Changement de contexte matériel sur FPGA, entre équipements reconfigurables et hétérogènes dans un environnement de calcul distribué

Alban Bourge

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Nano Électronique, Nano Technologies**

Arrêté ministériel : 25 mai 2016

Présentée par

Alban BOURGE

Thèse dirigée par **Frédéric ROUSSEAU**
et co-encadrée par **Olivier MULLER**

Préparée au sein du **Laboratoire TIMA, UGA / CNRS**
dans **l'École Doctorale Électronique, Électrotechnique, Automatique et Traitement du Signal (EEATS)**

Changement de contexte matériel sur FPGA entre équipements reconfigurables et hétérogènes dans un environnement de calcul distribué.

Thèse soutenue publiquement le **23 novembre 2016**,
devant le jury composé de :

M. Michel AUGUIN
Directeur de Recherche, LEAT, Examinateur

M. Loïc LAGADEC
Professeur, ENSTA Bretagne, Rapporteur

M. Sébastien PILLEMENT
Professeur, Université de Nantes, Rapporteur

M. Olivier SENTIEYS
Directeur de Recherche, INRIA, Examinateur

M. Frédéric ROUSSEAU
Professeur, Université de Grenoble Alpes, Directeur de thèse

M. Olivier MULLER
Maître de Conférences, Grenoble INP, Co-encadrant de thèse

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Un système informatique au sens classique du terme peut regrouper aussi bien les ordinateurs que les systèmes embarqués ou encore les robots. Tous ces systèmes, aussi différents soient-ils, ont un cœur de fonctionnement de nature identique, à savoir une unité centrale de traitement (CPU), une mémoire et un ou plusieurs actionneurs réalisant la fonctionnalité voulue. Dans le cas d'un ordinateur, les actionneurs sont des périphériques tels que le clavier, la souris et l'écran, réalisant les fonctionnalités de saisie et d'affichage d'information. D'autres périphériques existent dans le monde informatique, servant à des tâches spécifiques et potentiellement exigeantes en terme de ressources de calcul. Par exemple, il est courant d'avoir un périphérique s'occupant exclusivement de calculs liés à l'affichage graphique, car ce type de calcul occuperait beaucoup trop souvent les ressources du CPU. Ces périphériques sont appelés « accélérateurs matériels » dans le sens où ils libèrent le cœur du système informatique d'une tâche à accomplir et généralement, l'accélèrent car ils sont spécifiquement calibrés pour cette tâche précise. Pour ce faire, la tâche est déportée sur l'accélérateur matériel qui prend en charge son exécution. Les circuits logiques reconfigurables sont une cible très intéressante pour de telles applications. Pourtant, ces derniers sont sous utilisés au vu des avantages qu'ils proposent.

Circuits logiques reconfigurables

Les circuits logiques reconfigurables font partie de la famille des systèmes reconfigurables [Poc+13 ; Lyk+15]. Parmi les différentes technologies existantes émerge plus particulièrement le FPGA, pour *Field Programmable Gate Array*. Celui-ci est utilisé comme support de développement pour des applications très diverses [Tes+15] car il est possible de mettre en œuvre n'importe quel circuit puis de reconfigurer la puce afin de changer complètement le type d'application exécutée. De nombreux ouvrages traitent exclusivement de ce type de support, dont notamment [Hau+10].

Avantages des circuits reconfigurables

Les circuits reconfigurables possèdent un avantage très conséquent comparé aux circuits « classiques ». Ces derniers ont une architecture arrêtée et un mode de fonctionnement défini à sa création alors qu'un circuit reconfigurable est programmable donc son comportement est non fixé et à définir. Il est possible de traiter une diversité d'applications considérable ne serait-ce qu'avec une seule puce reconfigurable, pourvu qu'elle soit dotée des canaux de communication adéquats. La flexibilité d'un FPGA est telle que des applications comme des circuits évolutifs à chaud sont envisageables [Can+12]. En outre, il est souvent plus avantageux en terme de consommation ou de vitesse de réaliser un accélérateur ad-hoc sur cible reconfigurable plutôt qu'un logiciel. Par exemple, [Kin+06] montre qu'il est possible d'accélérer d'un facteur trois des applications fonctionnant dans des supercalculateurs en les transposant dans des FPGA , et ce même pour des logiciels fortement optimisés. D'ailleurs, [EG+08] invoque le passage de la HPC à l'HPRC (*High Performance Computing* vers *High-Performance Reconfigurable Computing*) afin d'assoir l'idée que la puce reconfigurable pourrait prendre la place du processeur pour le calcul de haute performance ou le calcul distribué.

Limitations des circuits reconfigurables

Malgré tous ses avantages, le FPGA n'est pourtant pas encore dans une position dominante ni sur le marché de l'accélération matérielle, ni sur celui du calcul haute performance. La capacité de reconfiguration ne vient bien sur pas sans cout. Dans l'état actuel des choses, développer une application pour une cible reconfigurable demande des solides compétences et un temps considérable. Les choses s'améliorent grâce à des techniques comme la synthèse de haut niveau, mais il reste nettement plus abordable de développer un logiciel, celui-ci bénéficiant des infrastructures telles qu'un système d'exploitation, plutôt qu'un accélérateur matériel. Intégrer la puce reconfigurable au sein de systèmes préexistants (processeur additionné d'un système d'exploitation, mémoires, etc.) est aussi chronophage. En somme, une certaine flexibilité d'utilisation des FPGA reste manquante.

Propositions

Afin d'améliorer la flexibilité des tâches matérielles, c'est-à-dire celles s'exécutant sur un FPGA, on souhaite pouvoir appliquer une méthode de changement de contexte. Une telle méthode, éprouvée dans le domaine du processeur, serait bénéfique dans le cadre des FPGA. Dans ce manuscrit, on propose un flot de conception ayant pour objectif principal de créer des circuits automatiquement dotés de la capacité de commutation et cela sans effort de la part du développeur d'application.

Plan du mémoire

Dans le premier chapitre de ce mémoire, on aborde de manière détaillée les raisons qui motivent ce travail ainsi que les problèmes identifiés qui restent à résoudre. Dans le chapitre 3, une analyse de l'état de l'art permet de mettre en perspective les travaux qui sont présentés dans la suite du manuscrit. La présentation globale dans le chapitre 4 du flot de conception imaginé permet ensuite d'aborder les différentes contributions soumises. Les deux chapitres suivants traitent des deux étapes distinctes du flot de conception proposé. Enfin, dans un dernier chapitre, on présente les résultats des expériences menées sur des circuits bâtis à l'aide du flot de conception proposé.

Chapitre 2

Motivations et problématiques

Ce chapitre présente les motivations et les problématiques liées à ce travail. À partir des observations et des questionnements posés en introduction, il reconstruit les questions auxquelles il faudra répondre tout au long de ce manuscrit.

Dans un premier temps, on aborde la notion d'accélération matérielle, son intérêt dans les systèmes modernes et la place qu'occupent les FPGA dans ce domaine. Au niveau d'un système, la gestion des accélérateurs matériels pose de nombreuses questions. Les FPGA, famille d'accélérateurs à part entière, ne se satisfont actuellement pas des réponses déjà apportées par la littérature pour les technologies plus matures. C'est le cas notamment pour les mécanismes dit de « changement de contexte ». Cette technique potentiellement bénéfique est rendue complexe par la nature hétérogène des ressources reconfigurables.

Sommaire

2.1 Accélération matérielle

2.1.1 Définition, intérêt et nature

Les accélérateurs matériels ont pour objectif principal de suppléer aux déficits des unités centrales de traitement, ou CPU, en terme de performances et de consommation. Ce sont des unités de traitement spécifiques, distribuées dans le système qui sont commandées par le CPU afin d'obtenir un comportement souhaité par l'utilisateur du système. Cela peut être l'affichage d'une vidéo en haute définition, le décodage d'une image dans un format particulier, la décapsulation d'un paquet reçu par le biais d'un réseau informatique... Toutes ces actions sont généralement couteuses pour le CPU en terme de temps (nombre d'instructions pour réaliser l'action) ou en terme de consommation (fonctionnement du processeur à une fréquence donnée pendant le nombre d'instructions requises). Les accélérateurs matériels sont de nature diverse, conformément au vaste panel d'actions qu'ils peuvent réaliser. Les plus répandus sont les processeurs graphiques, ou puces d'accélération graphique (en anglais *Graphical Processing Unit* ou GPU), qui ont pour rôle historique la prise en charge des calculs liés à l'affichage. Leur utilisation a depuis été étendue à d'autres tâches notamment le calcul scientifique [Owe+08]. On compte aussi les processeurs de traitement numérique du signal (*Digital Signal Processing* ou DSP). Ils sont spécialisés dans des tâches calculatoires sur signaux numériques comme le filtrage, la mesure ou la compression par exemple. Les ASIC, pour *Application Specific Integrated Circuit*, sont quant à eux des puces intégralement dédiées à la réalisation d'une tâche précise. Par exemple, la gestion des liens de télécommunication et surtout ceux nécessitant le traitement d'informations analogiques, nécessite des puces conçues pour ce travail. Ce type d'accélérateur fournit le meilleur ratio consommation/performance mais est aussi le plus couteux. Enfin, les circuits logiques programmables ont des capacités permettant leur utilisation en tant qu'accélérateurs matériels très polyvalents [EG+08]. Ces derniers, et plus particulièrement les réseaux de portes programmables plus communément appelés FPGA (pour *Field Programmable Gate Array*), font l'objet de ce manuscrit.

2.1.2 Les FPGA et l'accélération matérielle

Description des FPGA

Les FPGA sont des puces électroniques présentant la particularité de pouvoir être programmées physiquement afin d'adopter un comportement voulu. On parle dans ce cas particulier de puce reconfigurable [1], par opposition au mot programmable qui a un sens particulier dans le monde du microprocesseur, programmable signifiant capable d'exécuter des instructions stockées dans une mémoire. La Figure 2.1 présente un échantillon très simplifié et généraliste de l'architecture d'un FPGA du constructeur Xilinx. Les CLB, pour *Configurable Logic Block*, sont les éléments

1. On utilisera indistinctement dans tous ce manuscrit les termes de FPGA, puce reconfigurable ou cible reconfigurable pour désigner le même objet.

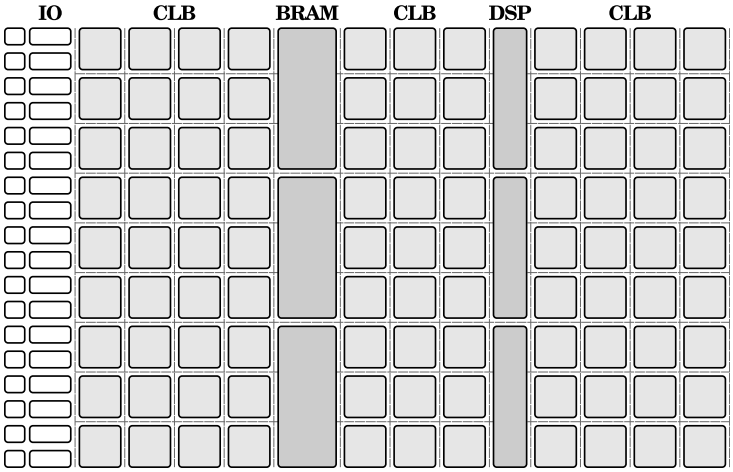


FIGURE 2.1 – Vue générale d'un FPGA (famille Xilinx).

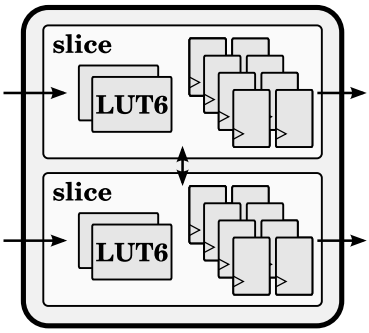


FIGURE 2.2 – *Configurable Logic Block* (CLB) : unité de base du FPGA (famille Xilinx).

de base conférant sa modularité au FPGA. Un CLB de la technologie Xilinx Virtex 7 est représenté en Figure 2.2 [Xil14]. Il contient deux éléments principaux : des *Look-Up Tables* ou LUT et des *Flip-Flops* ou FF. Les LUT servent à implémenter des fonctions logiques (`OR`, `AND`, `XOR`, etc.) et les FF sont des registres d'un bit permettant le stockage d'une information binaire. À partir de ces deux éléments simples, regroupés dans des *slices* elles-mêmes encapsulées dans les CLB, le développeur d'application FPGA est capable de générer n'importe quel circuit numérique, moyennant l'utilisation d'une suite d'outils dits « de synthèse ». Afin d'améliorer les capacités des FPGA, les constructeurs ont toutefois ajouté d'autres éléments en plus des CLB. On compte notamment des blocs contenant de la mémoire à accès direct (BRAM) facilitant le stockage de grandes quantités d'information. En plus, on retrouve des DSP de petite taille permettant d'améliorer la rapidité des calculs mathématiques comme des additions et des multiplications. Tous ces éléments, CLB, DSP et BRAM, sont connectés entre eux à l'aide d'une matrice d'interconnexion. Des blocs d'entrée/sortie (IO sur le schéma) sont responsables de la communication avec l'extérieur. La puce est configurée à l'aide d'un flux binaire dont la copie dans une mémoire de configuration définit l'état de tous les éléments constitutifs du FPGA.

Les FPGA pouvant potentiellement implanter matériellement n'importe quelle

fonction, ils sont des candidats idéaux pour côtoyer les processeurs classiques afin de les assister dans des tâches que ces derniers auront du mal à effectuer [Vah+08]. Ajoutons à cela qu'ils présentent des avantages en terme de consommation et de rapidité par rapport aux processeurs, étant donné le niveau de spécialisation qu'il est possible d'en tirer. Enfin, contrairement à la plupart des accélérateurs matériels, les FPGA peuvent servir à une multitude de tâches différentes puisqu'ils peuvent être reconfigurés.

Place des FPGA dans le monde de l'accélération

Les FPGA sont parfaitement adaptés afin de servir d'accélérateurs matériels. Les améliorations techniques se succèdent [Poc+13] et permettent aux champs applicatifs de se multiplier. Chaque année apporte son lot d'applications ayant une nouvelle implantation sur FPGA et ayant des résultats significativement performants : accélération de calculs liés aux génomes [Ols+12] ; accélération de multiplication matricielle [Dav+07] ; implémentation de réseaux neuronaux convolutifs utilisés en reconnaissance d'image [Zha+15a] ; accélération de classification [Ton+13]. Toutes ces disciplines ayant pour facteur commun l'accélération matérielle sur FPGA peuvent d'ailleurs s'appuyer sur certains pans de littérature ayant pour unique but de faciliter le développement sur cible reconfigurable [Her+07].

Cela fait donc plusieurs années que les FPGA occupent une place dans le domaine de l'accélération matérielle. Pourtant, toutes les applications accélérées ne le sont que de manière ponctuelle et les expériences ne sont pas systématiquement reproductibles. Il est d'ailleurs frappant de toujours observer de nos jours des publications scientifiques relatant le succès d'un implantation sur puce reconfigurable. Ce domaine n'est il pas considéré comme industriellement stable ? En réalité, le constat est le suivant : il est possible et la plupart du temps avantageux d'accélérer une application à l'aide d'un FPGA comparé à une implantation CPU ou GPU (parfois même dans le cadre du traitement d'image selon [Asa+09]). Pourtant, ce n'est qu'au cas par cas que les développeurs d'applications accèdent à ce genre de technologies. La place des FPGA dans le monde de l'accélération matérielle est finalement petite, car ces puces ne sont pas considérées de manière suffisamment systématique comme une solution d'accélération.

> — Pourquoi les FPGA ne sont-ils pas utilisés plus systématiquement en tant qu'accélérateurs matériels ?

