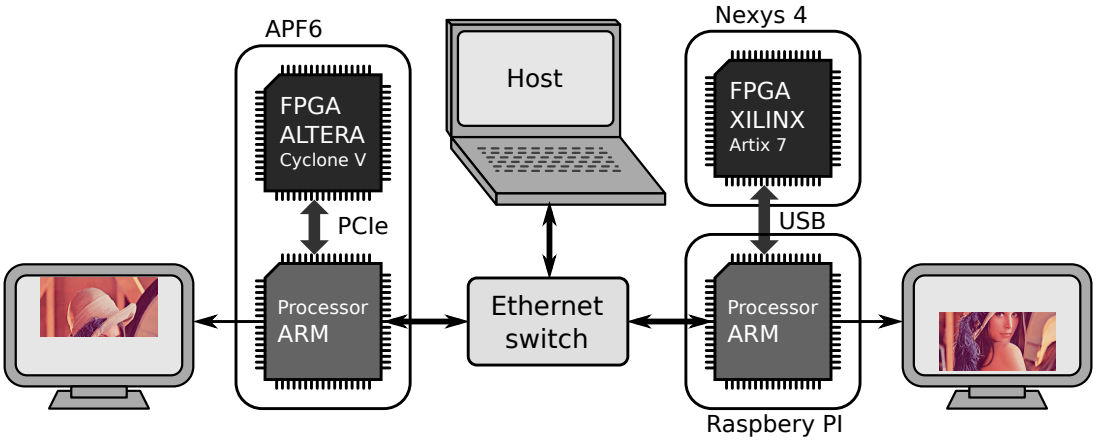


FIGURE 6.12 – Setup de la démonstration : deux nœuds de calculs implémentés sur des FPGAs différents et un contrôleur global (désigné *host*).

Trois applications virtuelles ont été synthétisées : un filtre de lissage gaussien, un filtre d'accentuation et un filtre de Sobel. Ces applications sont basées sur des noyaux de convolution de 3×3 pixels. Le flux d'entrée de chaque application est une image dont chaque pixel est codé sur 16 bits (5 bits pour le bleu, 6 pour le vert et 5 pour le rouge). Les applications consomment leur flux d'entrée pixel par pixel pour produire leur flux de sortie dans le même encodage. Les trois canaux de couleur de chaque pixel sont traités en parallèle. Pour avoir simultanément à disposition les neuf pixels du noyau à partir du flux d'entrée, les applications font usage de 4 kio de la mémoire virtuelle pour tamponner trois lignes de 512 pixels de l'image. Aussi, l'hyperviseur a été modifié pour visualiser sur un écran attaché à chaque nœud l'avancée des traitements : une partie de l'écran de chaque nœud est réservée chacune des applications virtuelles ordonnancées sur le nœud. Lorsqu'un buffer DMA de sortie est plein, l'hyperviseur le lit et l'affiche sur la partie de l'écran réservée à l'application qui l'a produit.

Cette démonstration illustre la capacité de la solution overlay à :
— donner une vue homogène d'un ensemble hétérogène de FPGAs,
— ordonnancer différentes applications sur un même overlay,
— offrir un contrôle dynamique via la migration à chaud d'application entre différents overlays.

Dans un premier temps, la même application virtuelle est envoyée aux deux nœuds de calcul qui l'exécutent chacun de leur côté. L'overlay implémenté sur chacun des deux nœuds permet ainsi d'exécuter le même binaire applicatif sur des FPGAs hôtes de modèles, d'architectures et de vendeurs différents.

Dans un deuxième temps, les trois applications virtuelles sont ajoutée à la liste d'ordonnancement d'un même nœud de calcul. Celui-ci exécute chaque application pendant une demie seconde avant de l'interrompre pour passer à l'application suivante, jusqu'à complétion des trois applications. L'avancement successif des trois applications est visualisé sur l'écran attaché au nœud.

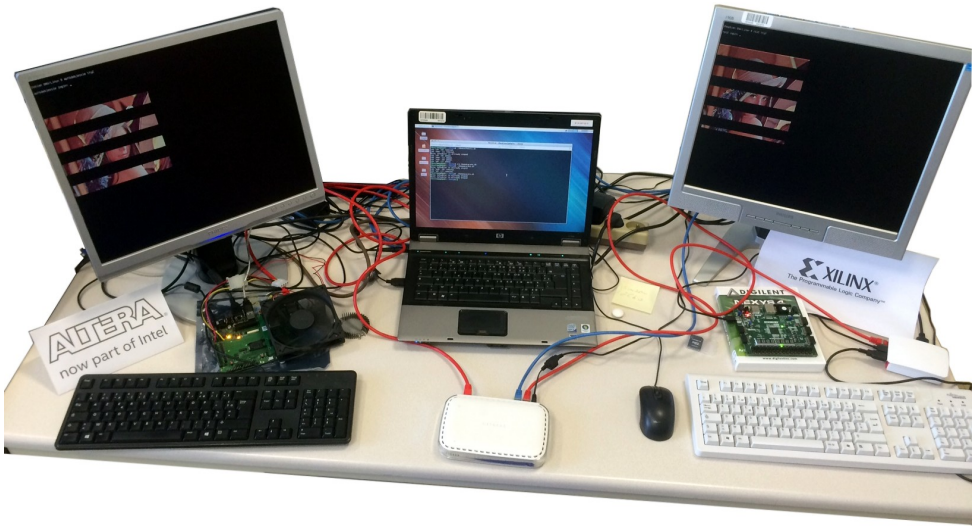


FIGURE 6.13 – Photographie de la démonstration lors de migrations successives d'une même application virtuelle entre les deux nœuds de calcul. Les images affichées sur les deux écrans sont complémentaires.

Finalement, une application virtuelle est envoyée sur un nœud, puis est interrompue pour être restaurée sur le deuxième nœud. L'application virtuelle comprenant entre autre l'intégralité du snapshot des registres applicatifs et le contenu de la mémoire virtuelle tamponnant les trois linges de l'image d'entrée est ainsi migrée et restaurée sur le deuxième nœud. L'application reprend son exécution de manière transparente. La figure 6.13 présente un photographie de la démonstration lors de migrations successives de l'application de filtrage gaussien entre les deux nœuds : les images affichées les écrans de droite et de gauche présentent des portions de l'image d'entrée filtrée par chacun des nœuds, entrelacées de bandes noires correspondant à la migration de l'application sur l'autre nœud. Les deux images sont complémentaires au pixel près ; leur superposition correspond à la totalité de l'image filtrée par l'application.

6.8 Résumé

Dans ce chapitre ont été présentés le cadre du Cloud et les exigences auxquelles doivent répondre des ressources de calcul pour être intégrées et utilisée dans l'infrastructure d'un datacenter. Notamment, les ressources de calcul doivent pouvoir être partagées entre différents clients/applications, permettre d'optimiser leur taux d'utilisation et de permettre à l'administrateur du datacenter de gérer dynamiquement la charge de travail sur ces ressources. De plus, la gestion de l'infrastructure

doit supporter l'hétérogénéité des ressources utilisées, et cette hétérogénéité ainsi que les détails techniques de l'infrastructure doivent être transparents pour les clients.

Différentes solutions ont été examinées concernant l'intégration de FPGAs dans l'infrastructure d'un datacenter comme ressources de calcul accessibles aux clients comme accélérateurs reconfigurables. Grâce à l'intégration d'un plan de snapshot, les overlays sont une solution qui répond à toutes ces exigences pour la mise en œuvre de FPGAs dans le Cloud.

Les mécanismes d'exploitation des overlays ont été expliqués, en s'appuyant sur l'abstraction des plateformes FPGA physiques en nœuds de calcul vus au chapitre 4. La mise en œuvre de l'ordonnancement et de migration d'applications a été détaillée. Il a aussi été montré qu'en plus d'assurer une vue homogène d'un ensemble de ressources physiques hétérogènes, la solution overlay supporte aussi l'évolution des architectures overlays utilisées.

Ensuite, un modèle de coût a été élaboré et évalué expérimentalement. Ce modèle permet de caractériser une plateforme physique pour ensuite mettre à jour les paramètres d'ordonnancement optimaux. Finalement, la mise en œuvre de la solution overlay a été illustrée sur un cluster de deux prototypes de nœuds de calcul implémentés sur des FPGAs de modèles et de marques différents et contrôlés via le réseau.