Contraintes liées aux FPGA

Tout d'abord, les puces reconfigurables présentent une accessibilité réduite comparé aux technologies existantes d'accélération matérielle sur processeur graphique par exemple. Ces derniers sont ciblés à l'aide de langages de programmation bien maitrisés de la part des développeurs logiciels, premiers utilisateurs d'accélérateurs graphiques. Au contraire, pour développer une application sur FPGA, il faut idéalement avoir une connaissance de son architecture afin d'en appréhender le paradigme de fonctionnement (cf. 2.1.2). La qualité des applications produites en

dépend. La conception de circuits est une discipline fondamentalement différente du génie logiciel et on utilise usuellement des langages de description d'application différents. On appelle ces langages des HDL, pour *Hardware Description Language*, parmi lesquels le VHDL et le Verilog sont les plus connus. Développer un accélérateur matériel ayant pour cible un FPGA requiert donc des compétences et des connaissances spécifiques. Le développement en lui même peut être compliqué de par la nature de la cible FPGA. On pense à la difficulté de déboguer une application qui est finalement cachée, déportée dans une puce électronique lors de sa phase d'utilisation. De plus, une fois la configuration du FPGA générée (le flux binaire), celle-ci est fixée pour le type de FPGA ciblé. Le port d'une application d'un FPGA à un autre n'est une tâche ni automatique, ni aisée.

Enfin, c'est tout ce qui doit environner le FPGA pour assurer son fonctionnement qui reste à établir. Insérer le FPGA dans un système complet pose de nombreux problèmes dont beaucoup sont encore le sujet de recherches, cette thèse en faisant d'ailleurs partie. Il faut définir le canal de communication entre le FPGA et son entourage par exemple. Il peut aussi y avoir plusieurs FPGA dans le système, ou le FPGA peut être considéré comme fractionné en plusieurs parties. Enfin, peu importe la topologie adoptée, il est nécessaire de gérer cette ressource, et notamment de la partager entre les différents utilisateurs du système. Ce partage est d'autant plus intéressant en théorie car le FPGA, capable d'accueillir *a priori* n'importe quelle application, peut servir à remplir de nombreuses fonctions au sein d'un système, au contraire de la plupart des accélérateurs matériels.

Afin de résumer ces paragraphes, on peut considérer que l'adoption des FPGA passe par trois axes. En effet, il faut rendre le FPGA :

— facile à programmer et les programmes, faciles à distribuer ;
— disponible dans les systèmes matériels ;
— d'une gestion transparente et flexible pour les utilisateurs.

Les deux premiers points sont d'un ressort qui dépasse le cadre scientifique de la plupart des travaux. C'est donc sur le dernier point que nous nous focalisons dans le reste de ce manuscrit. La question que pose la gestion du FPGA en tant que périphérique est centrale dans l'adoption massive de ceux-ci. Sont donc soulevées les questions qui suivent :

— Comment rendre la gestion des FPGA transparente pour les utilisateurs ?
— Quelle type de prise en charge du FPGA par le système doit être mise en place à ces fins ?

2.2 Gestion d'un système

2.2.1 Le partage des ressources d'un système

Un système complet est composé d'un ou plusieurs CPU, d'une mémoire distribuée de manière hiérarchique (caches, RAM, disques ...), de périphériques d'entrée-sortie, d'accélérateurs matériels etc. Il est complexe par nature et l'utilisation d'un système d'exploitation, ou SE, est nécessaire pour deux raisons. D'une part, le

SE fournit une abstraction des éléments matériels constitutifs du système ainsi que des méthodes simplifiées pour interagir avec eux. D'autre part, il permet le partage de ces ressources parmi les utilisateurs. De plus amples précisions sont disponibles dans les ouvrages de référence [Tan01] et [Sil+98], ainsi que dans la thèse [Fas01]. L'abstraction de la ressource qui nous intéresse plus particulièrement dans ce manuscrit, le FPGA, est un domaine de recherche à part entière. Nous nous intéresseront plutôt au second type d'actions que peut avoir le SE sur les ressources d'un système : l'assurance de leur partage parmi les utilisateurs.

Dans le monde informatique, la première ressource à partager parmi les utilisateurs est le processeur lui-même. Plusieurs programmes peuvent fonctionner de manière concurrentielle et il est important d'assurer à tous un accès équitable au processeur. Historiquement, ce partage était nécessaire car le temps d'attente lié à l'accès aux ressources mémoires rendait le processeur inactif la plupart du temps. La multiprogrammation est venu répondre à ce problème en allouant le processeur à une nouvelle tâche le temps que la mémoire réponde à la requête de la première tâche. L'évolution logique de cette fonctionnalité correspond au mécanisme de temps partagé. Le processeur peut être utilisé simultanément par plusieurs utilisateurs en mettant certaines tâches en liste d'attente pendant que d'autres sont exécutées pendant une certaine durée. Établir une liste de priorité des tâches à effectuer est le rôle de l'ordonnanceur, un composant du système d'exploitation. Cette tâche *a priori* anodine est en réalité extrêmement compliquée. En effet, de nombreuses hypothèses et choix d'implémentation entrent en jeu. Pour rester volontairement généralistes, nous dirons qu'il existe majoritairement trois types d'ordonnanceurs : coopératifs, préemptifs et hybrides. Un système coopératif laisse les tâches s'exécuter sans contrôle. L'ordonnanceur n'entre en jeu que lorsqu'une tâche relâche le processeur et qu'il faut décider de la tâche qui pourra accéder à la ressource à son tour. Un système préemptif, au contraire, alloue la ressource à une tâche pendant une durée déterminée. À la fin de cette période, si la tâche n'est pas terminée, l'ordonnanceur fait une sauvegarde de cette tâche (plus précisément de son contexte d'exécution) afin de libérer le processeur. Une fois ceci fait, le processeur est alloué à une autre tâche. Quand son tour viendra de nouveau, la tâche précédemment sauvegardée pourra redémarrer. Enfin, le mécanisme hybride possède aussi des fenêtres d'exécution bornées. Il est donc semi-préemptif. Mais à la fin de celles-ci, il éjecte la tâche de la ressource sans sauvegarde préalable. Ce type d'ordonnanceur est donc aussi semi-coopératif dans le sens ou la responsabilité de la sauvegarde est transférée au développeur d'application. Dans le cas général, seuls ces deux derniers types d'ordonnanceurs sont implantés dans les systèmes d'exploitation modernes [Sil+98].

2.2.2 Changement de contexte

Que le type d'ordonnanceur du système d'exploitation soit hybride ou préemptif, les tâches utilisant le processeur doivent se contenter d'une fenêtre d'exécution bornée. Cette fenêtre est potentiellement inconnue du développeur d'application. Lors d'une préemption, il est donc indispensable de sauvegarder le contexte d'exécution de l'application en cours afin de pouvoir la redémarrer par la suite. Dans le

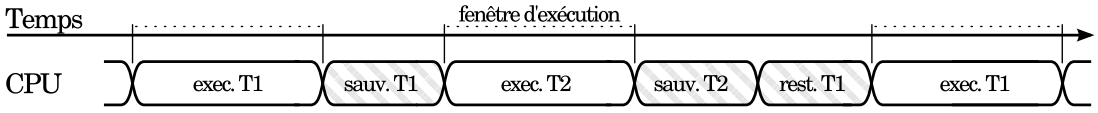


FIGURE 2.3 – Changement de contexte sur CPU.

cas contraire, une tâche durant plus longtemps que la fenêtre d'exécution pourrait ne jamais aboutir. La Figure 2.3 illustre ce mécanisme. Le processeur alterne les phases d'exécution des tâches T1 et T2, dont la durée est fixée par la fenêtre d'exécution. Entre ces exécutions ont lieu (phases hachurées) les sauvegardes (sauv.) et restaurations (rest.) de contexte. Celles-ci représentent le surcout temporel qu'engendre le mécanisme de changement de contexte. Sur la figure, la durée de ces différentes phases est non représentative. La sauvegarde de contexte peut-être effectuée par le système d'exploitation (système préemptif) ou prévue par le développeur d'application (système hybride). Cette précision n'est pas illustrée sur la figure.

Le mécanisme de changement de contexte sur processeur a posé et pose encore de nombreuses questions [Li+07]. Malgré ça, il reste utilisé très largement comme mécanisme de partage du processeur entre les utilisateurs. C'est de ce mécanisme dont nous souhaitons nous inspirer afin de permettre l'utilisation du FPGA en tant qu'accélérateur matériel accessible, partageable et distribuable. Les questions suivantes se posent donc une fois ce choix arrêté :

- Peut-on appliquer des méthodes de gestion de ressources éprouvées par ailleurs aux FPGA ?
- Comment libérer une ressource matérielle qu'il faut partager au sein d'un système multi-utilisateur potentiellement distribué ?

2.3 Changement de contexte sur FPGA

2.3.1 Adaptation du changement de contexte aux FPGA

Si l'on souhaite procéder à un changement de contexte sur FPGA, la marche à suivre n'est pas la même que lorsqu'on traite avec un microprocesseur, ces deux objets étant de nature très différente. D'une part, les manières de démarrer l'exécution d'un programme ou d'une tâche matérielle ne sont pas comparables. La Figure 2.4 montre la chronologie d'un changement de contexte sur FPGA qui permettrait à deux utilisateurs d'exécuter respectivement les tâches T1 et T2 sur une puce reconfigurable. La différence notable entre ce cas et celui présenté sur la Figure 2.3 est la phase de configuration (conf.) du FPGA. En effet, afin d'exécuter une tâche matérielle sur une puce reconfigurable, il faut au préalable configurer cette dernière avec le circuit chargé d'exécuter la tâche voulue. La phase de configuration précède donc toute utilisation du FPGA par un utilisateur. Celle-ci s'ajoute au surcout temporel du changement de contexte (phases hachurées), ce qui réduit d'autant la phase de fonctionnement effectif du FPGA.

Ensuite, sauvegarder le contexte d'un programme en cours d'exécution sur un

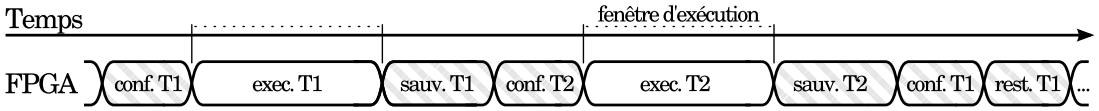


FIGURE 2.4 – Changement de contexte sur FPGA.

processeur est un processus non complexe. Cela consiste majoritairement à sauvegarder les registres du processeur en mémoire. Une tâche matérielle ne possède pas d'architecture prédéfinie et le contexte peut potentiellement se trouver dans tous les éléments mémorisants présents dans le circuit (mémoire de configuration, FF et blocs de mémoire RAM).

2.3.2 Applications d'un mécanisme de changement de contexte sur FPGA

Le changement de contexte sur FPGA n'est pas un sujet nouveau et la littérature abonde de projets de recherche allant dans ce sens. Une implantation matérielle d'une puce reconfigurable supportant le changement de contexte de manière native a été proposée par [Tri+97] puis par [Sca+98]. Un mécanisme de changement de contexte sur FPGA ouvre la porte à de nombreuses applications, présentes dans la littérature depuis déjà un certain nombre d'années. En plus de la gestion multi-utilisateur présentée dans ce chapitre, on compte par exemple la réallocation de tâches [Kal+06], la défragmentation [Fek+12] ou encore la tolérance aux fautes [Egw+13]. Toutes ces applications, déjà envisagées et implantées, peuvent voir leurs capacités démultipliées par l'application de la reconfiguration dans un premier temps, puis de la reconfiguration partielle [Pap+11], et enfin de la reconfiguration dynamique partielle [Sed+06 ; EA+09 ; Fek+12]. Cette dernière technique peut en outre donner naissance à des applications originales. On peut notamment citer l'atténuation des fautes liées au vieillissement des FPGA [Zha+15b].

2.3.3 Hétérogénéité des FPGA

Le développement d'un mécanisme de changement de contexte pour tâches matérielles s'exécutant sur une puce reconfigurable ouvre de nombreuses perspectives. Pourtant, loin d'être une évidence, sa mise en œuvre effective pose un certain nombre de questions. L'une des plus importantes concerne la portabilité d'un tel mécanisme. En effet, pour que celui-ci puisse s'appliquer à tout système, il faut tenir compte de la nature très hétérogène des FPGA. Alors que chaque système d'exploitation possède une marche à suivre pour chaque architecture de processeur (par exemple x86, arm, mips ...) supportant le changement de contexte, les FPGA ne présentent aucune architecture référencée de la sorte. Chaque constructeur de puce change l'architecture des matrices reconfigurables, parfois de manière radicale, entre les différentes générations. Il est dès lors incommode de définir une marche à suivre ne dépendant que du type d'architecture de FPGA. S'il n'est pas possible de s'appuyer sur des mécanismes prévus par les constructeurs de FPGA, il faut donc développer

la fonctionnalité souhaitée à l'aide de la matrice reconfigurable. Ce type d'approche présente deux défauts majeurs. D'une part il implique le développeur d'application dans la prise en charge du mécanisme, ce faisant le rendant peu accessible au plus grand nombre. D'autre part, il induit un surcout en terme d'utilisation des ressources de la puce. La nature des FPGA en font des candidats peu enclins à la mise en place d'un mécanisme de changement de contexte simple et universel, tel que c'est actuellement le cas dans les microprocesseurs actuels. Pour inverser cette tendance, il faut répondre aux interrogations suivantes :

- Comment produire un mécanisme adaptatif, fonctionnant pour tout type de FPGA ?
- Comment rendre l'utilisation de ce mécanisme transparente pour les développeurs ?
- Comment réaliser un mécanisme peu couteux et peu impactant ?

2.4 Synthèse de la problématique

Ce chapitre illustre la problématique qui sous-tend le contenu de ce manuscrit. Nous avons vu la place occupée par les FPGA dans le contexte actuel d'accélération matérielle, et les raisons de cet état des lieux. Afin de favoriser l'émergence des FPGA comme solution d'accélération matérielle, on a aussi vu qu'il était nécessaire de répondre à plusieurs questions et notamment celles qui concernent la gestion de ce genre d'équipement. Ceci nous a donc amené à considérer le changement de contexte comme brique de base de tout système de gestion de tâche matérielle sur FPGA. Les questions de synthèse auxquelles nous allons tenter de répondre dans ce manuscrit sont les suivantes :

- Quel mécanisme de changement de contexte imaginer pour fonctionner sur FPGA ?
- Comment rendre ce mécanisme adapté aux ordonnanceurs existants ?
- Comment rendre l'utilisation de ce mécanisme transparente pour les utilisateurs du système ? pour les développeurs d'application ?
- Comment faire pour que ce mécanisme soit le moins couteux et le moins impactant possible ?

Chapitre 3

État de l'art

Ce chapitre présente l'état de l'art relatif à la problématique générale soulevée au début du manuscrit : comment partager une ressource reconfigurable au sein d'un système ? La mise en perspective des travaux existant aura pour but d'éclairer la compréhension de cette thèse.

La première partie d'analyse de la littérature porte sur les techniques permettant de partager une ressource reconfigurable FPGA mis à part. On focalise ensuite l'analyse de l'état de l'art relatif au partage des FPGA. Enfin, on proposera les premières hypothèses de travail compte tenu des objectifs visés.

Sommaire

3.1 Définitions

Les définitions qui suivent, relatives au changement de contexte sur cible reconfigurable, serviront tout au long du manuscrit.

Système de base Un système de base, qui est un point de départ pour les raisonnements développés, est constitué d'un bloc de contrôle, d'un bloc reconfigurable ainsi que d'une mémoire partagée entre ces deux premiers éléments. Tous ces composants sont reliés par un système d'interconnexion. Le système générique servant à mettre en œuvre un changement de contexte sur cible reconfigurable est visible sur la Figure 3.1. On peut effectivement mettre en œuvre ce système de plusieurs façons. Par exemple, le bloc de contrôle peut être un CPU sur lequel s'exécute un programme de gestion (potentiellement un SE, voire un programme plus basique s'exécutant directement sur le processeur c'est-à-dire en *bare metal*). Le bloc reconfigurable peut être un FPGA connecté à un bus de communication. Enfin, la mémoire peut être une RAM connectée sur le même bus de communication. Ces éléments peuvent être connectés sur une carte mère en tant que puces différentes mais pourraient aussi bien être regroupés au sein d'un système sur puce (tous les éléments dans une même puce soit dans un SoC pour *System on Chip*). Un autre système, fonctionnel aussi, peut contenir tous les éléments au sein d'un seul FPGA. Une portion du FPGA est dans ce cas dédiée au bloc de contrôle, une autre à la mémoire et une dernière est laissée vierge pour la future application. Ces éléments sont connectés par un bus lui aussi émulé par la logique reconfigurable.

Tâche matérielle Dans ce manuscrit, on utilisera la définition suivante pour une tâche. Une **tâche** correspond à l'exécution d'un ou plusieurs algorithmes à l'aide d'un élément d'un système de calcul. Une tâche peut avoir besoin de données d'entrées et peut à son tour être productrice de données. Par exemple, l'exécution d'un algorithme de diagonalisation d'une matrice prend en entrée une matrice de dimension fixée et produit une seconde matrice, diagonalisée, de même dimension. Un tel algorithme peut s'exécuter sur un processeur, auquel cas les instructions successives du logiciel mettant en œuvre l'algorithme sont lues et exécutées par le CPU. Cet algorithme peut aussi avoir une implantation matérielle. C'est-à-dire qu'on peut créer un circuit réalisant l'algorithme et mettre en œuvre ce circuit sur un FPGA. Dans ce cas, on parle de tâche matérielle, car la tâche s'exécute matériellement. Une **tâche matérielle** est une tâche s'exécutant à l'aide d'un circuit idoine sur une cible matérielle (reconfigurable ou non). La tâche matérielle est aussi potentiellement consommatrice de données ainsi que productrice de données. La Figure 3.2 présente un système relativement générique composé de deux CPU, une zone reconfigurable, une mémoire, un gestionnaire d'entrée/sortie (appelé E/S) ainsi que d'un bus reliant tous ces éléments. Elle illustre la différence entre les tâches logicielles (« T »), exécutées sur CPU de la tâche matérielle (« T_m ») exécutée par la zone reconfigurable. T1 ne nécessite pas de données d'entrée mais est productrice de données. Celles-ci sont transférées à la mémoire une fois produites. T2, comme T1, est productrice de données mais aussi consommatrice. Elle obtient ses données d'entrée depuis le

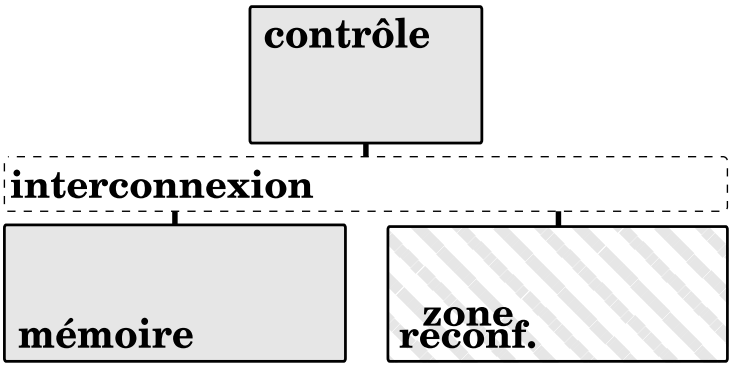


FIGURE 3.1 – Système générique servant à la mise en œuvre d'un changement de contexte sur FPGA.

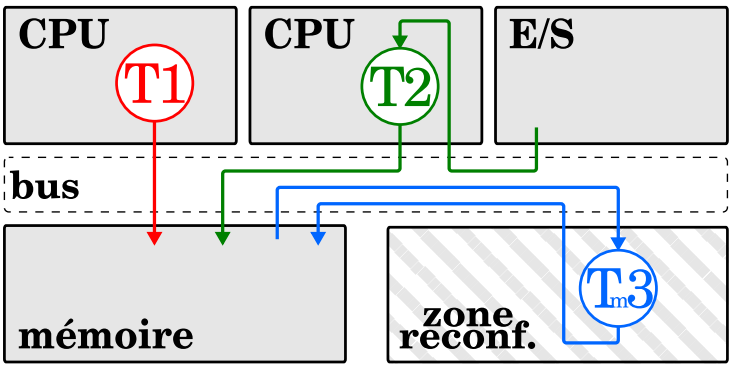


FIGURE 3.2 – Système générique, trois tâches s'exécutent.

gestionnaire d'entrée/sortie. Enfin, T_m3 est une tâche matérielle. Elle s'exécute dans la zone reconfigurable. Comme T2, elle est consommatrice et productrice de données. Enfin, T_m3 obtient ses données d'entrée depuis la mémoire.

Contexte matériel Le **contexte** d'une tâche est l'ensemble des informations nécessaires et suffisantes pour que la tâche puisse arrêter son exécution et la reprendre plus tard, éventuellement sur une plateforme hôte différente. En d'autres termes, c'est non seulement l'architecture mise en œuvre par la tâche, donc comment se comporte la tâche, mais aussi les éléments mémorisants (registres par exemple) sur lesquels l'architecture agit. Ces derniers évoluent tout au long de l'existence de la tâche.

3.2 Partage de ressources reconfigurables hors FPGA

Le partage d'une ressource reconfigurable au sens large, c'est-à-dire pas seulement des FPGA, est un domaine de recherche dont les ramifications sont nombreuses. Dans les paragraphes suivants, on se focalise sur les ressources reconfigurables hors

FPGA dans l'optique de saisir la diversité des solutions qu'il est possible d'adopter afin de partager une telle ressource.

3.2.1 Puces « multi-contexte »

Les efforts des chercheurs afin d'obtenir une puce reconfigurable ayant la capacité de changement de contexte sont apparus avant les années 2000 [Hau+10]. Ces architectures ne sont pas à proprement parler des FPGA mais plutôt des dérivés. Les premières publications du genre présentent une puce alors appelée « multi-contexte », avec par exemple les DPGA [DeH94 ; Tau+95]. Cette puce permet un échange rapide, de l'ordre du cycle d'horloge, de différentes configurations de la puce. L'objectif est de multiplexer temporellement plusieurs applications, ou portions d'une même application, déportées sur différents contextes, sur la ressource reconfigurable et ceci de manière potentiellement fine. La Figure 3.3 schématise ce type d'architecture. Afin de réaliser ce multiplexage, l'idée est d'ajouter autant de mémoires de configuration que d'applications, ou portions d'application, pouvant s'exécuter en parallèle. Dans le circuit pris en exemple, deux mémoires de configurations sont disponibles. Le changement de contexte s'effectue en échangeant le contenu des bascules de configuration avec celui de la mémoire de configuration voulue. Cette étape de commutation, contrairement à toutes les technologies existantes, s'effectue dans un temps très court, de l'ordre de la dizaine de nanoseconde. Une approche assez similaire mais possédant plutôt une orientation émulation est proposée dans la même période par [Jon+95]. Constatant que les architectures du genre manquaient de capacités en unités logiques, en mémoire de configuration et en mémoire embarquée [Tri+97] propose une nouvelle solution basée sur les même concepts tout en améliorant les points cités, de même que [Cha+97] avec l'outil Dharma. D'autres évolutions suivront comme celles de CSRC [Sca+98]. Les principales faiblesses de ces puces « multi-contexte » sont leur cout, leur efficacité énergétique et enfin la surface utile de la puce qui est très réduite [DeH96]. Un autre argument qui n'est pas en faveur de ce type de technologie concerne le bénéfice énergétique obtenu par multiplexage temporel d'une même application entre différents étages de configuration. Pour [Par+15], sauf cas particuliers, il est moins efficace en terme d'énergie consommée de mettre une application sur plusieurs plans de configuration d'une puce « multi-contexte » plutôt que d'utiliser un FPGA suffisamment grand pour accueillir l'application.

3.2.2 Architectures « gros-grain »

Il existe des puces cousines des FPGA faisant partie de la famille des puces reconfigurables. Les CGRA, pour *Coarse Grain Reconfigurable Arrays*, dont la notoriété a été au plus haut au milieu des années 2000 [Tod+05], possèdent la particularité d'avoir une densité d'éléments reconfigurables bien plus faible que les FPGA. Ces éléments, ou unités fonctionnelles, ont des capacités plus élaborées que les CLB des FPGA, et sont aussi connectés via une matrice de routage. Le nombre moindre d'éléments à configurer entraine une diminution drastique de la taille du flux binaire

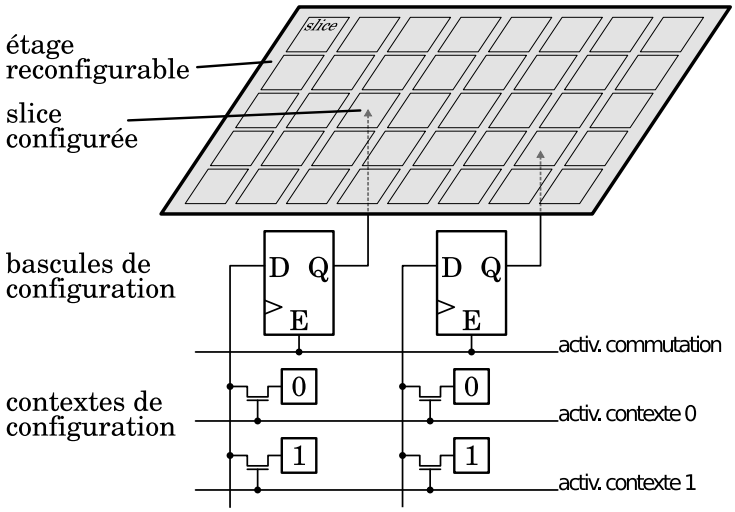


FIGURE 3.3 – Schéma d'un FPGA multi-contexte. Deux contextes (0 et 1) sont commutables.

de configuration d'un CGRA comparé à celui d'un FPGA. De nombreux prototypes ont été proposés dans la littérature, tels que : PADDI [Che+92] MATRIX [Mir+96], MorphoSys [Sin+00], Piperench [Gol+00], DART [Dav+02], ADRES [Mei+03]...

Seul MorphoSys propose le multiplexage temporel des contextes d'exécution et donc le partage de la ressource. Ce prototype propose en effet la même fonctionnalité que celle présentée dans le paragraphe précédent : plusieurs mémoires de configuration permettent un multiplexage des applications s'exécutant sur le CGRA. L'avantage principal des CGRA sur les FPGA réside dans la taille réduite du flux binaire de configuration qu'il est nécessaire de générer et transférer.

3.2.3 Solutions tierces

[Wat+11] présente une solution architecturale dans laquelle un système multi-cœur est fortement couplé à une matrice reconfigurable. Cette matrice a deux rôles : elle met en œuvre des accélérateurs matériels qui viennent en support des processeurs et, plus original, elle assure les communications entre des processeurs voisins. Techniquement, la matrice reconfigurable est partagée entre quatre processeurs. Ceux-ci peuvent donc utiliser la zone reconfigurable non seulement à leurs propres fins calculatoires mais peuvent aussi s'en servir pour communiquer. En effet, la sortie d'un accélérateur peut être consommée par un cœur différent que celui fournissant les entrées par exemple. Cette contribution ne propose pas à proprement parler un multiplexage temporel d'une ressource reconfigurable mais organise plutôt le partage en surface de celle-ci entre plusieurs cœurs. Pour finir, la ressource reconfigurable proposée par WATKINS et al. est de type « gros-grain ».

GARCIA et al. [Gar+09] proposent aussi une architecture nouvelle appelée OLLAF. Celle-ci, dérivée des FPGA dans le sens ou c'est une architecture reconfigurable à grains fins, est pensée pour permettre la préemption de tâches matérielles et faciliter la reconfiguration dynamique. L'architecture est reconfigurable par colonne.

Chaque colonne possède un gestionnaire de configuration ainsi qu'un gestionnaire de contexte. De plus, la colonne est divisée en deux parties identiques, l'une d'elles pouvant être active exclusivement. Ceci permet de changer l'application s'exécutant dans une colonne en un cycle d'horloge moyennant un pré-chargement du contexte et de la configuration dans la partie alors inactive de la colonne. OLLAF est adapté à la gestion dynamique de tâches matérielles car les différents niveaux de stockage de l'information (contexte et configuration) agissent comme des caches.

Une entreprise, Tabula – qui a disparu depuis, a tenté de commercialiser une puce particulière. L'objectif initial n'était pas de partager la ressource entre divers utilisateurs, mais de reconfigurer la puce à une fréquence très élevée ($1,6$ GHz) afin d'émuler une quantité de ressource plus grande. Huit plans de configuration sont utilisés et en les multiplexant, on obtient un circuit paraissant fonctionner à 200 MHz sur une surface multipliée par huit. La seule source décrivant cette technologie est [Hal10]. C'est un exemple de multiplexage temporel se rapprochant du DPGA cité précédemment.

Une méthode plus anecdotique consiste à utiliser une mémoire de configuration holographique permettant de contenir plusieurs contextes de configuration. Cette mémoire de configuration est nécessairement utilisée par une ressource configurée de manière optique (ORGA) pour *Optically Reconfigurable Gate Array* [Set+08 ; Nak+09]. Ces technologies n'ont pour l'heure pas encore eu de retombées industrielles.

3.3 Partage des FPGA

Le FPGA ne possède pas nativement les mécanismes nécessaires à son partage. Il faut donc exploiter les caractéristiques des FPGA afin d'ajouter une telle fonctionnalité. Le partage d'un FPGA se base sur une action fondamentale : le changement de contexte (cf. 2.2.2). On présente dans un premier temps la méthode historiquement la plus répandue de relecture du flux binaire de configuration. Ensuite, on aborde les méthodes dites « embarquées » car leur comportement est inclus dans la description du circuit. Enfin, on étudie des méthodes émergentes puis on liste des plateformes de démonstration existant dans la littérature.

3.3.1 Méthode de relecture

La méthode de relecture est la méthode historique et la plus répandue dans la littérature lorsqu'il s'agit de sauvegarde de contexte relative aux FPGA. Cette méthode ne concerne toutefois que les puces du constructeur Xilinx. En effet, celui-ci est le seul à proposer un port de configuration « bidirectionnel », c'est-à-dire qu'après la configuration d'un FPGA, il est possible de faire une requête pour relire le flux binaire de configuration en cours d'utilisation. Le contenu des éléments mémorisants étant inclus dans ce flux, même après déroulement d'une tâche, extraire le flux binaire constitue une relecture du contexte. La première proposition visant à produire un système reconfigurable basé sur FPGA multitâche est avancée par [Sim+00]. Ils

proposent en outre de filtrer les informations extraites de la configuration pour ne garder que les données utiles au redémarrage de la tâche extraite. Grâce à ce tri, ils réduisent de 90% la quantité d'information à stocker, la majorité du flux binaire de configuration contenant des informations de routage. La manipulation du flux n'est malheureusement pas reproductible de nos jours. Seules les configurations de certains modèles de FPGA Xilinx ont été révélées dans les années 2000 et cela n'a pas été reproduit depuis. La principale faiblesse de cette approche, outre son verrou technologique, est sa lenteur. Extraire la configuration d'un FPGA s'effectue en un temps de l'ordre de grandeur de la dizaine de millisecondes. Sur les technologies Virtex, la routine de relecture et de configuration est gérée par un bloc appelé *Internal Configuration Access Port* (ICAP) [Xil10]. C'est sur ce port que s'appuient [Blo+03a ; Blo+03b] et [Ull+04]. BLODGET et al. [Blo+03b] utilisent pour la première fois le port ICAP et proposent de reconfigurer le FPGA depuis un composant inclus dans la zone reconfigurable grâce à la reconfiguration partielle. ULLMANN et al. [Ull+04] proposent un démonstrateur mettant en œuvre un système permettant la reconfiguration dynamique partielle à la demande. [Kal+05] ont aussi cherché à tirer parti de la reconfiguration partielle. L'objectif est pour eux de n'extraire que les portions utiles du flux de configuration, faisant parti du contexte. Ils évitent ainsi l'extraction de la totalité de la configuration et diminuent donc la taille du contexte et le temps d'extraction. Plus récemment, [Han+11] ont proposé une amélioration du bloc matériel correspondant à l'ICAP afin d'augmenter sa fréquence de fonctionnement et ainsi améliorer son débit. Enfin, un système préemptif a été adapté sur Virtex 4 par [Joz+10].