Conclusion et perspectives

Synthèse des travaux

Afin de répondre à la demande croissante de puissance de calcul disponible dans le Cloud, de nouveaux accélérateurs matériels se voient intégrés dans l'infrastructure des datacenters. De part leur capacité de reconfiguration et les performances qu'ils offrent, aussi bien en puissance de calcul qu'en efficacité énergétique, les FPGAs sont de bons candidats pour accélérer des applications dans ce contexte. Cependant, les FPGAs présentent certaines caractéristiques qui font obstacle à leur utilisation dans le Cloud : premièrement, la programmation des FPGAs se fait à bas niveau et demande une expertise en micro-électronique aussi bien qu'une expertise propre aux outils de programmation. L'expérience de développement sur FPGA est loin de l'expérience de développement logicielle à laquelle sont habitués les clients du Cloud, ce qui peut empêcher leur adoption par ces utilisateurs. Deuxièmement, les FPGAs ne présentent pas de mécanismes natifs permettant de les intégrer dans le modèle de gestion dynamique d'une infrastructure Cloud, et la non-compatibilité entre les FPGAs ainsi que la difficulté de portage d'une application d'un FPGA à un autre fait obstacle à leur exploitation dans l'infrastructure d'un datacenter, qui présente souvent des ressources hétérogènes.

Dans ce travail, nous proposons d'utiliser des architectures overlay afin de faciliter l'adoption, l'intégration et l'exploitation de FPGAs dans le Cloud. Les overlays sont des architectures reconfigurables elles-mêmes implémentées sur FPGA. En tant que couche d'abstraction matérielle placée entre le FPGA et les applications, les overlays permettent de monter le niveau d'abstraction du modèle d'exécution présenté aux applications et aux utilisateurs, ainsi que d'implémenter des mécanismes facilitant leur intégration est leur exploitation dans une infrastructure Cloud.

Contrairement aux FPGAs qui sont des circuits directement disponibles dans le commerce, les overlays doivent être conçus puis implémentés sur FPGA, et leur outillage de programmation doit être réalisé avant de pouvoir utiliser les overlays. Ce travail présente une approche verticale adressant tous les aspects de la mise en œuvre d'overlays dans le Cloud en tant qu'accélérateurs reconfigurables par les clients : de la conception et l'implémentation des overlays, leur intégration sur des plateformes FPGA commerciales, la mise en place de leurs mécanismes d'exploi-

tation, jusqu'à la réalisation de leurs outils de programmation. L'environnement réalisé est complet, modulaire et extensible, il repose en partie sur différents outils existants, et démontre la faisabilité de notre approche.

Lors du prototypage d'overlays, étant données les dimensions de l'espace de conception qu'ils présentent, et le temps de synthèse d'un design (ici l'overlay) sur FPGA, le processus de synthèse physique des overlays doit être automatisé. Or, les overlays sont des designs FPGA atypiques, et l'outil de synthèse peut rencontrer des difficultés et demander une intervention manuelle pour aboutir. Pour répondre à ce problème, les architectures sont modélisées puis générées de façon à faciliter la synthèse, grâce notamment au VTPRs. Afin de répondre aux exigences d'une gestion dans le Cloud, l'implémentation des overlays générés intègre des mécanismes de sauvegarde et de restauration du contexte d'exécution des applications, ainsi qu'un mécanisme de configuration permettant de minimiser le coût d'une commutation de contexte lors de l'ordonnancement d'applications.

Le flot de programmation mis en place permet d'aboutir à un binaire de configuration propre à l'architecture overlay qu'il cible, à partir d'une description de l'application. Ce flot est indépendant des outils de synthèse fournis par les constructeurs FPGA : une fois l'overlay synthétisé sur FPGA, l'outillage constructeur n'est plus nécessaire et la synthèse applicative passe uniquement par le flot de programmation ciblant l'overlay. Ce flot permet d'évaluer les performances des applications réalisées, et implémente une méthode d'analyse de timing sur overlay permettant d'abstraire les performances des applications sur l'overlay des performances de cet overlay sur un FPGA hôte donné. Le flot permet d'accompagner le concepteur applicatif dans son processus de développement en permettant le débogage des applications à chacune des étapes de synthèse, de leur description RTL à l'exécution sur carte.

Pour rendre les overlays compatibles au modèle de gestion des ressources dans le Cloud et faciliter leur intégration sur différentes plateformes FPGA commerciales, les overlays sont d'abord intégrés dans une IP avec les contrôleurs qui leur sont spécifiques. L'interface de cette IP est fixe quelles que soient les spécificités de l'overlay qu'elle intègre, et permet de facilement venir greffer l'IP dans un nouveau design. Cette IP est ensuite intégrée sur la plateforme FPGA dans un système de type SoC, comprenant entre autre un processeur et une interface réseau. Le logiciel embarqué réalisé permet d'alimenter l'overlay en données, de le contrôler et d'ordonnancer des applications. Il communique avec l'extérieur via un protocole réseau défini, permettant ainsi de banaliser l'accès à l'overlay depuis un réseau informatique, rendant ainsi transparentes les spécificités physiques de chaque plateforme. Une fois connectée au réseau, chaque plateforme pourvue d'un overlay et de son logiciel embarqué est considérée comme un nœud de calcul overlay par un unique nœud de contrôle. Ce nœud de contrôle est le point d'entrée auquel les clients soumettent leurs applications, qui sont réparties sur les nœuds de calcul. Le nœud de contrôle orchestre dynamiquement l'exécution des applications sur les différents nœuds de calcul, afin d'appliquer sa politique d'optimisation de l'infrastructure, peut initier des migrations d'applications d'un nœud de calcul à une autre.

L'environnement réalisé a été mis en œuvre et validé par la conception, la génération, l'implémentation et l'intégration d'un même overlay sur deux plateformes FPGA différentes des constructeurs Xilinx et Altera. Des applications ont été synthétisées via le flot de programmation mis en place, puis exécutées, ordonnancées et migrées entre les deux plateformes réalisées. À notre connaissance, il s'agit de la première expérience illustrant la migration à chaud (i.e. avec sauvegarde et restauration de contexte) d'un même binaire de configuration entre deux FPGAs différents.

Ces travaux ont été valorisés dans différentes publications et communications. L'article de journal [114] publié dans *Journal of Systems Architecture* et l'article de conférence [115] (FPGA4GPC) traitent de la modélisation des temps pris par les différents mécanismes mis en jeux lors d'une commutation de contexte sur l'overlay, afin d'évaluer le surcoût de différents types d'ordonnancement. La démonstration de la migration à chaud d'applications entre deux FPGAs de marques différentes a été présentée à la conférence DASIP [116]. L'article [58] (Applied Reconfigurable Computing) démontre que les principes mis en œuvre pour la génération d'architectures et l'analyse de timing sur overlay sont aussi applicables et pertinents dans le cas d'architectures reconfigurables implémentées non pas sur FPGA mais directement sur une cible ASIC. Les articles [63] (ReCoSoC) et [117] (COMPAS) présentent ZeFF comme plateforme de prototypage et d'exploitation d'overlays. Les communications [118, 119] ont permis d'ouvrir des discutions sur la thématique des overlays au sein du GDR SoC-SiP.

Perspectives

Ces travaux ont été réalisés de façon à produire un environnement modulaire, extensible et réutilisables dans d'autres travaux. Il existe de nombreuses pistes pour améliorer les contributions de cette thèse, certaines à court terme, d'autres à plus long terme.

L'environnement réalisé est un prototype. Une des premières pistes pour améliorer le taux d'utilisation des ressources en limitant le surcoût en surface de l'overlay serait de considérer différentes structures qui ne sont pas encore prises en charge dans ce travail, telles que les LUTs fracturables [120]. La prise en charge de pistes de routage ayant une longueur dépassant plus d'un CLB permettrait de diminuer la longueur du chemin critique des applications en termes de VTPRS, et ainsi d'augmenter leur fréquence maximale de fonctionnement Une piste pour diminuer le surcoût de l'overlay sans nécessairement diminuer leur routabilité serait de diminuer la flexibilité des crossbars servants à alimenter les entrées des LUTs depuis les entrées des CLBs.