Une autre approche, portée par [Res+05], veut limiter le temps de reconfiguration et propose donc un gestionnaire de reconfiguration. Celui-ci précharge les flux binaires de configuration et sélectionne la meilleure zone pouvant accueillir la nouvelle tâche.

Ce type de méthode par relecture n'introduit aucun surcout matériel ni aucune dégradation des performances du circuit car elle s'appuie sur un mécanisme intégré dans la puce reconfigurable. Par contre, il n'est pas possible de transférer le contexte extrait d'un modèle de FPGA à un autre, les flots de configuration respectifs étant différents. Enfin, le contexte extrait contient systématiquement des informations non utiles au redémarrage de la tâche comme la configuration des éléments de routage et des LUT. KALTE et al. [Kal+05] indiquent que seulement 8% du contexte extrait par relecture est utile. Pour finir, seuls les FPGA du vendeur Xilinx possèdent la capacité de relecture.

3.3.2 Méthodes embarquées

Le principe d'une méthode d'extraction de contexte embarquée consiste à intégrer directement dans la représentation du circuit (avant sa transformation en flux binaire de configuration dans le cas d'une cible FPGA) les structures permettant la fonctionnalité. En d'autres termes, une fois construit, le circuit embarque le mécanisme qui permet l'extraction et la restauration de contexte. La technique historique utilisée pour les ASIC s'appuie sur une norme appelée JTAG, qui est

le standard IEEE 1149.1-1990. Celle-ci consiste à donner l'accès en écriture et en lecture à un certain nombre de registres à l'aide d'une chaîne de sérialisation et d'un protocole standardisé. Cet accès privilégié au cœur des circuits sert notamment à déverminer des logiciels en permettant le déroulement pas à pas d'un programme et la visibilité des registres du processeur. Il sert aussi à tester le circuit équipé. On parle dans ce cas de *boundary scan testing*. La chaîne de sérialisation utilisée est de un bit de large et relie entre eux tous les registres qu'on souhaite rendre accessibles. Les premières méthodes d'extraction de contexte sur FPGA s'inspirent de ce standard, c'est donc en général une chaîne de sérialisation connectant tous les registres d'un circuit qui est utilisée afin d'extraire le contexte d'une tâche matérielle. Cette chaîne est intégrée à la description du circuit, qu'elle soit HDL ou netlist.

Les premiers à donner des capacités d'extraction de contexte à des tâches matérielles s'exécutant sur FPGA sont [Whe+01]. Ils exposent la méthode basée sur une chaîne de sérialisation à mettre en place afin de permettre l'extraction des registres et des mémoires d'un circuit. L'article décrit la procédure à mettre en œuvre pour développer de tels circuits. C'est donc un ajout manuel, effectué par le développeur d'application, qui est nécessaire afin de fournir la capacité d'extraction de contexte au circuit. Par ailleurs, [Mig+03] proposent de sélectionner des moments spécifiques lors desquels l'extraction de contexte est moins couteuse. Typiquement, ils souhaitent n'autoriser une extraction qu'après déroulement des phases calculatoires afin de limiter les données nécessaires au seul résultat. Toutefois, ni la méthode de construction du mécanisme ni la procédure de sélection des moments idéaux ne sont présentés. Afin de fournir une automatisation partielle de la construction des chaînes d'extraction, [Jov+07] proposent d'utiliser un composant registre de plus haut niveau, possédant les interfaces nécessaires à sa connexion dans une chaîne de sérialisation. De plus, il avancent l'idée d'une chaîne de sérialisation dont la largeur serait supérieure à un bit. C'est aussi l'une des propositions de [Koc+07], qui présentent un mécanisme de chaîne de sérialisation basée sur un registre amélioré. En outre, deux autres méthodes embarquées sont proposées. L'une consiste à dupliquer les registres afin de continuer l'exécution du circuit même durant une extraction de contexte. Pour extraire le contexte, une copie des registres du circuit est réalisée dans leur registre miroir. C'est ensuite cette chaîne de registres miroirs qui est extraite pendant que le circuit continue de fonctionner en utilisant les registres usuels. L'autre méthode consiste à utiliser des registres dotés d'une capacité de communication avec une mémoire externe au FPGA. Extraire le contexte consiste alors à copier le contenu de chacun des registres directement en mémoire. [Sch+11] proposent une plateforme dont la tolérance aux fautes est assurée par un mécanisme d'extraction de contexte embarqué. [Gua+08] présentent aussi une plateforme fonctionnelle utilisant un système embarqué, dont l'objectif est de mesurer les performances de différents algorithmes d'ordonnancement. Les instrumentations sont réalisées manuellement et la plateforme n'a pas pour objectif de démontrer la faisabilité d'une méthode embarquée. Néanmoins, ses avantages sont clairement exposés même si aucun résultat concernant le mécanisme n'est exposé.

Embarquer le système d'extraction de contexte à même le circuit possède deux

inconvénients majeurs. D'une part, il nécessite une action de la part du développeur d'application. De plus, le mécanisme ajouté occupe, à des fins d'extraction de contexte, des ressources matérielles présentes dans le FPGA qui aurait été inutilisées. Cette technique est donc responsable d'un cout supplémentaire en surface et peut dégrader les performances du circuit en allongeant son chemin critique. Elle présente néanmoins de solides atouts. La portabilité du mécanisme en est l'un des principaux. La quantité de données à extraire est aussi réduite par rapport au mécanisme de relecture car seul le contexte de la tâche matérielle est extrait.

3.3.3 Autres solutions

Des solutions alternatives existent. La première qu'on peut citer correspond justement à éviter d'extraire le contexte d'une tâche matérielle dans un système multitâche préemptif. Dans [Rup+09], les auteurs présentent trois pratiques de gestion des tâches s'exécutant sur une plateforme reconfigurable. Il faut préciser que dans le système présenté, une tâche matérielle est directement reliée à un processus s'exécutant sur CPU. C'est quand le processus parent est préempté du processeur qu'il s'agit de définir le comportement de la tâche matérielle. La première pratique, simpliste, consiste à bloquer la ressource matérielle avec la tâche mise en pause jusqu'à ce que le processus parent ait de nouveau accès au CPU. Elle ne favorise pas le travail coopératif. La deuxième pratique requiert de libérer la ressource matérielle dès que le processus parent est préempté. Il n'y a aucune sauvegarde de contexte et une potentielle exécution future du noyau d'accélération devra redémarrer de zéro. Enfin, pour éviter les phénomènes de famines et de redémarrage en boucle d'une tâche matérielle trop longue, la dernière pratique consiste à ne pas redémarrer le noyau d'accélération en matériel mais en logiciel, pour cette fois-ci, profiter d'un changement de contexte logiciel à moindre cout. Cette dernière est la plus efficace selon les auteurs, qui prônent un abandon des techniques d'extraction de contexte pour les ressources reconfigurables. Selon eux, cette extraction prend trop de temps. Le temps d'extraction de contexte de la méthode qu'on présente est généralement petit devant le temps d'exécution d'une application, cette réflexion ne s'applique donc pas.

Une seconde approche vise à pallier le déficit de la méthode de relecture. Cette méthode s'appuie sur la notion de FPGA virtuel [Lag+01]. Le FPGA virtuel est une architecture générique qui vient en surcouche d'une puce reconfigurable hôte. L'intérêt majeur de cette surcouche, c'est sa portabilité potentielle sur n'importe quelle cible FPGA configurée pour se comporter tel que le FPGA virtuel le prévoit. Une application décrite pour cibler le FPGA virtuel peut alors s'exécuter sur n'importe quel FPGA hôte. D'autre part, les FPGA virtuels sont généralement à gros grain. La relecture de leur configuration est dans ce cas moins couteuse. Les contraintes des FPGA virtuels sont leur cout en ressources (entre $40\times$ pour [Bra+12] et $100\times$ pour [Lys+05]) et les performances réduites des circuits (multiplication par 10 du chemin critique pour [Lys+05]). Un changement de contexte assorti d'une migration de tâche utilisant une telle technologie est présentée dans [Bol+16].

3.3.4 Plateformes existantes

Dans ce paragraphe on présente une sélection de plateformes bâties dans l'objectif de démontrer la faisabilité de systèmes reconfigurables multitâches.

Les démonstrateurs basés sur la méthode de relecture sont les plus anciens et les plus nombreux. Les premiers à réaliser un tel système sont SIMMLER et al. [Sim+00], sachant qu'ils intègrent un filtrage du flux binaire de configuration. L'architecture présentée par [Lan+02] est similaire, seule la technologie de FPGA cible évolue. Viennent ensuite les technologies de reconfiguration partielle, sur lesquelles s'appuient [Ull+04]. Les technologies récentes font aussi l'objet de démonstrations comme dans [MV+13], dont la particularité est d'embarquer un composant chargé de la gestion de tâche dans la puce reconfigurable. Une intégration à un système d'exploitation spécifique aux ressources reconfigurables (ReconOS [Lüb+09]) des mécanismes de gestion de contexte est réalisée par [Hap+15].

Les méthodes d'extraction de contexte embarquées ont fait l'objet de moins de démonstrateurs. Des exemples notoires existent néanmoins. L'un des premiers démonstrateurs est développé par [Hua+07], mais il ne traite que des applications assez minimalistes et toutes les modifications des circuits sont manuelles. [Jov+07] constituent un exemple utilisant des primitives spécialement pensées pour la commutation de contexte. Enfin, [Joz+12] présentent une plateforme basée sur du *memory map*, mais comme prévu par les concepteurs de la méthode ([Koc+07]), la commutation est lente et l'équipement couteux.

3.4 Synthèse et choix d'une technique

La littérature, abondante, présente une diversité de techniques pour répondre à certaines des questions posées par la problématique de ce manuscrit. Avant d'aller plus loin, il faut sélectionner l'hypothèse de base pour la suite de notre travail : sur quelle technique de commutation de contexte s'appuyer.

Une précision initiale est que l'on cherche à développer une brique de base technologique utilisable le plus simplement possible. Un protocole permet de piloter le circuit doté de la capacité de commutation, peu importe où se trouve le pilote. Un système complet amène son lot de questions dont un certain nombre sont encore débattues et l'objet de recherches actives. Par exemple, quelle topologie pour un système doté de plusieurs cibles reconfigurables ? Comment établir les communications entre le ou les pilotes et les zones reconfigurables ? Ces questions dépassent le cadre des réflexions menées dans ce manuscrit.

D'autre part, on cherche à faciliter l'adoption de puces reconfigurables en élargissant les possibilités qu'elles offrent. Dans le but de cibler le plus de développeurs possibles, il est crucial de rester sur la technologie FPGA, aussi intéressantes que semblent les architectures spécifiques vues en 3.2. Il est en outre indispensable de penser à l'utilisation de plusieurs familles de FPGA voire même de FPGA de constructeurs différents, ceci afin que les produits d'aujourd'hui soient adaptables aux technologies de demain, de manière à de faciliter la maintenance d'un système s'appuyant sur la commutation de contexte etc. Dans ce même objectif est inclus la

volonté d'automatisation de création des circuits commutables.

[But+13] démontrent la supériorité des points de préemption fixes dans le cadre des systèmes temps réels. On cherche donc à faire de la commutation de contexte dont la latence est bornée et la valeur de cette borne, connue. Or, la méthode de relecture ne peut pas assurer le temps d'extraction avant la phase de placement et routage dans le cas d'une extraction limitée par colonne (état de l'art). Dans le cas de l'utilisation d'une méthode embarquée, le temps d'extraction est connu sitôt le mécanisme construit.

Enfin, une question de première importance concerne l'impact que l'ajout d'une telle fonctionnalité peut avoir sur les performances et les couts des circuits ainsi que sur l'ensemble du système. Les méthodes à base de FPGA virtuels répondent aux précédentes attentes (portabilité, temps d'extraction fixé) mais possèdent pour l'instant un surcout peu acceptable par les développeurs d'applications. L'examen de la littérature montre que les méthodes de relectures sont plus lentes que les méthodes embarquées, mais sont néanmoins sans surcout matériel ni dégradation de performances. Embarquer le mécanisme de commutation permet une extraction de données de moindre importance que lors d'une relecture de configuration ainsi qu'une vitesse d'extraction accrue, au détriment du surcout matériel et de performances dégradées. C'est néanmoins cette technique qu'on va adopter car elle l'emporte sur tous les autres aspects à prendre en compte. Nous pensons que la production d'une technique de commutation portable et durable dans le temps ne peut qu'aider à son adoption par une communauté.

Chapitre 4

Méthode

Ce chapitre présente la méthode d'un point de vue global. La portabilité des différents mécanismes proposés dans la littérature a été remise en question dans le chapitre 3. La méthode proposée tente de répondre à ce premier problème en proposant un flot de conception qui permet une utilisation flexible des circuits produits.

En premier lieu, on présente le flot de conception imaginé pour répondre au problème. On étudie ensuite l'impact de la solution adoptée. Enfin, on dresse une liste des éléments à prendre en compte afin de répondre de la meilleure manière possible aux questions posées dans la section 2.4.

Sommaire

4.1 Flot de conception d'un circuit sur cible reconfigurable

Le flot de conception d'un circuit constitue l'ensemble des outils et étapes qu'un développeur d'applications doit utiliser ou suivre afin d'obtenir un FPGA mettant en œuvre l'application qu'il désire. On distingue actuellement deux approches : l'une « classique » et l'autre « de haut niveau ». Les paragraphes suivants les décrivent et les différencient.

4.1.1 Flot de conception « classique »

Afin d'obtenir un circuit fonctionnel sur un FPGA, il est nécessaire de passer par un flot de conception comprenant plusieurs étapes (Figure 4.1). L'ensemble des étapes de construction du circuit pour un FPGA est appelé synthèse. Ce flot est plus complexe, du point de vue de l'utilisateur, que le flot de production d'un logiciel par exemple. De manière générale, un développeur d'application décrit l'architecture qu'il désire dans un langage de description matérielle (langage dit HDL) tel que VHDL, Verilog ou encore SystemVerilog. Ces langages ont été spécifiquement développés pour décrire des architectures de circuits. Le circuit décrit par un code source HDL subit une phase de synthèse logique dont l'objet est de traduire le circuit dans une représentation *netlist*, c'est-à-dire mettant en exergue les liens (*net* en anglais) entre les différents éléments logiques du circuit. Cette représentation *netlist* est adaptée à la manipulation. La suite du flot se compose donc d'une étape de correspondance visant à traduire les éléments de la *netlist* en des composants disponibles dans le FPGA ciblé, typiquement des LUT et des FF. Les deux dernières étapes de la synthèse, le placement et le routage, visent à sélectionner sur le FPGA les composants qui mettent en œuvre le circuit et à les connecter. Cette étape est algorithmiquement très complexe. Enfin, une fois le flux binaire de configuration (*bitstream*) obtenu, il ne reste plus qu'à le charger dans le FPGA.

Étant donné l'utilisation obligatoire d'un langage de type HDL comme point d'entrée de ce flot, le développeur d'application doit posséder de solides compétences en matériel afin de générer des circuits performants. De plus, traduire dans un langage de description matérielle un algorithme n'est pas une étape anodine.

4.1.2 Flot de conception HLS

Afin de réduire le temps de développement d'un circuit et d'éviter des étapes de traduction algorithme/description matérielle parfois fastidieuses, une autre couche d'abstraction est utilisable. Le point d'entrée d'un flot de haut niveau (ou HLS pour *High-Level Synthesis*), visible en Figure 4.2, n'est plus une description matérielle mais une description algorithmique dans un langage dit de haut niveau (C, C++ ...), offrant des abstractions facilitant le développement. À la sortie de l'étape de synthèse de haut niveau, le développeur obtient un ou plusieurs fichiers décrivant son application dans un langage HDL.