L'espace de conception des overlays est vaste, et des changements architecturaux des overlays peuvent avoir un impact important sur les différentes étapes nécessaires à leur mise en œuvre, particulièrement sur leurs outils de program-

mation. Bien que pour démontrer la validité de notre approche, l'environnement réalisé a été focalisé sur les overlays grain fin, qui sont, à notre sens, le cas le plus compliqué à mettre en œuvre, les overlays grain fin sont aussi ceux qui présentent les surcouts les plus importants. Aussi, l'une des pistes d'amélioration de ce travail est d'étendre l'environnement réalisé pour supporter des overlays de granularité plus élevée, ainsi que des architectures hybrides intégrant des opérateurs gros grain dans leur matrice grain fin (comme c'est le cas des FPGAs commerciaux qui intègrent des blocs RAM et des blocs DSP). L'intégration dans une matrice grain fin de blocs comprenant un multiplieur et un accumulateur a été investiguée a été réalisée dans le cadre du projet ANR ARDyT.

L'environnement réalisé permet de segmenter une application en sous modules, puis d'ordonnancer sur un même overlay l'ensemble de ces sous modules de façon à ce que la sortie de l'un, placée dans un tampon mémoire, puisse être consommée comme entrée d'un autre, permettant ainsi d'implémenter des pipelines de sous modules aussi longs que désiré. Cette approche permet de présenter aux applications des ressources overlay en quantités virtuellement infinies, cependant, elle n'est réalisable qu'à condition que chacun des sous modules (qui ne peuvent pas à nouveau être fragmentés), ne demande pas plus de ressources que celles disponibles dans l'overlay. Une piste intéressante serait d'ajouter un mécanisme de masquage des bits du plan de snapshot, de le coupler avec un plan de configuration multi contextes, afin d'injecter la valeur de certains des éléments séquentiels résultants de l'exécution d'une configuration dans le plan de calcul propre à une autre configuration. Il serait ainsi possible d'échanger des données entre sous modules sans passer par un tampon mémoire extérieur à l'overlay, et de diminuer la taille minimale des sous modules à un seul niveau de registres applicatifs, permettant ainsi d'exécuter des applications plus importantes sur des overlays limités en ressources. Cela aurait un impact important sur l'outillage de synthèse virtuelle qui devrait alors effectuer simultanément le placement et routage de différentes configurations.

Les overlays peuvent aussi servir pour augmenter la sécurité d'un design. En effet, le rôle et la signification de chacun des bit du bitstream d'un overlay est déterminé lors de la génération de l'architecture de l'overlay, et est indépendante du FPGA sous-jacent. Ainsi, un utilisateur de FPGA voulant obfusquer ses applications peut réaliser un overlay dont il ne diffuse pas les sources RTL afin d'ajouter un niveau de sécurité supplémentaire (en effet, pour la plupart des FPGAs commerciaux, le format de bitstream n'est pas non plus ouvert).

La capacité de migrer à chaud des applications d'un overlay à un autre peut être étendue au-delà du périmètre d'un seul datacenter. En effet, on peut imaginer un Cloud économe en énergie et distribué géographiquement sur un ensemble de petits datacenters, où chacun est alimenté par des sources d'énergies plus ou moins volatiles, telles que des éoliennes, des panneaux solaires ou des sources d'énergies marémotrices. Dans ce contexte, l'utilisation d'overlays permettrait de migrer les applications matérielles d'un datacenter à un autre (de manière transparente aux utilisateurs) de façon à ce que les traitements soient réalisé au plus près des sources d'énergie. On peut ainsi imaginer des applications matérielles migrant le

long de la côte afin de suivre la marée une nuit sans vent.

Une application directe de ces travaux est l'enseignement. En effet, lors des travaux pratiques, les professeurs disposent de plusieurs générations et modèles de cartes FPGAs. En déployant un même overlay sur ces cartes, il serait alors possible d'utiliser toutes les cartes disponibles tout en présentant un environnement de développement unique aux élèves. Par ailleurs, les overlays sont un outil formidable pour comprendre le fonctionnement i) des architectures reconfigurables telles que les FPGAs, et ii) des outils de synthèse et des transformations effectuées sur une netlist générique pour aboutir à un binaire de configuration. En effet, les spécifications publiques des FPGAs commerciaux ne divulguent pas certains détails, et l'outillage propriétaire fourni par les constructeurs utilise des algorithmes et des formats de fichier qui ne sont pas ouverts. En revanche, les overlays permettent de mettre en œuvre des architectures ouvertes sur FPGA, et les outils de synthèse les ciblant sont indépendant des outils constructeur. De plus, la capacité de mettre en pause l'horloge applicative, de l'activer pour un nombre de cycles donné et d'accéder à l'état d'exécution de l'application permettent de visualiser en pas à pas l'exécution matérielle de l'application. Finalement, en donnant une vue discrète de phénomènes qui sont normalement analogiques, VTPRs font des overlays un excellent moyen de comprendre l'analyse de timing : lors de la simulation RTL d'un overlay exécutant une application (via Modelsim ou GHDL), la présence des VTPRs permet de suivre la propagation des signaux applicatifs combinatoires de VTPR en VTPR, ainsi que les changements de valeur transitoires en sortie des LUTs dus aux différents temps de propagation de leurs signaux d'entrées.

Glossaire

ALM Adaptive Logic Module
ALU Arithmetic and Logic Unit
API Application Programming Interface
ASIC Application-specific integrated circuit
BLE Basic Logic Element
CGRA Coarse-Grained Reconfigurable Architecture
CLB Complex Logic Block
CPU Central Processing Unit
DDR Double Data Rate
DFG Data Flow Graph
DMA Direct Memory Access
DPGA Dynamically Programable Gate Array
DPR Dynamic Partial Reconfiguration
DSL Design Specific Language
DSP Digital Signal Processor
ETRR Equal Time Round Robin
FCFS First Come First Serve
FFT Fast Fourier Transform
FPGA Field Programmable Gate Array
GMII Gigabit Media-Independent Interface
GPGPU General Purpose Graphical Porcessing Unit
GPU Graphical Porcessing Unit
HDL Hardware Description Language
HLS High Level Synthesis
IDE Integrated Development Environment
IF Intermediate Fabirc
IP Intelectual Property
IPC Inter-Process Communication

JIT Just In Time
JVM Java Virtual Machine
LUT Lookup Table
MAC Media Access Control
MII Media-Independent Interface
NFS Network File System
NRC Non Recurring Cost
PCI Peripheral Component Interconnect
PLL Phase-Locked Loop
PSL Property Specification Language
RAM Random Access Memory
RGMII Reduced gigabit media-independent interface
RII Réponse Impulsionnelle Infinie
RMII Reduced media-independent interface
ROM Read Only Memory
RTL Register Transfer Level
RTOS Real Time Operating System
SIMD Single Instruction Multiple Data
SLA Service Level Agreement
SPI Serial Peripheral Interface
SPMD Single Program, Multiple Data
VCD Value Change Dump
VHDL VHSIC Hardware Description Language
VHPI VHDL Procedural Interface
VTPR Virtual Time Propagation Register

Bibliographie

[1] Peter Figuli, Michael Hübner, Romuald Girardey, Falco Bapp, Thomas Bruckschlögl, Florian Thoma, Jörg Henkel, and Jürgen Becker. A heterogeneous SoC architecture with embedded virtual FPGA cores and runtime Core Fusion. In *Adaptive Hardware and Systems (Ahs), 2011 Nasa/esa Conference on*, pages 96–103. IEEE, 2011.

[2] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT) : A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7) :1645–1660, 2013.

[3] Saurabh Kumar Garg, Srinivasa K Gopalaiyengar, and Rajkumar Buyya. SLA-based resource provisioning for heterogeneous workloads in a virtualized cloud datacenter. In *International conference on Algorithms and architectures for parallel processing*, pages 371–384. Springer, 2011.

[4] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5) :879–899, 2008.

[5] John E Stone, David Gohara, and Guochun Shi. OpenCL : A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3) :66–73, 2010.

[6] Vignesh T Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 217–228. ACM, 2011.

[7] Michael John Sebastian Smith. *Application-specific integrated circuits*. Addison-Wesley Professional, 2008.

[8] Stephen D Brown, Robert J Francis, Jonathan Rose, and Zvonko G Vranesic. *Field-programmable gate arrays*, volume 180. Springer Science & Business Media, 2012.

[9] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs : From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4) :473–491, 2011.

[10] Wang Lie and Wu Feng-Yan. Dynamic partial reconfiguration in FPGAs. In *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*, volume 2, pages 445–448. IEEE, 2009.

[11] Xilinx. Acceleration in the AWS Cloud. `http://www.xilinx.com/products/design-tools/acceleration-zone/aws.html`.

[12] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[13] Timothy Prickett Morgan. Why Hyperscalers And Clouds Are Pushing Intel Into FPGAs. `http://www.nextplatform.com/2015/07/29/why-hyperscalers-and-clouds-are-pushing-intel-into-fpgas/`, 2015.

[14] William Fornaciari and Vincenzo Piuri. Virtual FPGAs : Some steps behind the physical barriers. *Parallel and Distributed Processing*, pages 7–12, 1998.

[15] William Fornaciari, Vincenzo Piuri, and Luigi Ripamonti. Virtualization of FPGA via Segmentation. In *FPGA*, page 222, 2000.

[16] Harald Simmler, Lorne Levinson, and Reinhard Männer. Multitasking on FPGA coprocessors. In *International Workshop on Field Programmable Logic and Applications*, pages 121–130. Springer, 2000.

[17] Herbert Walder and Marco Platzner. Reconfigurable Hardware Operating Systems : From Design Concepts to Realizations. In *Engineering of Reconfigurable Systems and Algorithms*, pages 284–287, 2003.

[18] Oliver Diessel, Hossam ElGindy, Martin Middendorf, Hartmut Schmeck, and Bernd Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. *IEE Proceedings-Computers and Digital Techniques*, 147(3) :181–188, 2000.

[19] Heiko Kalte and Mario Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 223–228. IEEE, 2005.

[20] Alban Bourge, Olivier Muller, and Frédéric Rousseau. Generating Efficient Context-Switch Capable Circuits through Autonomous Design Flow. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(1) :9, 2016.

[21] Loïc Lagadec, Dominique Lavenier, Erwan Fabiani, and Bernard Pottier. Placing, routing, and editing virtual FPGAs. In *FPL*, volume 1, pages 357–366. Springer, 2001.

[22] Robert Kirchgessner, Greg Stitt, Alan George, and Herman Lam. VirtualRC : a virtual FPGA platform for applications and tools portability. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 205–208. ACM, 2012.

[23] Alexander Brant and Guy GF Lemieux. ZUMA : An open FPGA overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM),*

2012 IEEE 20th Annual International Symposium on, pages 93–96. IEEE, 2012.

[24] James Coole and Greg Stitt. Intermediate fabrics : Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 13–22. ACM, 2010.

[25] Greg Stitt and James Coole. Intermediate fabrics : Virtual architectures for near-instant fpga compilation. *IEEE Embedded Systems Letters*, 3(3) :81–84, 2011.

[26] Ilia Lebedev, Shaoyi Cheng, Austin Doupnik, James Martin, Christopher Fletcher, Daniel Burke, Mingjie Lin, and John Wawrzynek. MARC : A many-core approach to reconfigurable computing. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 7–12. IEEE, 2010.

[27] Damien Picard. *Méthodes et outils logiciels pour l'exploration architecturale d'unités reconfigurables embarquées*. PhD thesis, Université de Bretagne Occidentale, 2010. Thèse de doctorat dirigée par Loïc Lagadec.

[28] Lukáš Sekanina and RI Richard. Design of the Special Fast Reconfigurable Chip Using Common FPGA. In *In : Proc. of the Design and Diagnostic of Electronic Circuits and Systems IEEE DDECS'2000, Polygrafia SAF*. Citeseer, 2000.

[29] André DeHon. DPGA utilization and application. In *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, pages 115–121. ACM, 1996.

[30] Roman L Lysecky, Kris Miller, Frank Vahid, and Kees A Vissers. Firm-core virtual FPGA for just-in-time FPGA compilation. In *FPGA*, page 271, 2005.

[31] Roman Lysecky, Frank Vahid, and Sheldon X-D Tan. Dynamic FPGA routing for just-in-time FPGA compilation. In *Proceedings of the 41st annual Design Automation Conference*, pages 954–959. ACM, 2004.

[32] Alexander Dunlop Brant. *Coarse and fine grain programmable overlay architectures for FPGAs*. PhD thesis, University of British Columbia, 2013.

[33] Dirk Koch, Christian Beckhoff, and Guy GF Lemieux. An efficient FPGA overlay for portable custom instruction set extensions. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8. IEEE, 2013.

[34] Davor Capalija and Tarek S Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8. IEEE, 2013.

[35] Aaron Severance and Guy GF Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–10. IEEE, 2013.

[36] Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. Are coarse-grained overlays ready for general purpose application acceleration on FPGAs? In *Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2016 IEEE 14th Intl C*, pages 586–593. IEEE, 2016.

[37] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. QuickDough : a rapid fpga loop accelerator design framework using soft CGRA overlay. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 56–63. IEEE, 2015.

[38] Sunil Shukla, Neil W Bergmann, and Jürgen Becker. Quku : A two-level reconfigurable architecture. In *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, pages 6–pp. IEEE, 2006.

[39] Abhishek Kumar Jain, Suhaib A Fahmy, and Douglas L Maskell. Efficient Overlay architecture based on DSP blocks. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 25–28. IEEE, 2015.

[40] Lu Hao and Greg Stitt. Virtual finite-state-machine architectures for fast compilation and portability. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 91–94. IEEE, 2013.

[41] Patrick Cooke, Lu Hao, and Greg Stitt. Finite-state-machine overlay architectures for fast FPGA compilation and application portability. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(3) :54, 2015.

[42] James Coole and Greg Stitt. Adjustable-cost overlays for runtime compilation. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 21–24. IEEE, 2015.

[43] Vaughn Betz and Jonathan Rose. VPR : A new packing, placement and routing tool for FPGA research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[44] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for deep-submicron FPGAs*, volume 497. Springer Science & Business Media, 2012.

[45] Vaughn Betz and Jonathan Rose. Automatic generation of FPGA routing architectures from high-level descriptions. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 175–184. ACM, 2000.

[46] Loïc Lagadec. *Abstraction, modélisation et outils de CAO pour les architectures réconfigurables*. PhD thesis, Université de Rennes 1, 2000.

[47] Loïc Lagadec and Bernard Pottier. Object-oriented meta tools for reconfigurable architectures. In *Reconfigurable Technology : FPGAs for Computing and Applications II*, volume 4212, pages 69–80. International Society for Optics and Photonics, 2000.

[48] Loïc Lagadec, Ciprian Teodorov, Jean-Christophe Le Lann, Damien Picard, and Erwan Fabiani. Model-driven toolset for embedded reconfigurable cores : Flexible prototyping and software-like debugging. *Science of Computer Programming*, 96 :156–174, 2014.

[49] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. Coarse-grained reconfigurable array architectures. In *Handbook of signal processing systems*, pages 553–592. Springer, 2013.

[50] Xi Yue. Rapid Overlay Builder for Xilinx FPGAs. 2014.

[51] Tobias Wiersema, Ame Bockhorn, and Marco Platzner. Embedding FPGA overlays into configurable systems-on-chip : ReconOS meets ZUMA. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pages 1–6. IEEE, 2014.

[52] Michael Hubner, Peter Figuli, Romuald Girardey, Dimitrios Soudris, Kostas Siozios, and Jürgen Becker. A heterogeneous multicore system on chip with run-time reconfigurable virtual FPGA architecture. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 143–149. IEEE, 2011.

[53] Andreas Agne, Markus Happe, Ariane Keller, Enno Lubbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. ReconOS : An operating system approach for reconfigurable computing. *IEEE Micro*, 34(1) :60–71, 2014.

[54] Enno Lübbers and Marco Platzner. ReconOS : Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1) :8, 2009.

[55] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, page 3. ACM, 2014.

[56] OpenStack is open source software for creating private and public clouds. `https://www.openstack.org/`.

[57] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. Fpgas in the cloud : Booting virtualized hardware accelerators with openstack. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 109–116. IEEE, 2014.

[58] Théotime Bollengier, Loïc Lagadec, Jean-Christophe Le Lann, and Pierre Guilloux. Soft Timing Closure for Soft Programmable Logic Cores : The ARGen Approach. In *Applied Reconfigurable Computing : 13th International Symposium, ARC 2017, Delft, The Netherlands, April 3-7, 2017, Proceedings*, volume 10216, page 93. Springer, 2017.