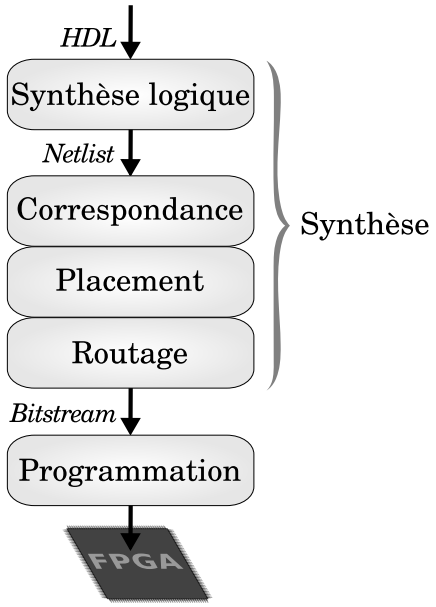


FIGURE 4.1 – Du code source au FPGA : flot de conception « classique » pour FPGA partant d'une représentation HDL.

Un des intérêts majeurs de ce type de flot est que le développeur n'a plus à se focaliser sur les détails de la mise en œuvre de son application tels que : combien d'additionneurs sont nécessaires afin de réaliser le maximum d'opérations en parallèle, combien de registres sont nécessaires, etc. D'un autre côté, un circuit décrit en HDL est plus finement modifiable par un développeur compétent et donc potentiellement plus performant.

Le livre de référence écrit par COUSSY et al. [Cou+10] décrit plus en détail le flot de conception de haut niveau et ses spécificités.

4.1.3 Flot de conception modifié

C'est dans la continuité d'un flot de HLS que les travaux présentés dans ce manuscrit s'inscrivent. Il est nécessaire de détailler les raisons de ce choix majeur impactant l'intégralité des développements proposés. C'est pourquoi ce paragraphe présente notre flot de conception et introduit notre méthode basée sur les deux étapes que sont la sélection des points de sauvegarde et l'insertion des chaînes de sérialisation, afin de souligner dans quel contexte la méthode présentée s'applique.

L'outil de HLS dans son fonctionnement interne, visible sur la partie droite de la Figure 4.3, associe trois étapes principales. La première est la compilation. Elle sert à transformer le code d'entrée de l'outil de HLS en une représentation intermédiaire (IR, pour *Intermediate Representation*) adaptée aux besoins de l'outil. Suit une étape lors de laquelle le circuit dans sa représentation intermédiaire subit des transformations. Celles-ci ont pour but d'adapter l'algorithme initial afin que sa mise en œuvre soit le plus efficace possible sur la cible. Par exemple, des transformations telles que le déroulement d'une boucle ou l'ajout d'opérateurs

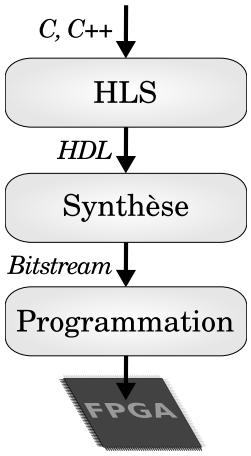


FIGURE 4.2 – Du code source au FPGA : flot de haut niveau pour FPGA partant d'une représentation C, C++.

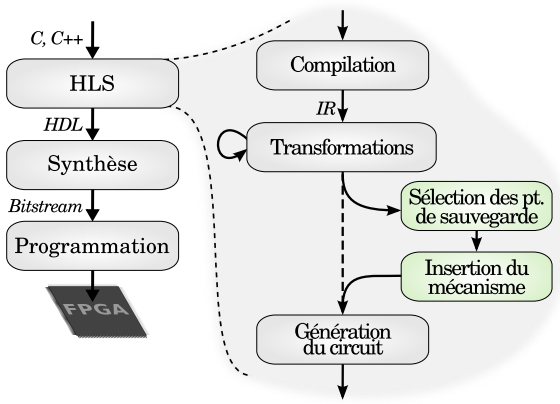


FIGURE 4.3 – La méthode dans le flot de haut niveau : détail des étapes proposées.

permettant un calcul complexe en un cycle sont possibles. Les transformations ont un cout et un outil de HLS peut adopter différentes stratégies en vue de choisir s'il faut les appliquer ou non. Dans la plupart des outils de HLS actuels, cette étape visant à adapter le circuit est manuelle, pilotée par le développeur. Pour d'autres, c'est une étape automatique (cf. section 7.1.1, page 78). Enfin, une fois la représentation intermédiaire transformée, la dernière étape génère le circuit (à l'aide des sous-étapes d'ordonnancement et d'assignation non développées ici) dans sa représentation HDL. La généricité de ce flot n'est pas garantie : tous les logiciels de HLS ne suivent pas le même schéma, mais le principe est généralement comparable.

La méthode proposée est composée de deux étapes qui viennent s'ajouter au flot de HLS. La génération d'un circuit commutable respectant les critères donnés précédemment est donc assurée par l'association d'une étape de sélection des points de sauvegarde et d'une étape de construction du mécanisme d'extraction de contexte. Ces deux étapes, bien que distinctes, sont fortement couplées. La description de

l'étape de sélection des points de sauvegarde ainsi que son principe seront décrits dans le chapitre 5. La description de l'étape d'insertion du mécanisme et toutes les problématiques liées seront décrites dans le chapitre 6.

4.2 Impact de l'utilisation de la HLS

Les raisons pour lesquelles le flot de génération de circuit commutable est introduit au cœur d'un flot de synthèse de haut niveau sont multiples. Cette section détaille les impacts, positifs ou négatifs, d'une méthode de haut niveau.

4.2.1 Représentation intermédiaire

La représentation intermédiaire d'un circuit dans un outil de HLS est généralement un HTG ou l'un de ses dérivés. Selon GIRKAR et al. [Gir+94], le HTG est particulièrement adapté pour l'optimisation d'application ainsi que la génération de code. Nombre d'informations présentes dans l'IR d'une application subissant le flot de conception proposé sont indispensables aux deux étapes présentées précédemment. Notamment la durée de vie des variables, qui est utilisée par l'étape de sélection des points de sauvegarde, est une donnée qu'il est complexe de retrouver dans la représentation intermédiaire (*netlist*) d'un outil de synthèse par exemple.

Un HTG est aussi particulièrement efficace lorsqu'il s'agit de manipuler la machine d'état d'un circuit. Il est en effet possible de manipuler l'IR afin d'ajouter des structures de contrôle et des états, ce qui n'est pas possible à partir d'une représentation HDL.

C'est sa proximité avec l'algorithme d'entrée donné par le développeur qui permet à l'IR de l'outil de HLS de posséder des informations qui sont occultées dans la suite du flot de conception.

4.2.2 Automatisation

L'ajout d'un mécanisme d'extraction (comme une chaîne de sérialisation par exemple) est généralement effectué au niveau HDL ou *netlist* et à la main. Des outils d'automatisation existent mais ils sont situés après la synthèse, ce qui peut leur procurer des avantages (optimisations liées à la technologie cible ou bas niveau) mais aussi des inconvénients (cités dans le paragraphe précédent). Ajouter des étapes dans un flot de synthèse de haut niveau automatise le processus et le rend générique.

4.2.3 Indépendance des circuits produits

La sortie d'un flot de HLS est un ensemble de fichiers en langage HDL. Pour la suite du flot de conception, l'utilisateur peut choisir sa chaîne d'outils en fonction du FPGA cible. L'avantage très conséquent est que la sortie de l'outil de HLS est portable sur théoriquement toutes les architectures de FPGA. Cette caractéristique

est primordiale dans l'optique de produire un mécanisme qui se veuille portable. En pratique, il peut être nécessaire d'adopter un style de programmation différent en fonction des outils utilisés ensuite. Ce type de comportement est succinctement étudié dans la suite du manuscrit.

4.2.4 Circuits générés

Le flot de HLS utilise des mécanismes internes afin de produire des circuits adaptés à une cible matérielle, l'objectif étant généralement de supplanter une implémentation logicielle du même algorithme et donc de s'affranchir d'une exécution trop séquentielle. Un architecte matériel, lorsqu'il réalise la description matérielle d'un circuit en HDL, doit penser à sa mise en œuvre matérielle et donc procède naturellement à des améliorations en vue d'obtenir le circuit le plus performant possible selon ses critères (latence, fréquence de fonctionnement, consommation). Un outil de HLS procède aux même types d'optimisations mais les résultats ne sont pas forcément à la hauteur d'une implémentation manuelle. De plus l'ajout d'un mécanisme d'extraction à haut niveau n'est pas sans inconvénient : suivant la suite d'outil générant finalement le circuit, les performances obtenues seront variables. En définitive, prévoir les performances du circuit et l'impact de l'ajout du mécanisme de commutation de contexte n'est pas faisable de manière précise à haut niveau à l'heure actuelle.

4.3 Contraintes et objectifs d'optimisation

Ce paragraphe a pour but de rendre plus formelles les questions posées dans la section 2.4 maintenant qu'une vision globale de la méthode a été dépeinte. Nous nous poserons les questions suivantes : l'ajout de la fonctionnalité du changement de contexte est couteuse, quels sont ces couts et quels sont les objectifs d'optimisation qu'on y associera ? quelles sont les contraintes que doit respecter un circuit produit par notre flot ?

Le Tableau 4.1, contenant tous les sous-problèmes rencontrés au cours de ce manuscrit, est décrit dans les deux paragraphes qui suivent.

4.3.1 Contraintes

La première partie du Tableau 4.1 est consacrée aux contraintes qu'il est nécessaire de fixer afin que la méthode soit fonctionnelle. On peut considérer que ce sont des caractéristiques nécessaires mais non suffisantes. Certaines de ces contraintes ont déjà une solution du fait même de l'utilisation d'un flot de synthèse de haut niveau.

La méthode proposée se veut devenir une brique de base technologique pour de nombreuses applications, déjà mentionnées dans la section 2.3.2. Il est donc primordial de proposer un flot qui fonctionne de manière autonome, automatique, prévisible et reproductible. C'est en développant la méthode au sein d'un flot de

	Type	Solution
Contraintes	Méthode automatique	flot HLS
	Portabilité du circuit produit	flot HLS
	Latence de commutation bornée	pt. de sauv.
Optimisations	Impact sur le système complet (communication principalement)	pt. de sauv. + mécanisme
	Surcout matériel du mécanisme d'extraction	pt. de sauv. + mécanisme
	Impact sur les performances du circuit	pt. de sauv. + mécanisme

TABLE 4.1 – Contraintes et objectifs d'optimisation de la méthode.

haut niveau que cette contrainte est respectée. L'automatisation de la construction de circuits commutables est obtenue grâce à un outil de HLS. Toutefois, il aurait été possible de générer automatiquement un circuit commutable à d'autres niveaux d'abstraction, ce qui par ailleurs existe déjà dans la littérature. Mais le flot de haut niveau offre d'autres avantages.

En effet, une autre contrainte concerne la portabilité du mécanisme et du circuit obtenu. Un des objectifs est d'offrir la fonctionnalité de changement de contexte à des circuits pouvant être émulés sur des FPGA de familles voire de constructeurs différents. Le flot de HLS produisant un circuit décrit en HDL, celui-ci peut théoriquement cibler n'importe quel FPGA pourvu de suffisamment de ressources matérielles.

Enfin, la dernière contrainte concerne le temps de réponse d'un circuit commutable généré à l'aide de la méthode. Pour adapter un mécanisme de changement de contexte tel qu'il est mis en œuvre dans les SE actuels, il faut rendre borné le temps de commutation (comme vu dans la section 3.4). L'étape de sélection des points de sauvegarde a pour principal objet d'être à même de produire des circuits dotés de cette caractéristique.

4.3.2 Objectifs d'optimisation

La seconde moitié du Tableau 4.1 est consacrée aux objectifs d'optimisation. C'est la qualité des optimisations fournies qui rendra la méthode proposée viable ou acceptable.

Dans un premier temps, il faut répondre à la question de l'impact sur le système complet de la commutation d'un circuit. Excepté les aménagements qu'il faut apporter au bloc de contrôle (SE par exemple) pour gérer les circuits commutables, l'impact d'une commutation sur le système se résume à l'impact qu'aura l'échange des données sur le sous-système de communication. Afin de réduire les congestions dans ce sous-système, il est important de limiter la quantité de données qui transitent par celui-ci. Pour ce faire, c'est sur la taille du contexte des tâches matérielles qu'il est possible d'agir, au niveau de la méthode proposée. Les deux étapes de la

méthode auront pour tâche de minimiser la quantité de données à extraire afin de procéder à un changement de contexte. D'autres aspects systémiques impactant les communications entrent en jeu (la fréquence des commutations par exemple ou encore la gestion des données en transit lors des commutations) mais ne sont pas du ressort de la brique de base technologique proposée dans ce manuscrit.

Les deux derniers objectifs d'optimisation sont liés à l'impact de l'ajout d'un mécanisme d'extraction de contexte au cœur d'un circuit. Il est souhaitable de rendre le mécanisme le plus transparent possible pour, d'une part, que le surcout matériel lié à son ajout soit faible, et d'autre part, que son impact sur les performances du circuit soit réduit. En définitive, ajouter le mécanisme d'extraction de contexte doit se faire avec un surcout minimal en terme de LUT, FF et autres ressources de routage. En outre, son addition ne doit pas dégrader les performances initiales du circuit c'est-à-dire sa fréquence de fonctionnement et sa consommation. Bien que l'étape d'insertion du mécanisme soit concernée au premier chef, nous verrons dans la suite du manuscrit que l'étape de sélection des points de sauvegarde a aussi un rôle à jouer dans la minimisation de ces deux objectifs.

Chapitre 5

Sélection des points de sauvegarde

La solution générale pour le changement de contexte proposée dans ce travail de thèse se décompose en deux étapes principales : la sélection de points de sauvegarde et l'insertion du mécanisme d'extraction. Ce chapitre présente la première étape du flot de conception, qui consiste à analyser une application donnée afin d'en extraire des points de sauvegarde.

Dans un premier temps, après un bref historique sur la notion de point de sauvegarde, on présente l'idée générale de notre proposition. Ensuite, afin de résoudre le problème posé, nous utilisons une formalisation mathématique. Une résolution de ce problème est enfin proposée.

Sommaire

5.1 Présentation générale

Après une courte définition de la représentation d'une tâche matérielle, ce paragraphe présente l'état de l'art sur les points de sauvegarde ainsi que l'idée générale de notre méthode de sélection de points de sauvegarde.