[59] Mike Gordon, Joe Hurd, and Konrad Slind. Executing the formal semantics of the accellera property specification language by mechanised theorem proving. In *CHARME*, pages 200–215. Springer, 2003.

[60] Marc Boulé and Zeljko Zilic. Efficient automata-based assertion-checker synthesis of PSL properties. In *High-Level Design Validation and Test Workshop, 2006. Eleventh Annual IEEE International*, pages 69–76. IEEE, 2006.

[61] Specification for the WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Revision : B.3. http://cdn.opencores.org/downloads/wbspec_b3.pdf, 2002.

[62] Armadeus. L'APF6_SP pour les systèmes Linux embarqués. http://www.opossom.com/francais/produits-cartes_microprocesseur-apf6_sp.html.

[63] Loïc Lagadec, Jean-Christophe Le Lann, and Theotime Bollengier. A prototyping platform for virtual reconfigurable units. In *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip, ReCoSoC 2014, Montpellier, France, May 26-28, 2014*, pages 1–7, 2014.

[64] Texas Instruments. AN-1469 PHYTER Design & Layout Guide. http://www.ti.com/lit/an/snla079d/snla079d.pdf, 2006.

[65] OpenCores. http://opencores.org/.

[66] Zylin. Zylin CPU. http://opensource.zylin.com/zpuref.html, 2008.

[67] Motorola Inc. SPI Block Guide V03.06. https://web.archive.org/web/20150413003534/http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf, 2000.

[68] Sebastien Bourdeauducq. Minimac – the minimalist Ethernet MAC. https://opencores.org/project,minimac, 2010.

[69] Real Time Engineers Ltd. FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. http://www.freertos.org/, Consulté en 2016.

[70] ChaN. FatFs – Generic FAT File System Module. http://elm-chan.org/fsw/ff/00index_e.html, Consulté en 2017.

[71] Adam Dunkels. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science*, 2 :77, 2001.

[72] Altera. Avalon Interface Specifications. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf.

[73] Tristan Gingold. GHDL. *A VHDL compiler*, 2007.

[74] Francoise Martinolle and Adam Sherer. A procedural language interface for VHDL and its typical applications. In *Verilog HDL Conference and VHDL International Users Forum, 1998. IVC/VIUF. Proceedings., 1998 International*, pages 32–38. IEEE, 1998.

[75] Christian Pilato and Fabrizio Ferrandi. Bambu : A Free Framework for the High-Level Synthesis of Complex Applications. *University Booth of DATE*, 2012.

[76] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp : high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[77] Berkeley Logic Interchange Format (BLIF). `http://vlsi.colorado.edu/~vis/blif.ps`, 1992. University of California Berkeley.

[78] EDIF Steering Committee et al. EDIF Electronic Design Interchange Format Version 2 0 0. *Electronic Industries Association*, 1987.

[79] Peter Jamieson, Kenneth B Kent, Farnaz Gharibian, and Lesley Shannon. Odin ii-an open-source verilog hdl synthesis tool for cad research. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 149–156. IEEE, 2010.

[80] Peter Bellows and Brad Hutchings. JHDL-an HDL for reconfigurable systems. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 175–184. IEEE, 1998.

[81] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, and Matthew French. Torc : towards an open-source tool flow. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 41–44. ACM, 2011.

[82] Christian Beckhoff, Dirk Koch, and Jim Torresen. The xilinx design language (xdl) : Tutorial and use cases. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pages 1–8. IEEE, 2011.

[83] Berkeley Logic Ssynthesis. Verification Group. *ABC : A system for sequential synthesis and verification*, 2013.

[84] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-aware AIG rewriting : A fresh look at combinational logic synthesis. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 532–535. IEEE, 2006.

[85] Robert Brayton and Alan Mishchenko. ABC : An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

[86] Ellen M Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R Stephan, Robert K Brayton, and Alberto Sangiovanni-Vincentelli. SIS : A system for sequential circuit synthesis. 1992.

[87] Narendra Shenoy and Richard Rudell. Efficient implementation of retiming. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 226–233. IEEE Computer Society Press, 1994.

[88] Mentor Graphics. ModelSim ASIC and FPGA Design. `https://www.mentor.com/products/fv/modelsim/`.

[89] Stephen Williams. Icarus Verilog. `http://iverilog.icarus.com/`.

[90] IEEE Design Automation Sub-Committee et al. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language. 1996.

[91] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B Kent, Peter Jamieson, and Jason Anderson. The VTR project : architecture and CAD for FPGAs from verilog to routing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 77–86. ACM, 2012.

[92] Lukas PPP Van Ginneken. Buffer placement in distributed RC-tree networks for minimal Elmore delay. In *Circuits and Systems, 1990., IEEE International Symposium on*, pages 865–868. IEEE, 1990.

[93] Steven JE Wilton, Noha Kafafi, James CH Wu, Kimberly A Bozman, Victor O Aken'Ova, and Resve Saleh. Design considerations for soft embedded programmable logic cores. *IEEE Journal of Solid-State Circuits*, 40(2) :485–497, 2005.

[94] VC Aken'Ova, Guy Lemieux, and Resve Saleh. An improved" soft" eFPGA design and implementation strategy. In *Custom Integrated Circuits Conference, 2005. Proceedings of the IEEE 2005*, pages 179–182. IEEE, 2005.

[95] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. Post-placement C-slow retiming for the Xilinx Virtex FPGA. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 185–194. ACM, 2003.

[96] Altera. AN715 Hyper-Pipelining for Stratix 10 Designs. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/an/an715.pdf`, 2015.

[97] Ken Eguro and Scott Hauck. Armada : timing-driven pipeline-aware routing for FPGAs. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 169–178. ACM, 2006.

[98] Larry McMurchie and Carl Ebeling. PathFinder : a negotiation-based performance-driven router for FPGAs. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 111–117. ACM, 1995.

[99] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2) :203–215, 2007.

[100] Peter Mell and Tim Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology*, 53(6) :50, 2009.

[101] Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. Service level agreement in cloud computing. 2009.

[102] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat T. Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8-11, 2011*, pages 217–228, 2011.

[103] Shuai Che, Jie Li, Jeremy W Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with GPUs and FPGAs. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107. IEEE, 2008.

[104] Anil Madhavapeddy and Satnam Singh. Reconfigurable data processing for clouds. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 141–145. IEEE, 2011.

[105] A Putnam, A M Caulfield, E S Chung, D Chiou, K Constantinides, J Demme, H Esmaeilzadeh, J Fowers, G P Gopal, J Gray, M Haselman, S Hauck, S Heil, A Hormati, J Y Kim, S Lanka, J Larus, E Peterson, S Pope, A Smith, J Thong, P Y Xiao, and D Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, 2014.

[106] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of FPGA, GPU and CPU in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 126–131. IEEE, 2009.

[107] Jed Scaramella, Matthew Marden, John Daly, and Randy Perry. The Cost of Retaining Aging IT Infrastructure. `http://www.lenovo.com/images/products/server/pdfs/whitepapers/IDC%20Whitepaper%20246755.pdf`, 2014 (consulté : 2017-02-27).

[108] Tiago C Ferreto, Marco AS Netto, Rodrigo N Calheiros, and César AF De Rose. Server consolidation with migration control for virtualized data centers. *Future Generation Computer Systems*, 27(8) :1027–1034, 2011.

[109] Nidhi Jain Kansal and Inderveer Chana. Cloud load balancing techniques : A step towards green computing. *IJCSI International Journal of Computer Science Issues*, 9(1) :238–246, 2012.

[110] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. Virtualized FPGA accelerators for efficient cloud computing. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 430–435. IEEE, 2015.

[111] Jagath Weerasinghe, Raphael Polig, Francois Abel, and Christoph Hagleitner. Network-Attached FPGAs for Data Center Applications. 2016.

[112] Alban Bourge. *Changement de contexte matériel sur FPGA entre équipements reconfigurables et hétérogènes dans un envirtonnement de calcul distribué*. PhD thesis, Communauté Université Grenoble Alpes, 2016. Thèse de doctorat dirigée par Frédérique Rousseau et co-encadrée par Olivier Muller.

[113] Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. Throughput oriented FPGA overlays using DSP blocks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1628–1633. IEEE, 2016.

[114] Mohamad Najem, Théotime Bollengier, Jean-Christophe Le Lann, and Loïc Lagadec. Extended Overlay Architectures For Heterogeneous FPGA Cluster Management. *Journal of Systems Architecture*, 2017.

[115] Mohamad Najem, Théotime Bollengier, Jean-Christophe Le Lann, and Loïc Lagadec. A cost-effective approach for efficient time-sharing of reconfigurable architectures. In *FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC), 2017 International Conference on*, pages 7–12. IEEE, 2017.

[116] Théotime Bollengier, Mohamad Najem, Jean-Christophe Le Lann, and Loïc Lagadec. Demo : Overlay architectures for heterogeneous FPGA cluster ma-

nagement. In *Design and Architectures for Signal and Image Processing (DASIP), 2016 Conference on*, pages 239–240. IEEE, 2016.

[117] Théotime Bollengier, Mohamad Najem, Jean-Christophe Le Lann, and Loïc Lagadec. Zeff : Une plateforme pour l'intégration d'architectures overlay dans le cloud. In *COMPAS 2016*, 2016.

[118] Théotime Bollengier, Mohamad Najem, Jean-Christophe Le Lann, and Loïc Lagadec. Overlay architectures for fpga resource virtualization. In *GDR SOC SIP*, 2016.

[119] Mohamad NAJEM, Théotime BOLLENGIER, LE Jean-Christophe, and Loıc LAGADEC. Fpgas in the cloud : a hybrid hadrware/software framework. 2016.

[120] David Dickin and Lesley Shannon. Exploring FPGA technology mapping for fracturable LUT minimization. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–8. IEEE, 2011.

Annexes

Annexe A

Code de description d'architecture vFPGA1

```
(architecture "titearchi"
  (withConfigPreloading false)
  (withSnapshotRegister true)
  (withVTPR true)
  (routeChannelWidth 4)
  (clbs
    (clb "clb"
      (N 4)
      (I 12)
      (K 4)
      (clbRouting
        (interface "dflt"
          (inputs
            (input 0  top    (connectTo 0))
            (input 4  top    (connectTo 1))
            (input 8  top    (connectTo 2))
            (input 1  right  (connectTo 0))
            (input 5  right  (connectTo 1))
            (input 9  right  (connectTo 3))
            (input 2  bottom (connectTo 0))
            (input 6  bottom (connectTo 2))
            (input 10 bottom (connectTo 3))
            (input 3  left   (connectTo 1))
            (input 7  left   (connectTo 2))
            (input 11 left   (connectTo 3))
          )
          (outputs
            (output 0 top    (connectTo 0 1))
            (output 1 right  (connectTo 0 1))
            (output 2 bottom (connectTo 2 3))
            (output 3 left   (connectTo 2 3))
          )
        )
        (interface "left"
          (inputs
            (input 0  top    (connectTo 0))
            (input 4  top    (connectTo 1))
```

```
            (input 8  top    (connectTo 2))
            (input 1  right  (connectTo 0))
            (input 5  right  (connectTo 1))
            (input 9  right  (connectTo 3))
            (input 2  bottom (connectTo 0))
            (input 6  bottom (connectTo 2))
            (input 10 bottom (connectTo 3))
            (input 3  left   (connectTo 0))
            (input 7  left   (connectTo 2))
            (input 11 left   (connectTo 3))
          )
          (outputs
            (output 0 top    (connectTo 0 1))
            (output 1 right  (connectTo 0 1))
            (output 2 bottom (connectTo 2 3))
            (output 3 left   (connectTo 1 2))
          )
        )
        (interface "bot"
          (inputs
            (input 0  top    (connectTo 0))
            (input 4  top    (connectTo 1))
            (input 8  top    (connectTo 2))
            (input 1  right  (connectTo 0))
            (input 5  right  (connectTo 1))
            (input 9  right  (connectTo 3))
            (input 2  bottom (connectTo 1))
            (input 6  bottom (connectTo 2))
            (input 10 bottom (connectTo 3))
            (input 3  left   (connectTo 1))
            (input 7  left   (connectTo 2))
            (input 11 left   (connectTo 3))
          )
          (outputs
            (output 0 top    (connectTo 0 1))
            (output 1 right  (connectTo 0 1))
            (output 2 bottom (connectTo 1 2))
            (output 3 left   (connectTo 2 3))
          )
        )
        (interface "botleft"
          (inputs
            (input 0  top    (connectTo 0))
            (input 4  top    (connectTo 1))
            (input 8  top    (connectTo 2))
            (input 1  right  (connectTo 0))
            (input 5  right  (connectTo 1))
            (input 9  right  (connectTo 3))
            (input 2  bottom (connectTo 1))
            (input 6  bottom (connectTo 2))
            (input 10 bottom (connectTo 3))
            (input 3  left   (connectTo 0))
            (input 7  left   (connectTo 2))
            (input 11 left   (connectTo 3))
          )
          (outputs
            (output 0 top    (connectTo 0 1))
```

```
95              (output 1 right  (connectTo 0 1))
                (output 2 bottom (connectTo 1 2))
97              (output 3 left   (connectTo 1 2))
              )
99          )
        )
101   )
  )
103   (ioBlocks
    (ioBlock "topIO" 1
105     (ios
        (io 0 (inputConnectTo 2 3) (outputConnectTo 2 3))
107     )
    )
109   (ioBlock "rgtIO" 1
      (ios
111     (io 0 (inputConnectTo 0 2) (outputConnectTo 2 3))
      )
113   )
    (ioBlock "botIO" 1
115     (ios
        (io 0 (inputConnectTo 0 2) (outputConnectTo 0 3))
117     )
    )
119   (ioBlock "lftIO" 1
      (ios
121     (io 0 (inputConnectTo 0 1) (outputConnectTo 0 3))
      )
123   )
  )
125   (layout (size 3 3)
              "topIO"      "topIO"    "topIO"
127     "lftIO"  "clb.left"   "clb.dflt" "clb.dflt" "rgtIO"
      "lftIO"  "clb.left"   "clb.dflt" "clb.dflt" "rgtIO"
129     "lftIO" "clb.botleft" "clb.bot"  "clb.bot"  "rgtIO"
              "botIO"      "botIO"    "botIO"
131   )
  (switchBoxes
133     (switchBox "top"
        (mux (track left   0) (drivenBy (track right 0) (track bottom 0)))
135     (mux (track left   1) (drivenBy (track right 1) (track bottom 1)))
        (mux (track left   2) (drivenBy (track right 2) (track bottom 2)))
137     (mux (track left   3) (drivenBy (track right 3) (track bottom 3)))
        (mux (track right  0) (drivenBy (track left  0) (track bottom 0)))
139     (mux (track right  1) (drivenBy (track left  1) (track bottom 1)))
        (mux (track right  2) (drivenBy (track left  2) (track bottom 2)))
141     (mux (track right  3) (drivenBy (track left  3) (track bottom 3)))
        (mux (track bottom 0) (drivenBy (track right 0) (track left   0)))
143     (mux (track bottom 1) (drivenBy (track right 1) (track left   1)))
        (mux (track bottom 2) (drivenBy (track right 2) (track left   2)))
145     (mux (track bottom 3) (drivenBy (track right 3) (track left   3)))
    )
147   (switchBox "left"
        (mux (track bottom 0) (drivenBy (track top    0) (track right  0)))
149     (mux (track bottom 1) (drivenBy (track top    1) (track right  1)))
        (mux (track bottom 2) (drivenBy (track top    2) (track right  2)))
151     (mux (track bottom 3) (drivenBy (track top    3) (track right  3)))
```

```
      (mux (track top    0) (drivenBy (track bottom 0) (track right  0)))
      (mux (track top    1) (drivenBy (track bottom 1) (track right  1)))
      (mux (track top    2) (drivenBy (track bottom 2) (track right  2)))
      (mux (track top    3) (drivenBy (track bottom 3) (track right  3)))
      (mux (track right  0) (drivenBy (track top    0) (track bottom 0)))
      (mux (track right  1) (drivenBy (track top    1) (track bottom 1)))
      (mux (track right  2) (drivenBy (track top    2) (track bottom 2)))
      (mux (track right  3) (drivenBy (track top    3) (track bottom 3)))
    )
    (switchBox "right"
      (mux (track top    0) (drivenBy (track bottom 0) (track left 0)))
      (mux (track top    1) (drivenBy (track bottom 1) (track left 1)))
      (mux (track top    2) (drivenBy (track bottom 2) (track left 2)))
      (mux (track top    3) (drivenBy (track bottom 3) (track left 3)))
      (mux (track bottom 0) (drivenBy (track top    0) (track left 0)))
      (mux (track bottom 1) (drivenBy (track top    1) (track left 1)))
      (mux (track bottom 2) (drivenBy (track top    2) (track left 2)))
      (mux (track bottom 3) (drivenBy (track top    3) (track left 3)))
      (mux (track left   0) (drivenBy (track bottom 0) (track top  0)))
      (mux (track left   1) (drivenBy (track bottom 1) (track top  1)))
      (mux (track left   2) (drivenBy (track bottom 2) (track top  2)))
      (mux (track left   3) (drivenBy (track bottom 3) (track top  3)))
    )
    (switchBox "bot"
      (mux (track right 0) (drivenBy (track left  0) (track top  0)))
      (mux (track right 1) (drivenBy (track left  1) (track top  1)))
      (mux (track right 2) (drivenBy (track left  2) (track top  2)))
      (mux (track right 3) (drivenBy (track left  3) (track top  3)))
      (mux (track left  0) (drivenBy (track right 0) (track top  0)))
      (mux (track left  1) (drivenBy (track right 1) (track top  1)))
      (mux (track left  2) (drivenBy (track right 2) (track top  2)))
      (mux (track left  3) (drivenBy (track right 3) (track top  3)))
      (mux (track top   0) (drivenBy (track left  0) (track right 0)))
      (mux (track top   1) (drivenBy (track left  1) (track right 1)))
      (mux (track top   2) (drivenBy (track left  2) (track right 2)))
      (mux (track top   3) (drivenBy (track left  3) (track right 3)))
    )
  )
  (switchBoxLayout
    wilton "top"  "top"   wilton
    "left" wilton wilton "right"
    "left" wilton wilton "right"
    wilton "bot"  "bot"   wilton
  )
)
```