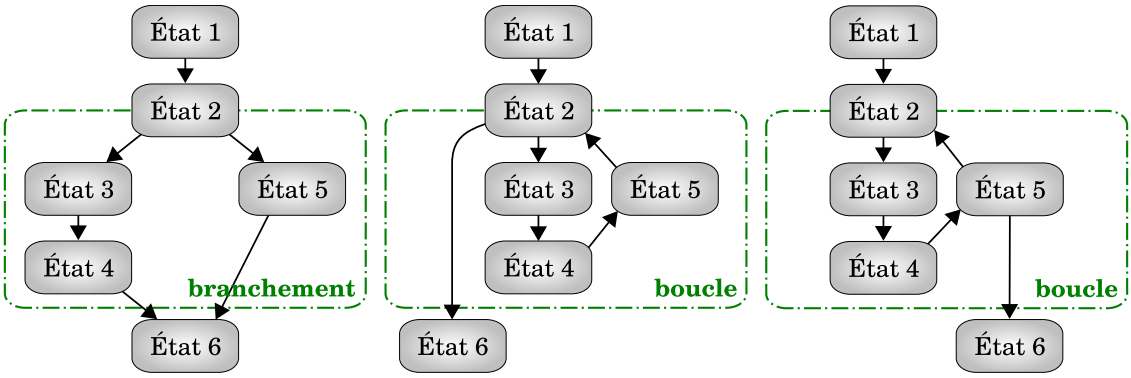
5.1.1 Représentation d'une tâche matérielle

Les tâches matérielles ont une représentation classique différentiant la machine d'état des chemins de données. Cette représentation communément utilisée est aussi appelée *FSM/Datapath*, pour *Finite State Machine and Datapath*. Elle consiste à séparer la partie qui contrôle le comportement du circuit (la FSM ou la machine d'état) de la partie qui réalise effectivement les calculs bruts. La Figure 5.1 illustre différentes machines d'états représentant des flots de contrôle distincts : branchement conditionnel (Figure 5.1a), boucle de type *for* ou *while* (Figure 5.1b) et boucle de type *do-while* (Figure 5.1c). À chaque état dans la machine d'état correspond une action réalisée par la partie chemin de données. Par exemple, dans la Figure 5.1a, l'état 1 pourrait correspondre au calcul d'une valeur dans le chemin de données avec stockage dans un registre. L'état 2, réalisant le branchement conditionnel, pourrait quand à lui correspondre à la comparaison de la valeur précédemment calculée à une constante. En fonction du résultat de la comparaison, la tâche matérielle procèderait ensuite à la réalisation de l'état 3 ou à celle de l'état 5.

On parle donc dans ce manuscrit de l'état d'une tâche matérielle par rapport à l'état dans lequel se trouve la machine d'état contrôlant la tâche.

5.1.2 Travaux connexes

La notion de point de sauvegarde, ou *checkpoint*, est généralement utilisée dans le cadre de la tolérance aux fautes [Eln+02 ; Egw+13]. Ce domaine de recherche regorge de références : il remonte aux origines de la recherche en informatique. L'insertion de points de sauvegarde ajoute une qualité de tolérance aux fautes à un système. Ce type d'approche consiste à sauvegarder régulièrement l'état de ce système afin de pouvoir restaurer cet état en cas de fautes durant la suite de son exécution. Dans ce manuscrit, il est plus approprié de s'intéresser à la sélection des points de sauvegarde dans le cadre des architectures matérielles, donc s'appliquant à des tâches matérielles. En logiciel, rencontrer un point de sauvegarde implique un enregistrement de données comme le compteur ordinal et les valeurs des registres du processeur par exemple. Cette technique est focalisée sur la sauvegarde d'une architecture de processeur en particulier, cette sauvegarde étant généralement pilotée par un logiciel. En matériel, la sauvegarde concerne l'intégralité d'un circuit et *a priori*, aucun point commun n'existe entre deux tâches matérielles adoptant cette technique, alors que pour deux tâches logicielles distinctes, ce sont les mêmes registres du processeur qui seront sauvegardés. KOCH et al. [Koc+07] et KOCH [Koc12] proposent d'appliquer les principes d'utilisation des points de sauvegarde aux FPGA et présentent donc des solutions techniques permettant



(a) Branchement (b) Boucle de type *for/while* (c) Boucle de type *do-while*

FIGURE 5.1 – Exemples de machines d'état.

d'y arriver, c'est-à-dire d'extraire le contexte d'une tâche matérielle. À ces fins, ils présentent trois mécanismes dont les deux plus intéressants sont les mécanismes de *memory-map*, permettant de lire et écrire dans une mémoire externe depuis un registre, et de *scan-chain* permettant l'extraction en série de tous les registres. Grâce à ces mécanismes, il est possible de faire du *checkpointing* sur une tâche matérielle en sauvegardant son contexte pour le relancer après une faute. Dans le cas général, le *checkpoiting* est utilisé pour faire une simple sauvegarde de la tâche matérielle s'exécutant sur FPGA [Lan+02]. Après détection d'une erreur, la tâche matérielle est relancée en utilisant un précédent point de sauvegarde. Dans des cas plus complexes, il est possible d'utiliser des points de sauvegarde afin de réaliser de la mitigation d'erreurs *soft*, c'est-à-dire éphémères [Asa+05]. Des techniques moins couteuses existent lorsqu'il s'agit de traiter des processeurs émulés dans un FPGA (*soft processors*) [Reo+09 ; Pha+13]. Enfin, SCHMIDT et al. [Sch+11] proposent une infrastructure de gestion permettant le contrôle de plusieurs FPGA ou zones reconfigurables accueillant des tâches soumises à une sauvegarde régulière.

5.1.3 Définition d'un point de sauvegarde

Dans tous les cas précédents, où le point de sauvegarde est utilisé en tant que technique de tolérance aux fautes, le contexte de l'application est sauvegardé à chaque fois qu'un point de sauvegarde est atteint. Lorsqu'une faute est introduite dans le circuit, un protocole est responsable de recharger le dernier état sauvegardé afin de restaurer l'application dans un état stable. Il est possible d'utiliser un mécanisme de ce type afin d'obtenir un changement de contexte. Par exemple, le contexte d'une tâche matérielle peut être sauvé lorsqu'un point de sauvegarde est rencontré et que le système requiert un changement de contexte. La restauration de contexte peut alors avoir lieu à l'aide du contexte enregistré au point de sauvegarde rencontré, et la tâche peut repartir de ce point.

Dans ce manuscrit, on appellera **points de sauvegarde** les états d'une tâche matérielle lors desquels un changement de contexte est autorisé. À partir de cette définition, on peut établir un scénario typique de changement de contexte. Consi-

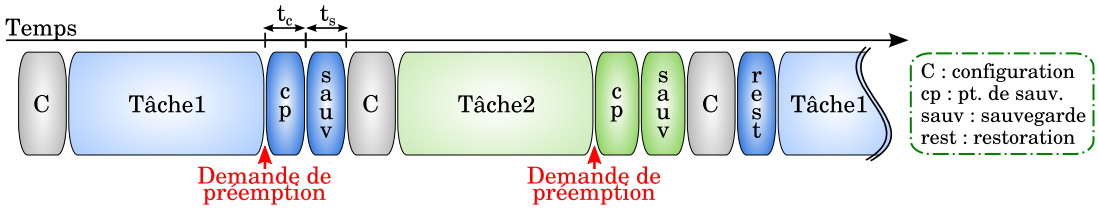


FIGURE 5.2 – Scénario d'un changement de contexte entre deux tâches.

dérons deux tâches matérielles devant s'exécuter sur la partie reconfigurable du système typique. Ce scénario est illustré par la Figure 5.2. Il commence par une configuration de la partie reconfigurable avec le circuit correspondant à la tâche 1, qui démarre ensuite. À un instant donné, le système de contrôle émet une demande de préemption à destination du FPGA. La tâche entre donc dans une phase de recherche de point de sauvegarde : elle continue son exécution jusqu'à ce qu'un point de sauvegarde soit trouvé (bloc « cp » sur la figure). Le point de sauvegarde atteint, la tâche procède à l'extraction, ou la sauvegarde de son contexte (bloc « sauv »). À la fin de cette étape, la région reconfigurable est libérable et donc est configurée avec la tâche 2, qui peut ensuite démarrer. Un changement de contexte vient d'avoir lieu. Une nouvelle demande de préemption, visant à restaurer la première tâche, arrive pendant l'exécution de la seconde tâche. C'est après avoir atteint un point de sauvegarde et extrait son contexte que la tâche 2 rend le FPGA. Celui-ci est de nouveau configuré avec la tâche 1, dont on restaure le contexte (bloc « rest ») précédemment sauvegardé avant redémarrage. Deux grandeurs sont introduites ici : t_c, ou le temps nécessaire afin d'atteindre un point de sauvegarde, ainsi que t_s, ou le temps nécessaire afin de sauvegarder le contexte de la tâche une fois que celle-ci a atteint un point de sauvegarde. À l'aide de ces grandeurs, on peut traduire mathématiquement la contrainte de latence. Soit t_{lat} la latence maximale d'extraction de contexte admissible par le système. Alors, une tâche matérielle commutable doit respecter la contrainte $t_c + t_s < t_{lat}$.

5.1.4 Application à la problématique

Pour rendre une tâche matérielle commutable, il faut être capable d'extraire le contexte de cette tâche lorsqu'elle s'exécute. Le contexte d'une tâche matérielle, défini en 3.3, consiste en l'architecture de la tâche ainsi que son « histoire », c'est-à-dire le contenu des éléments mémorisants qui sont susceptibles d'avoir évolués au cours de l'exécution de la tâche. Partons de l'hypothèse que le système générique considéré garde stocké en mémoire le flux binaire de configuration relatif à la tâche. Il n'est donc pas nécessaire de sauvegarder l'architecture de la tâche dont on cherche à extraire le contexte étant donné que celle-ci est encore disponible en mémoire. On réduit donc le contexte d'une tâche matérielle à l'état de ses éléments mémorisants. Or, tous les éléments mémorisants ne font pas, en réalité, partie du contexte qu'il est nécessaire d'extraire. En effet, une variable, tant qu'elle n'a pas d'utilité ou si elle n'est plus utilisée après un état particulier, n'est pas nécessaire afin de redémarrer la tâche. Une telle variable ne fait donc pas partie du contexte à chaque instant. Par

exemple, un itérateur de boucle n'est pas utile avant sa boucle d'appartenance et n'est plus utile une fois celle-ci terminée. Sur la Figure 5.1b, un tel itérateur aurait besoin d'être extrait si le changement de contexte avait lieu dans les états 2, 3, 4 ou 5, mais non dans les états 1 et 6. Cette analyse est appelée « analyse de la durée de vie ». En procédant à une analyse de la durée de vie de toutes les variables d'une tâche matérielle, il est possible de définir dans chaque état quelles sont les variables à extraire. La quantité de données à extraire dans chacun des états d'une tâche est changeante. Or réduire la quantité de données à extraire va dans le sens des objectifs d'optimisation fixés. Moins il y a de variables à équiper, moins le surcout matériel est important, moins les contraintes de routage sont élevées et donc les performances du circuit réduites et enfin moins il y a de données à faire transiter dans le système. L'idée est donc de sélectionner un certain nombre d'états, des points de sauvegarde, qui assureront d'une part que le temps pour réaliser un changement de contexte est limité (contrainte principale restante) et d'autre part, que la quantité de variables à extraire est réduite.

5.2 Formalisation mathématique

D'après le scénario d'un changement de contexte présenté précédemment, lorsqu'une demande de préemption est initiée par le bloc de contrôle, la tâche matérielle est dans un état quelconque. Un point de sauvegarde doit alors être atteint avant de procéder à l'extraction du contexte de la tâche en cours. Les temps caractéristiques correspondants à ces deux étapes sont respectivement t_c et t_s. Tous les états par lesquels la tâche est passée avant d'atteindre un point de sauvegarde sont dits « couverts » par ce point de sauvegarde. C'est-à-dire que si une demande de préemption émane du bloc de contrôle alors que la tâche est dans un de ces états, ce point de sauvegarde est susceptible d'être utilisé afin de procéder à l'extraction de contexte. Pour rappel, l'étape de sélection des points de sauvegarde est censée répondre à la contrainte de latence imposée à la tâche matérielle à savoir : $t_c + t_s < t_{lat}$. À partir de cette contrainte ainsi que des objectifs d'optimisation, nous formalisons mathématiquement le problème rencontré.

Soit n le nombre d'états d'une tâche matérielle donnée. Un vecteur d'entiers naturels de dimension n ($\in \mathbb{N}^n$) peut représenter un ensemble d'états en assignant une composante par état. Un état est considéré comme faisant partie de l'ensemble si sa composante vaut 1. Si sa composante vaut 0, il ne fait pas partie de l'ensemble.

Soit $x \in \mathbb{N}^n$ un vecteur représentant un ensemble de points de sauvegarde parmi les états de la tâche matérielle. Soit $A = (a_{ij})_{1 \leq i,j \leq n}$, une matrice d'entiers naturels de dimension $n \times n$ ($A \in M_n(\mathbb{N})$) représentant la couverture de chacun des états. Le coefficient booléen a_{ij} vaut 1 lorsque l'état j couvre l'état i. On a donc $Ax \in M_{n,1}(\mathbb{N})$, un vecteur d'entiers naturels de dimension n illustrant combien de points de sauvegarde couvrent chaque état. La matrice de couverture A est construite en fonction de t_{lat} et de la nature de la tâche. Généralement, plus t_{lat} est grand, plus la couverture de chaque état est grande. Afin d'être un ensemble de points de sauvegarde valide, x doit assurer que pour chaque état de la tâche, le

changement de contexte soit possible, c'est-à-dire que chaque état soit couvert par au moins un point de sauvegarde. Cette contrainte est formalisée de la manière suivante :

$$A.x \geq \mathbf{1}$$

où **1** est un vecteur d'entiers naturels de dimension n dont toutes les composantes valent 1.

À cette contrainte on associe un objectif d'optimisation. On part de l'hypothèse que les trois objectifs de minimisation présentés en Tableau 4.1 ont une résolution potentielle commune qui correspond à la minimisation de la taille du contexte. Réduire la taille du contexte conduit à réduire :

— la quantité de données transitant dans le système ;
— le nombre d'éléments mémorisants à équiper ;
— les contraintes de routage.

On estime donc que le surcout général du mécanisme est proportionnel à la taille en bit du contexte à extraire. Ceci conduit à définir le cout d'un point de sauvegarde. Soit $c_i \in \mathbb{R}+$ le cout de l'état i en tant que point de sauvegarde. D'après les remarques précédentes, le cout d'un point de sauvegarde est proportionnel à la quantité de données qu'il possède dans son contexte d'exécution. On définit donc le cout d'un point de sauvegarde comme étant la taille en bit de son contexte propre, c'est-à-dire le volume total des variables vivantes dans cet état. Les données étant des éléments mémorisants, plus un point de sauvegarde contient d'éléments mémorisants, plus son cout est élevé. Une première estimation du cout du mécanisme complet consiste à additionner le cout de tous les points de sauvegarde finalement sélectionnés. D'où l'équation de minimisation suivante :

$$\min \sum_{j=0}^{n-1} c_j x_j$$

Ce type de problème mathématique se nomme « problème de couverture par ensemble » et a été décrit par MINOUX [Min83]. C'est un problème NP-complet.

Cette première approche n'est en réalité pas tout à fait satisfaisante. En effet, l'équation de minimisation est plus complexe car le cout total du mécanisme n'est pas égal à la somme des couts des points de sauvegarde sélectionnés. Une version plus complète doit tenir compte du cout induit par la sélection d'un point de sauvegarde sur le cout des autres, l'ajout dans le mécanisme d'extraction d'un élément mémorisant n'étant pas valable que pour un seul point de sauvegarde. Ainsi, une fois un point de sauvegarde sélectionné, les points de sauvegarde contenant tout ou partie des éléments mémorisants en commun avec celui-ci voient leur cout évoluer. En d'autres termes, si un élément mémorisant E est présent dans plusieurs points de sauvegarde de x, alors le cout de E ne doit être compté qu'une fois. Il faut donc passer d'une équation de minimisation linéaire à une équation polynomiale afin d'estimer le cout de x correctement. L'équation de minimisation devient alors :

$$\min \sum_{j=0}^{2^n-1} c'_j x'_j$$

où j est le nième bit d'index d'un ensemble d'états placés dans un ordre naturel (par exemple, pour $j = 9 = 2^0 + 2^3$, l'ensemble est constitué de 0 et 3), c'_j est le cout de l'ensemble j (c'est-à-dire la taille des éléments mémorisants contenus dans les

contextes des points de sauvegarde de j), x'_j est un booléen étant vrai si l'intersection entre les ensembles d'états indexés par j et représentés par x n'est pas nulle.

On remarque que le nombre de termes de cette équation croient exponentiellement en fonction de n. Pour parvenir à trouver une solution, on utilise une heuristique étant donné qu'une résolution exacte n'est pas envisageable.

5.3 Résolution du problème

Deux étapes sont nécessaires afin de trouver une solution. La première consiste à identifier la matrice A à l'aide d'une analyse de la représentation intermédiaire (IR) du circuit. La seconde utilise cette matrice A ainsi que des heuristiques afin de sélectionner un ensemble de points de sauvegarde x satisfaisant.