Listing A.1 – Exemple de description d'architecture vFPGA1

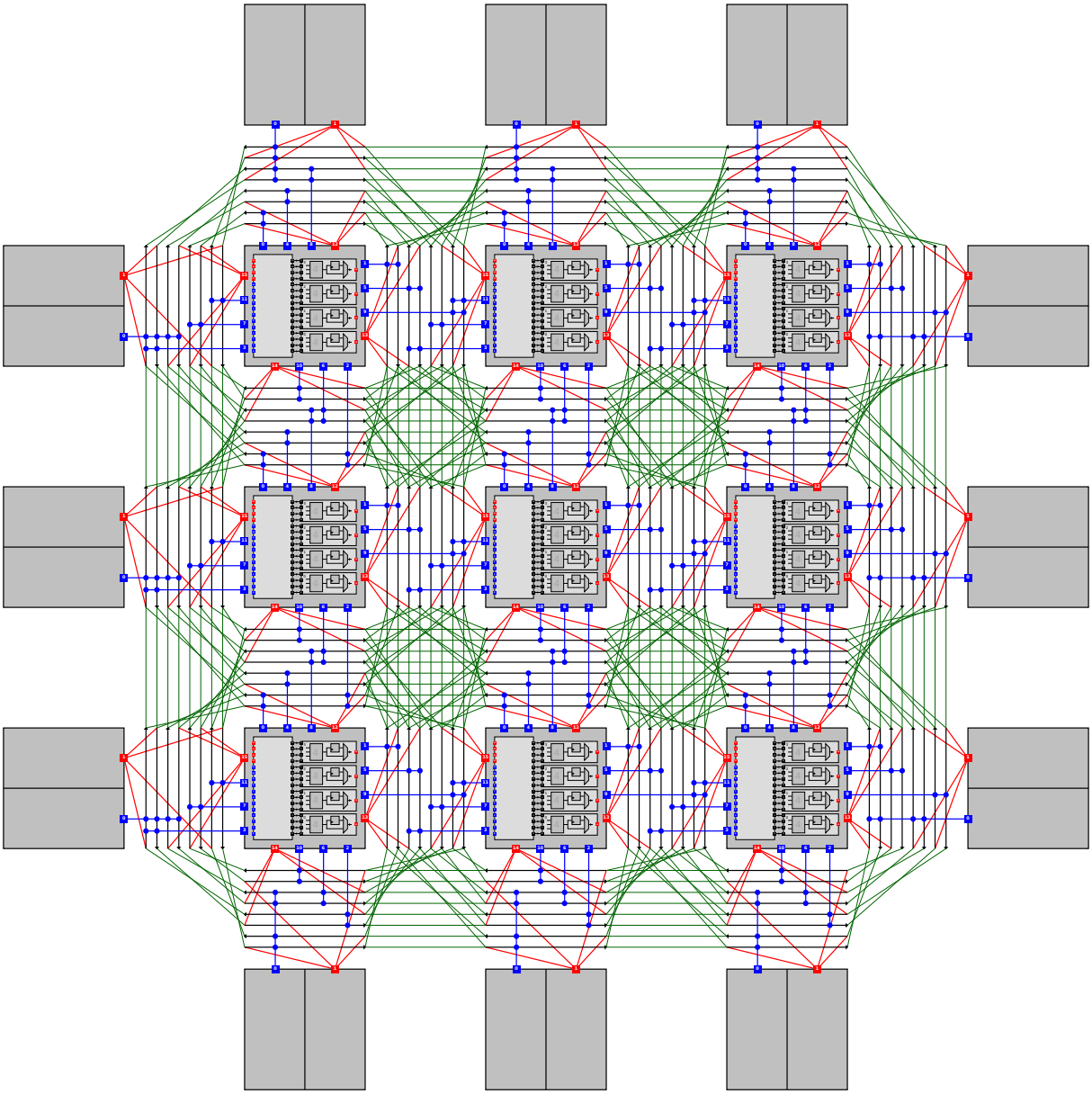


FIGURE A.1 – Représentation du plan de calcul de l'architecture générée à partir du code du listing A.1

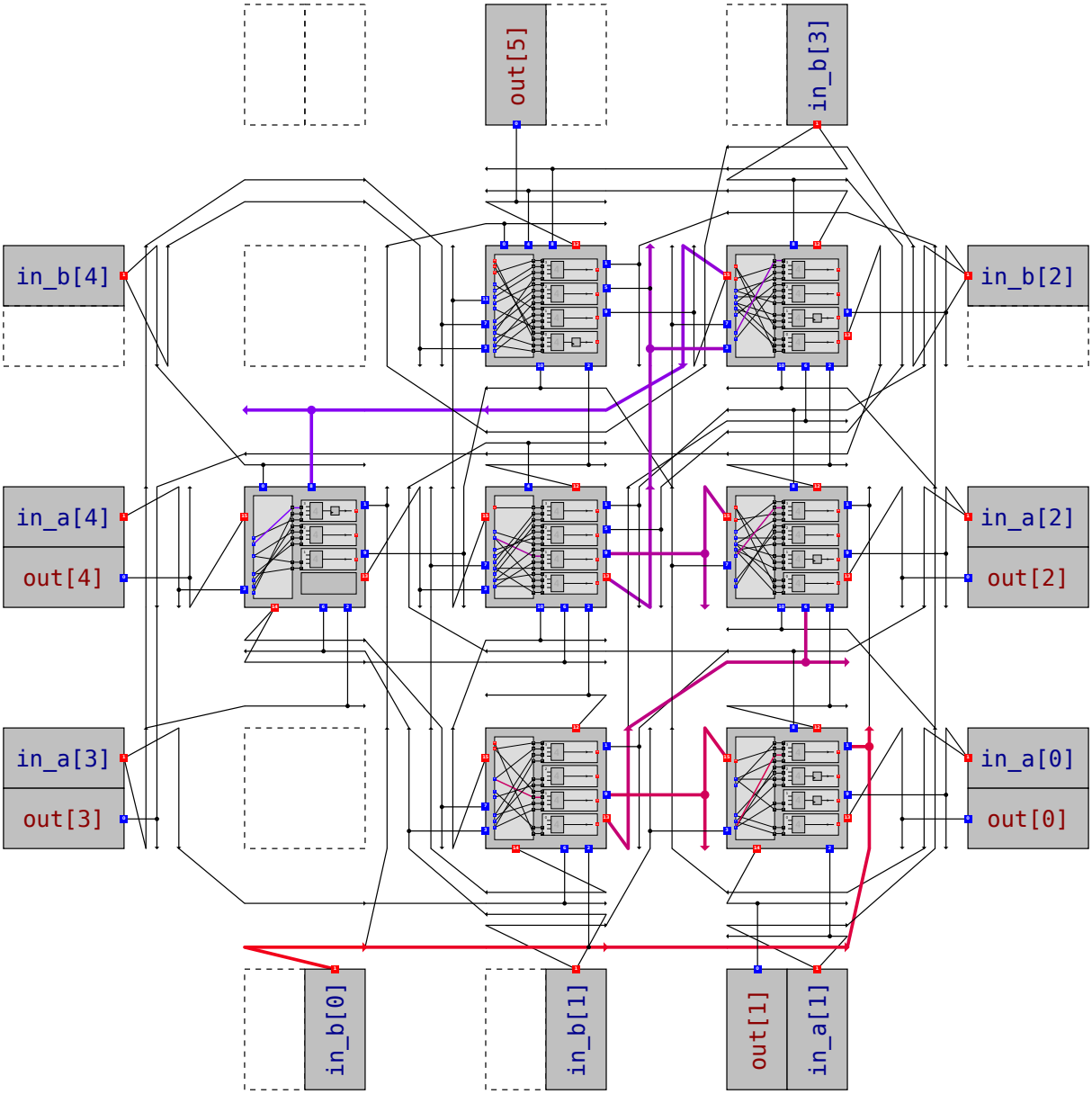


FIGURE A.2 – Placement et routage d'un multiplieur $5 \times 5 \rightarrow 6$ bits sur l'architecture de la figure A.1. Le chemin critique est surligné en gras, il traverse 19 VTPRs.

Annexe B

Appels systèmes de l'hyperviseur local

Les appels systèmes relatifs à l'overlay se fient aux trois registres de présentation de l'IP pour déterminer les caractéristiques de l'overlay. Ces registres comprennent des informations relatives au plan de calcul, permettant ainsi de déterminer si un bitstream virtuel cible bien l'overlay géré par l'hyperviseur :

- la largeur et la hauteur de la matrice de CLBs ;
- le nombre de pistes de routage par canal de routage ;
- le nombre de BLEs par CLB ;
- le nombre d'entrées par CLB ;
- le nombre d'entrées par LUT ;
- le nombre d'IOs virtuelles de l'overlay.

Ces registres comprennent aussi des informations relatives au plan de configuration et de snapshot qui permettent de déterminer si un bitstream a une taille valide et combien de mots doivent être lus pour extraire un snapshot valide :

- la taille du registre de configuration en bits (configuration de la matrice + configuration de réorganisation des IOs virtuelles) ;
- la taille du registre de snapshot en bits ;

Les registres de présentation comprennent aussi des informations sur les capacités des contrôleurs de l'IP :

- si l'IP overlay intègre ou non un contrôleur DMA ;
- si l'IP overlay intègre ou non un contrôleur de mémoire virtuelle ;
- si l'IP overlay permet ou non à la matrice de générer un signal d'interruption via une IO virtuelle ;
- la taille d'un mot DMA en bits ;
- la taille d'un mot de donnée de la mémoire virtuelle en bits ;
- la taille du mot d'adresse de la mémoire virtuelle en bits.

Pour qu'une application puisse être acceptée par un nœud de calcul, il faut donc que les ressources utilisées par l'application soient annoncées dans les registres de présentation. Aussi, certains appels systèmes sont valides ou non suivant les capacités de l'IP annoncée dans ces registres.

Ces appels systèmes permettent de lire le statu de l'IP overlay, de réaliser son contrôle et sa configuration bas niveau via des opérations atomiques :

- pousser un bitstream virtuel dans le plan de configuration de l'overlay depuis un fichier ou un buffer mémoire ;
- charger ou sauvegarder le contenu du plan de snapshot de l'overlay depuis/-vers un fichier ou un buffer mémoire ;
- sauvegarder l'état des registres applicatifs dans le plan de snapshot, ou restaurer l'état des registres applicatifs depuis le plan de snapshot ;
- configurer le diviseur de l'horloge physique pour configurer la fréquence de l'horloge applicative ;
- stopper, activer indéfiniment ou pour un certain nombre de cycles l'horloge applicative ;
- activer ou désactiver la DMA ;
- activer ou désactiver la mémoire virtuelle ;
- configurer l'adresse de l'espace mémoire utilisée pour stocker le contenu de la mémoire virtuelle ;
- configurer les adresses de début et de fin des buffers DMA de données d'entrées et de sorties ;
- lire l'état de consommation/remplissage des buffers DMA, le nombre de cycles d'horloge applicative écoulés ;
- masquer ou démasquer les interruptions générées par la matrice, la fin d'un buffer DMA d'entrée ou de sortie, ou l'achèvement du nombre de cycles d'horloge applicative voulu ;
- d'activer ou désactiver le reset des registres applicatifs de la matrice.

La gestion haut niveau de l'overlay est automatisée par l'ordonnanceur qui séquence les appels à ces fonctions.

Résumé

De part leur capacité de reconfiguration et les performances qu'ils offrent, les FPGAs sont de bons candidats pour accélérer des applications dans le Cloud. Cependant, les FPGAs présentent certaines caractéristiques qui font obstacle à leur utilisation dans le Cloud et leur adoption par les clients : premièrement, la programmation des FPGAs se fait à bas niveau et demande une certaine expertise, que n'ont pas nécessairement les clients habituels du Cloud. Deuxièmement, les FPGAs ne présentent pas de mécanismes natifs permettant leur intégration dans le modèle de gestion dynamique d'une infrastructure Cloud.

Dans ce travail, nous proposons d'utiliser des architectures overlay afin de faciliter l'adoption, l'intégration et l'exploitation de FPGAs dans le Cloud. Les overlays sont des architectures reconfigurables elles-mêmes implémentée sur FPGA. En tant que couche d'abstraction matérielle placée entre le FPGA et les applications, les overlays permettent de monter le niveau d'abstraction du modèle d'exécution présenté aux applications et aux utilisateurs, ainsi que d'implémenter des mécanismes facilitant leur intégration et leur exploitation dans une infrastructure Cloud.

Ce travail présente une approche verticale adressant tous les aspects de la mise en œuvre d'overlays dans le Cloud en tant qu'accélérateurs reconfigurables par les clients : de la conception et l'implémentation des overlays, leur intégration sur des plateformes FPGA commerciales, la mise en place de leurs mécanismes d'exploitation, jusqu'à la réalisation de leurs outils de programmation. L'environnement réalisé est complet, modulaire et extensible, il repose en partie sur différents outils existants, et démontre la faisabilité de notre approche.

Mots clés : Architecture reconfigurable, Overlay FPGA, Virtualisation matérielle, Compatibilité des bitstreams, Migration de tâches matérielles, Ordonnancement de tâches matérielles

Abstract

Due to their reconfigurable capability and the performance they offer, FPGAs are good candidates for accelerating applications in the cloud. However, FPGAs have some features that hinder their use in the Cloud as well as their adoption by customers : first, FPGA programming is done at low level and requires some expertise that usual Cloud clients do not necessarily have. Secondly, FPGAs do not have native mechanisms allowing them to easily fit in the dynamic execution model of the Cloud.

In this work, we propose to use overlay architectures to facilitate FPGA adoption, integration, and operation in the Cloud. Overlays are reconfigurable architectures synthesized on FPGA. As hardware abstraction layers placed between the FPGA and applications, overlays allow to raise the abstraction level of the execution model presented to applications and users, as well as to implement mechanisms making them fit in a Cloud infrastructure.

This work presents a vertical approach addressing all aspects of overlay operation in the Cloud as reconfigurable accelerators programmable by tenants : from designing and implementing overlays, integrating them on commercial FPGA platforms, setting up their operating mechanisms, to developping their programming tools. The environment developped in this work is complete, modular and extensible, it is partially based on several existing tools, and demonstrate the feasibility of our approach.

Keywords : Reconfigurable architecture, FPGA overlay, Hardware virtualization, Bitstream compatibility, Hardware task migration, Hardware task scheduling