5.3.1 Analyse du circuit

La première étape de résolution consiste à identifier A. Une analyse de la représentation intermédiaire du circuit et plus particulièrement de sa machine d'état permet d'identifier cette matrice.

La couverture de l'état j, qu'on appelle S_j, correspond à l'ensemble des états que j peut couvrir en tant que point de sauvegarde. S_j est un sous-ensemble de $\{1, \cdots, n\}$ tel que $S_j = \{i | a_{ij} = 1\}$. S_j correspond en fait à la liste des états dont la composante vaut 1 dans la $j$ème colonne de A. Pratiquement, cela signifie que l'état j peut être atteint ainsi que le contexte de la tâche dans cet état extrait dans une durée inférieure à t_{lat} depuis tous les états de S_j. L'ensemble des S_j pour $j \in \{1, \cdots, n\}$, c'est-à-dire A, est identifié à l'aide d'un algorithme récursif (voir l'Algorithme 1). Celui-ci prend en entrée l'IR de la tâche matérielle considérée.

Une boucle principale s'exécute sur tous les états de la tâche afin d'identifier les S_j correspondants. Si le contexte associé à l'état j peut être extrait dans un intervalle de temps inférieur à t_{lat}, alors on procède à l'identification de S_j. Dans le cas contraire, j est retiré de l'ensemble des points de sauvegarde potentiels (ligne 5). On procède ensuite récursivement à l'identification de la couverture de l'état j en utilisant la procédure *analyse_couverture* définie à la ligne 16. Chaque état qui est potentiellement un état menant à j est analysé pour déterminer si sa couverture est possible. Cette analyse relativement complexe est décrite dans un paragraphe suivant. Si l'état précédent considéré a déjà été visité, plusieurs chemins pouvant y mener, on met à jour t_r, le temps restant afin de continuer l'analyse, avec la pire des valeurs rencontrées. Si l'état j couvre l'état k, alors k est ajouté à S_j. La recherche continue avec $t_r = t_{pire} - 1$ seulement s'il reste du temps afin de couvrir des états. À la fin de cette procédure, tous les S_j sont identifiés et la matrice A est donc complète.

La description textuelle de la ligne 22 qui suit est intentionnelle, une description en pseudo-code n'étant pas envisageable car relativement complexe. La principale difficulté de cette étape est de répondre à la question suivante : l'état j couvre-t-il l'état k? La réponse n'est pas immédiate car les cas particuliers sont nombreux. Boucles, branchements conditionnels, sauts ou imbrications de tous ces cas com-

Algorithme 1 Identification de la matrice A

```
1: for j ∈ {0, ··· , n − 1} do                ▷ itération sur tous les états de la FSM
2:     est_cp ← vrai                          ▷ au départ, tout état est un point de sauvegarde
3:     CALCUL(t_s)                            ▷ temps de sauvegarde de j
4:     if t_s > t_lat then
5:         est_cp ← faux                      ▷ le contexte de j est trop grand pour s'extraire en t_lat
6:         S_j ← ∅                            ▷ l'état j ne couvre aucun état
7:         continue
8:     else
9:         t_r ← t_lat − t_s                  ▷ temps restant après prise en compte de t_lat (t_c)
10:        S_j ← {j}                          ▷ l'état j se couvre lui-même
11:    end if
12:    if t_r > 0 then
13:        ANALYSE_COUVERTURE(j,t_r)          ▷ récursion identifiant S_j
14:    end if
15: end for
16: procedure ANALYSE_COUVERTURE(j,t_r)
17:    static t_pire = ∞
18:    for all k, états précédant j do
19:        if t_r < t_pire then
20:            t_pire = t_r
21:        end if
22:        if j couvre k then                 ▷ en fonction de t_r et de la FSM
23:            t_r = t_pire − 1               ▷ décrément du temps restant
24:            S_j ← S_j ∪ {k}
25:            if t_r > 0 then                ▷ il reste du temps pour explorer la FSM
26:                ANALYSE_COUVERTURE(k,t_r)
27:            end if
28:        end if
29:    end for
30: end procedure
```

plexifient l'analyse. La Figure 5.3, qui concerne l'analyse de la couverture d'un branchement conditionnel, éclaircit un cas particulier. On considère que la traversée d'un état dure exactement un cycle d'horloge, $t_{lat} = 12$ cycles, et l'état 7 possède 8 bits de contexte propre à extraire. On a donc $t_s = 8$ cycles avec une chaîne de sérialisation d'un bit de large. Étant donné que la contrainte $t_c + t_s < t_{lat}$ doit être respectée, le temps restant afin d'atteindre l'état 7 est $t_l = t_{lat} - ts$, ce qui donne dans ce cas particulier $t_l = 4$ cycles d'horloge. On démarre donc la récursion afin d'identifier S_7 depuis l'état 7 et avec 4 cycles d'horloge disponibles. L'étape ① teste simplement l'état 6 avec le temps restant, étant donné qu'aucune structure de contrôle n'existe entre les états 6 et 7. Le résultat est positif (c'est-à-dire que l'état 7 couvre l'état 6) car le temps restant n'est pas nul. L'étape ②, qui correspond au premier appel récursif à *analyse_couverture* à la ligne 26, teste les états précédant

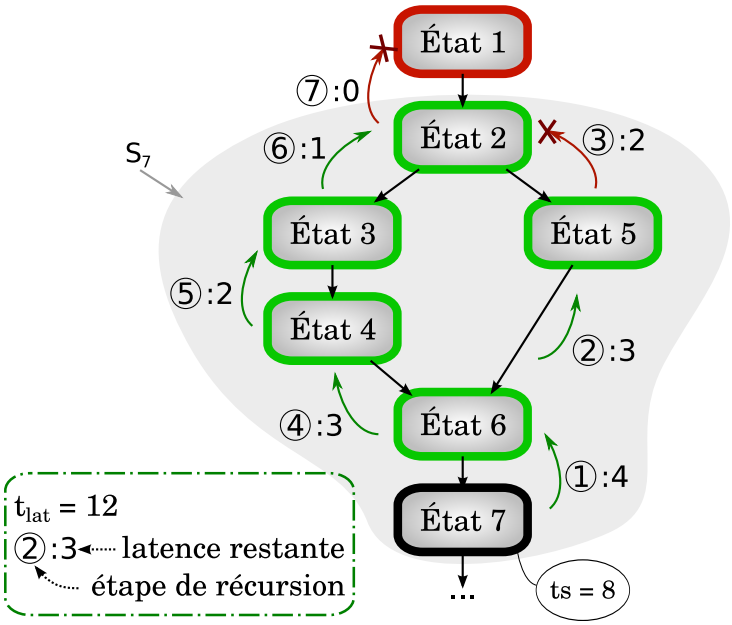


FIGURE 5.3 – Exemple de calcul de couverture pour l'état 7

l'état 6. Cette récursion démarre donc arbitrairement par l'état 5. Étant donné qu'il reste du temps, celui-ci est couvert. L'étape ③ ne peut pas être complétée car l'état 2 ne peut pas être considéré comme couvert si toutes les branches dont il est la source ne sont pas déjà couvertes. La récursion reprend donc à l'état 6 avec l'étape ④. La seconde branche du branchement conditionnel est couverte à son tour, l'étape ⑥ est complétée et l'état 2 est couvert. Par contre, l'état 1 n'est pas couvert car la latence restante est nulle pour l'étape ⑦. Le résultat de l'identification de la couverture de l'état 7 donne donc $S_7 = \{2, 3, 4, 5, 6, 7\}$.

L'analyse de la couverture peut gérer toutes les situations qu'il est possible de rencontrer dans l'IR d'une tâche quelconque. Par exemple, dans le cas d'une condition, un état appartenant à une branche ne peut couvrir que des états qui sont ses précédents et qui sont dans la même branche mais pas à l'extérieur. Dans la Figure 5.1a, c'est le cas de l'état 4 qui ne peut couvrir ni l'état 1 ni l'état 2 mais qui peut en revanche couvrir l'état 3. On peut aussi vérifier la couverture à travers le passage dans une boucle si tant est qu'on connaisse le nombre d'itérations de celle-ci. L'exemple de la Figure 5.1b montre que la question se pose pour l'état 6 par rapport aux états internes de la boucle (les états 2, 3, 4 et 5) ainsi que l'état précédant la boucle (l'état 1). Une réponse à cette question est possible seulement si l'IR dispose de l'information du nombre d'itérations au moment de la compilation du circuit. Si le nombre d'itérations d'une boucle n'est pas connu, on doit tenir compte du pire cas possible afin de respecter la contrainte de latence. La matrice A est donc une illustration du pire des cas en terme de couverture. Pour finir, toutes les situations croisées et imbriquées possibles sont prises en compte, même si l'analyse est potentiellement délicate.

5.3.2 Heuristique gloutonne

Étant donné la complexité du problème rencontré (NP-complet), on choisit une heuristique gloutonne afin de sélectionner l'ensemble des points de sauvegarde parmi les états. CHVATAL [Chv79] a présenté un algorithme glouton proposant une solution à un problème de couverture par ensemble avec un objectif d'optimisation linéaire. Cet algorithme a une complexité de $O(n^2)$. Il assure aussi que la solution obtenue est proche de la solution optimale d'un facteur prédéterminé. Cet algorithme repose sur le cout moyen d'un point de sauvegarde, défini comme étant $c_j/|S_j|$. Dans le cas présent, $|S_j|$ est le nombre d'états couverts par l'état j en tant que point de sauvegarde, et c_j est le cout de ce point de sauvegarde tel que défini en 5.2. Cet algorithme met itérativement la composante de x, x_j à 1 pour l'état j possédant le plus petit cout moyen, jusqu'à ce que tous les états de la tâche matérielle aient été couverts. À chaque itération, le cout moyen de tous les points de sauvegarde est mis à jour étant donné que les états déjà couverts par les points de sauvegarde déjà sélectionnés sont retirés du cout moyen (c'est-à-dire que le terme $|S_j|$ change). De plus on adapte l'heuristique à l'objectif d'optimisation polynomial. Les mises à jour du cout moyen tiennent donc aussi compte du fait que le cout de certains points de sauvegarde (c_j) restants est impacté. En effet, les points de sauvegarde choisis et ceux restants à choisir partagent certains couts, quand ils ont des éléments mémoire communs à extraire. Ces couts partagés sont représentés par le terme c'_j dans la formulation du problème d'optimisation polynomiale. Ces couts partagés doivent être pris en compte dans les itérations suivantes.

L'heuristique gloutonne présentée dans l'Algorithme 2 commence par une initialisation des différentes variables de travail (R, M et x^*). Ensuite, l'état k qui possède le cout moyen le plus faible est choisi, tant que tous les états ne sont pas couverts (ligne 4), et toutes les variables sont mises à jour. Il faut remarquer que le cout est calculé avec la fonction γ qui retourne la taille en bit d'un ensemble d'éléments mémorisants. L'ensemble M_k représente l'ensemble des éléments mémoire vivant pendant l'état k. Dans cet algorithme, M_k est une constante fournie par l'outil de HLS après la phase d'élaboration. Lors de la première itération $\gamma(M_k \backslash M)$ est équivalent à c_k, comme dans le cas du problème d'optimisation linéaire, étant donné que $M = \emptyset$. Le cout moyen est, lors des itérations suivantes, effectivement mis à jour avec les états qui ont déjà été couverts, l'ensemble R étant retiré de l'ensemble des états couverts par k. Cela correspond à l'expression $|S_k \backslash R|$.

La solution finalement retenue est x^*, le vecteur booléen de dimension n représentant un ensemble d'états. À la fin de l'application de l'heuristique, l'ensemble des points de sauvegarde obtenu vérifie la contrainte de la latence d'extraction inférieure à t_{lat}. De plus, cet ensemble favorise la réduction des surcouts liés à l'ajout de la fonctionnalité au cœur du circuit. Bien que non optimale par nature, les expériences présentées dans la suite du manuscrit montrent que la solution obtenue offre de bons résultats.

Algorithme 2 Heuristique gloutonne pour résoudre le problème de couverture par ensemble polynomial

$R \leftarrow \emptyset$ ▷ ensemble d'états couverts
$M \leftarrow \emptyset$ ▷ ensemble d'éléments mémoire couverts
$x_j^* \leftarrow 0$ pour $j \in \{0, \cdots, n-1\}$ ▷ vecteur décrivant l'ensemble des points de sauvegarde
while $|R| < n$ **do**
    Choisir k donnant $\frac{\gamma(M_k \backslash M)}{|S_k \backslash R|} = \min_{i=\{0 \ldots n-1\}} \frac{\gamma(M_i \backslash M)}{|S_i \backslash R|}$
    $x_k^* \leftarrow 1$
    $R \leftarrow R \cup S_k$
    $M \leftarrow M \cup M_k$
end while

Chapitre 6

Mécanisme d'extraction de contexte

UNe fois les points de sauvegarde sélectionnés, le mécanisme d'extraction de contexte peut être construit. Dans ce chapitre, on examine en premier lieu de quoi se compose le contexte d'une tâche matérielle. On présente ensuite les principes utilisés afin d'ajouter la fonctionnalité désirée au cœur de la tâche matérielle. Les conséquences de cet ajout matériel, c'est-à-dire la fragmentation induite par le mécanisme, sont expliquées. Pour finir, on présente un premier travail d'ouverture qui consiste à prendre en compte, à des degrés différents, la durée de vie des variables en mémoire, et on explique en quoi ce problème est complexe.

Sommaire

6.1 Contexte d'une tâche matérielle

Afin de réaliser un changement de contexte, ce dernier doit être sauvegardé et donc extrait d'une tâche en cours d'exécution. Les précédentes hypothèses, formulées dans la section 5.1.4, associent le contexte d'une tâche matérielle à l'ensemble des éléments mémorisants de cette tâche. Ce paragraphe décrit les deux catégories d'éléments mémorisants rencontrées dans une tâche matérielle et analyse leurs différences. L'impact de ces différences est ensuite étudié.

6.1.1 Composition du contexte

Jusqu'ici, aucune différence n'a été faite entre les éléments mémorisants. Plus particulièrement, la sélection des points de sauvegarde n'a pas tenu compte des différents types qu'il est possible de rencontrer dans un circuit synthétisé pour FPGA. Une des hypothèses est la suivante : « moins il y a d'éléments mémorisants à extraire, moins le mécanisme est couteux », le terme « couteux » contenant les trois objectifs d'optimisation précédemment cités (réduire l'impact sur le système, sur la surface du circuit et sur ses performances). Bien que cette proposition soit exacte, il convient de préciser que la nature des éléments mémorisants doit être prise en compte pour construire le mécanisme et que le surcout lié à l'équipement d'un élément mémorisant n'est pas fixe.

Il existe deux types d'éléments mémorisants dans les FPGA, dont les représentations schématiques sont proposées dans la Figure 6.1. Les premiers sont les registres, ou FF pour Flip-Flop, qui stockent une information binaire (cf. Figure 6.1a). Cette unité très simple est par exemple inférée lorsqu'un développeur utilise dans un code source de haut niveau un mot clé d'instanciation de variable. Une telle déclaration crée généralement un registre de 32 bits afin de stocker les futures valeurs assignées à la variable. Les seconds sont les mémoires (cf. Figure 6.1b). Une mémoire est un tableau de valeurs. Ces valeurs sont accessibles à l'aide d'un index ou d'une adresse. Une mémoire est donc un ensemble d'éléments mémorisants, donc chacun est accessible par adresse. Sur la figure, la largeur des mots stockés en mémoire est de L bits. Dans un FPGA, deux technologies peuvent être utilisées afin de mettre en œuvre une telle structure. On peut soit utiliser des LUT dotées de capacités d'écriture afin d'obtenir une LUTRAM qui agira comme une mémoire. On peut aussi utiliser une primitive donnée par le constructeur de FPGA, c'est-à-dire un bloc matériel présent dans la matrice d'interconnexion réalisant la fonction de mémoire. La plupart des constructeurs de FPGA fournissent ce type de bloc. En effet, un tel circuit ad-hoc, spécifiquement adapté aux contraintes liées aux mémoires, présente des performances intéressantes. Chez le constructeur Xilinx, ils sont appelés BRAM pour *Block RAM*.

La part du contexte constituée par les éléments mémorisants de type registre comparée aux mémoires est une métrique intéressante. Afin de présenter celle-ci sur un panel représentatif d'applications, nous utiliserons ici, ainsi que dans toute la suite du manuscrit, les applications de référence CHStone [Har+09] complétées par une idct et un décodeur mjpeg. Le Tableau 6.1 recense la quantité en bit de chaque

(a) Registre (b) Mémoire

FIGURE 6.1 – Représentation des deux types d'éléments mémorisants d'un FPGA.

	registres (bit)	mémoires (bit)	registres / total
adpcm	3545	12608	21.9%
aes	664	20608	3.1%
blowfish	632	34112	1.8%
dfadd	1766	0	100.0%
dfdiv	2155	0	100.0%
dfmul	885	0	100.0%
dfsin	4624	0	100.0%
gsm	752	3408	18.0%
idct	1000	2560	28.0%
jpeg	3433	371177	0.9 %
mips	320	3072	9.4%
mjpeg	2623	291600	0.8%
mpeg2	919	16768	5.1%
sha	510	3232	13.6%

TABLE 6.1 – Composition en registres et mémoires de différents contextes.

partie du contexte au sens large, c'est-à-dire la totalité des registres et mémoires sans analyse de la durée de vie. La dernière colonne de ce tableau présente la part du contexte relatif aux registres par rapport à la taille totale du contexte, c'est-à-dire la somme des contextes liés aux registres et aux mémoires. Hormis les cas des applications de calcul flottant (dfadd, dfdiv, dfmul et dfsin), toutes les applications possèdent des mémoires et des registres. De manière générale, la taille du contexte lié aux mémoires est plus grande que celle du contexte relatif aux registres.

6.1.2 Retour sur la sélection des points de sauvegarde

Lors de la sélection des points de sauvegarde, une analyse de la durée de vie des variables a été effectuée, mais uniquement sur le contexte relatif aux registres. L'analyse de la durée de vie des variables résidant en mémoire est un problème complexe, qu'on abordera en section 6.4. On ne fait donc pas d'analyse sur les variables en mémoire, ce qui peut sembler contradictoire compte tenu du volume plus important de données que les mémoires hébergent. Ce choix s'explique de la façon qui suit. Même si le contexte lié aux registres est globalement plus petit que le contexte lié aux mémoires, permettre l'extraction d'un registre est plus

couteux que de permettre l'extraction d'une variable en mémoire. En effet, dans cette mémoire, le mécanisme est partagé entre toutes les variables. À l'inverse, chaque bit appartenant au contexte lié aux registres aura un mécanisme d'extraction propre. L'ajout de ressources nécessaires à l'insertion de registres dans une chaîne de sérialisation est la principale composante du surcout matériel du mécanisme complet, comme les expérimentations tendent à démontrer (cf. Figure 7.5). En somme, l'analyse de la durée de vie des variables en mémoire est un levier agissant majoritairement sur un objectif d'optimisation : l'impact sur la communication au sein du système. En réduisant radicalement la taille du contexte à extraire, les surcouts de communication se trouvent réduits. Quant à elle, l'analyse de la durée de vie des variables situées dans des registres agit majoritairement sur les deux autres objectifs d'optimisation que sont la réduction du surcout matériel ainsi que le maintien des performances du circuit.

6.2 Équipement d'un circuit pour sa commutation

Afin de préparer un circuit à l'extraction de son contexte, il faut équiper sa machine d'état ainsi que son chemin de données. Pour ce dernier, les deux familles d'éléments mémorisants nécessitent des mécanismes d'extraction différents. Les deux paragraphes qui suivent présentent les deux paradigmes d'équipement mis en place ainsi que leurs variations. Enfin, la modification de la machine d'état nécessaire à la gestion des migrations est proposée.

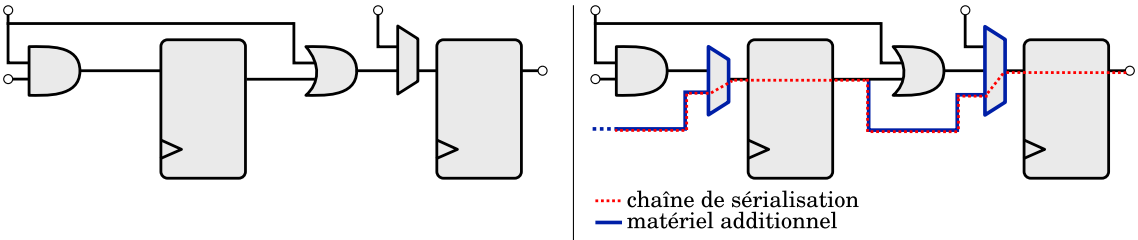
6.2.1 Registres

Équipement des registres

L'équipement d'un registre afin de permettre son extraction est une technique éprouvée. Afin d'accéder à la valeur d'un registre en évitant les composants qui y sont connectés, le plus simple est d'ajouter des multiplexeurs ou de rajouter des entrées à des multiplexeurs existants à ses interfaces (Figure 6.2). Dans le cas présenté, on crée un multiplexeur à l'entrée du premier registre et on ajoute une entrée à un multiplexeur existant à l'entrée du second registre. De cette manière, les registres sont chainés afin de permettre le transfert des données de l'entrée jusqu'à la sortie de la chaîne. Celle-ci est donc utilisée pour extraire les données des registres mais aussi pour les y réintroduire.

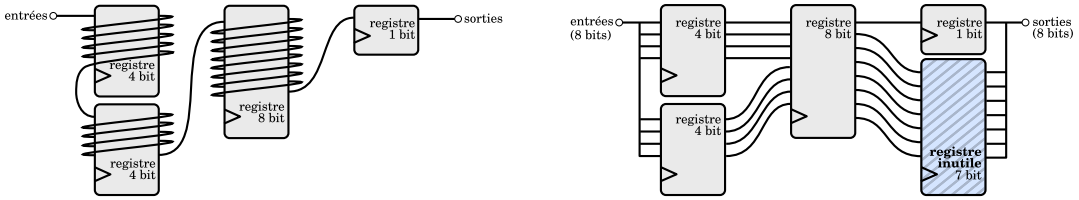
Chaînes simples et parallèles

Une difficulté reste toutefois à prendre en compte. Le cas précédent montre une chaîne entre deux registres de la même taille en bit. Or, la taille des registres d'une tâche matérielle est variable. Celle-ci peut varier entre 1 et 64 bits dans les applications de notre banc de test. C'est pourquoi il faut choisir la largeur de cette chaîne avant de la construire. Cette grandeur est définie à l'aide de « L_m », représentant la largeur des chaînes du mécanisme d'extraction. Il est possible de



(a) Chemin de données original (b) Ajout d'une chaîne de sérialisation

FIGURE 6.2 – Insertion d'une chaîne de sérialisation dans un circuit simple.



(a) $L_m = 1$ (b) $L_m = 8$

FIGURE 6.3 – Chaînes de sérialisation : simple et parallèle.

construire une chaîne minimaliste ayant un bit de large, comme sur la Figure 6.3a où $L_m = 1$. L'avantage de ce type de chaîne réside dans sa simplicité et les contraintes de routages réduites qu'il engendre sur les outils de construction du circuit. À contrario, le temps d'extraction est extrêmement long car aucun parallélisme n'est utilisé. De plus, il faudra en général désérialiser le contexte extrait avant de pouvoir le transférer.

Il est aussi possible de construire plusieurs chaînes de sérialisation qui une fois mises en parallèle réaliseront la fonction d'extraction en moins de temps. Pour faciliter la construction d'une telle chaîne, on s'appuie sur les registres existants pour trouver les bits à connecter en parallèle, comme le montre la Figure 6.3b. En contrepartie, il est parfois nécessaire d'ajouter des registres dits *inutiles* dans la chaîne parallèle ainsi créée, afin de correctement synchroniser le déplacement des mots dans la chaîne. Un des avantages de ce type de chaîne est la rapidité d'extraction des registres. L'inconvénient majeur est donc lié au surcout des registres inutiles ainsi que les contraintes fortes que ces liens parallèles ajoutent aux outils chargés des étapes de construction du circuit. Ce type de chaîne a été présenté pour la première fois par [Gar+07] puis par [Koc+07]. Son impact n'a pas été analysé sur des circuits complexes. Les expérimentations du chapitre suivant montreront les surcouts liés à ces deux méthodes.

Chaînes et groupements

Dans le chapitre 5, on a sélectionné pour une tâche matérielle un ensemble de points de sauvegarde qui respectent le problème posé tout en réduisant les surcouts. Or, chaque point de sauvegarde possède un contexte qui lui est propre. Une variable peut faire partie du contexte du point de sauvegarde $C1$ mais pas $C2$ par exemple.

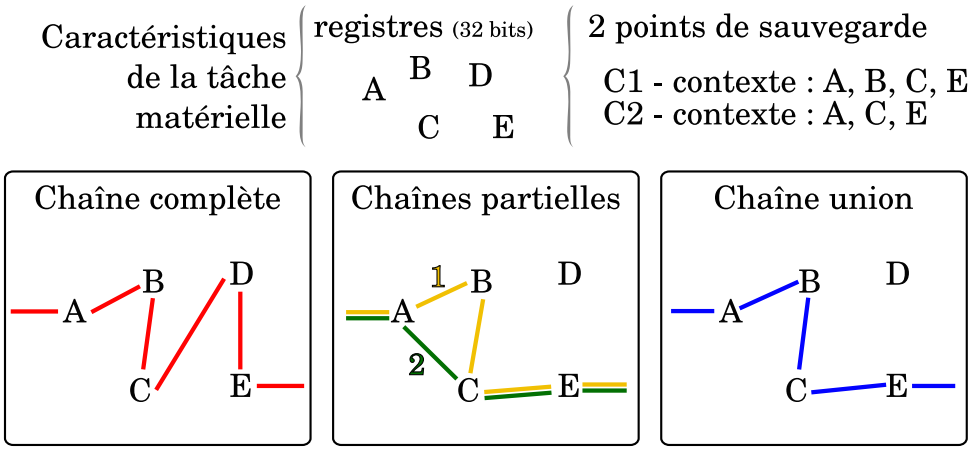


FIGURE 6.4 – Les différents regroupements de chaînes possibles.

Il faut donc décider comment construire la ou les chaînes d'extraction. La Figure 6.4 illustre trois groupements qu'il est possible de faire à partir d'une tâche matérielle possédant cinq registres, pas de mémoire et deux points de sauvegarde. Il n'est pas nécessaire de préciser le nombre d'états de la tâche ni la nature de son chemin de données. Les cinq registres, A, B, C, D, et E, sont des registres de 32 bits. Les deux points de sauvegarde ont pour contextes respectifs $R_{C1} = \{A, B, C, E\}$ et $R_{C2} = \{A, C, E\}$, et on note $|R_{Ci}|$ le cardinal de l'ensemble R_{Ci}. Dans cette figure, on considère que tous les liens entre les registres sont réalisés de manière parallèle (cf. 6.2.1) et que la chaîne possède une largeur 32 bits.

Chaîne complète Un premier mécanisme consiste à mettre dans une chaîne tous les registres présents dans le circuit. Sur la figure, on l'appelle « chaîne complète ». Elle est utilisée uniquement à but comparatif. Un tel mécanisme ne permet en effet pas à la tâche de respecter la contrainte de latence, tous les points de sauvegarde possédant virtuellement le même contexte constitué de tous les registres et ceci après la phase de sélection tenant justement compte des tailles de contexte différentes. Soit r le nombre de registres présents dans une tâche matérielle. Soit N la taille de chaque registre considéré. Le temps d'extraction d'une telle chaîne est de r cycles d'horloge et la quantité de données extraites en bit est de $r \times N$. Appliqué au cas concret de la Figure 6.4, cela donne un temps d'extraction de 5 cycles d'horloge et une quantité de données extraites de $5 \times 32 = 160$ bits. Le surcout matériel est quant à lui difficile à appréhender de manière analytique. En effet, il est impossible de prévoir quels seront les choix des outils de synthèse logique, placement et routage et donc le cout réel de chaque portion du mécanisme. Il est quand même possible de donner une estimation, si ce n'est de contraintes de routage additionnelles et leur impact, au moins du nombre d'entrées de multiplexeurs à ajouter au circuit afin de créer les nouvelles connexions. Dans le cas d'une chaîne complète, on ajoute r entrées de multiplexeurs N bits de large, donc 5 entrées de multiplexeurs 32 bits.

Chaînes partielles Un deuxième groupement, fonctionnel du point de vue des contraintes imposées, consiste à créer une chaîne de sérialisation propre à chaque point de sauvegarde. Le mécanisme est donc constitué d'autant de chaînes de sérialisation qu'il y a de points de sauvegarde avec des contextes relatifs aux registres différents. Bien entendu, deux points de sauvegarde avec des contextes similaires partagent la même chaîne. On appelle ce groupement un mécanisme de « chaînes partielles » et on utilise le sigle CSP pour « Chaîne de Sérialisation Partielle » pour désigner une chaîne parmi l'ensemble. Il est possible de partager certains chemins de données entre les chaînes afin de limiter les pénalités. Par exemple, les liens entre l'extérieur du circuit et le registre A, entre C et E et entre E et la sortie du mécanisme sont partagés. Le temps d'extraction moyen de ce mécanisme est le plus faible de tous, car n'est extrait que le contexte nécessaire au bon redémarrage de la tâche pour un point de sauvegarde précis. Soit c le nombre de points de sauvegarde d'une tâche matérielle. En admettant que tous les registres aient la même largeur que les chaînes partielles, le temps d'extraction moyen d'un contexte par point de sauvegarde t_m et la quantité de données moyenne extraite par point de sauvegarde q_m sont calculables avec les formules :

$$t_m = \frac{\sum_{i=1}^{c} |R_{Ci}|}{c}$$
$$q_m = t_m * N$$

Appliquées au cas présent, elles donnent $t_m = \frac{3+4}{2} = 3,5$ cycles d'horloge ainsi que $q_m = 3,5 \times 32 = 112$ bits extraits en moyenne par points de sauvegarde. Ces chaînes partielles sont les plus efficaces du point de vue de ces deux métriques. Toutefois, elles ont un impact non négligeable sur le surcout matériel du mécanisme complet. La formule suivante indique le nombre d'entrées de multiplexeurs n_{mux} à ajouter :

$$n_{mux} = \sum_{i=1}^{c} |R_{Ci}| - duplets\_communs(R_{C1}, ..., R_{Ci})$$

La fonction $duplets\_communs$ retourne le nombre de paires communes entre tous les éléments des chaînes partielles, c'est-à-dire le nombre de chemins partagés entre plusieurs chaînes partielles. Dans le cas présent, les chemins partagés sont au nombre de deux : celui qui va de l'entrée de la chaîne à A et celui qui va de C à E. Sur les sept entrées de multiplexeurs initiales, on en économise deux, ce qui donne $n_{mux} = 5$. La répartition est donc de une pour A, une pour B, deux pour C et une pour E.

Chaîne union Afin de tenter de faire une synthèse entre les deux regroupements précédents, nous proposons une « chaîne union ». Cette chaîne unique, contrairement au paradigme précédent, est constituée du plus petit ensemble possible parmi tous les registres de la tâche matérielle. Cet ensemble est donc constitué de tous les registres qu'il faudrait extraire, peu importe le point de sauvegarde atteint si la tâche

Chaîne(s)	Complète	Partielles	Union
Temps d'extraction (en cycles d'horloge)	5	$3 \leq t \leq 4$ ($t_m = 3,5$)	4
Taille des données (en bit)	160	$92 \leq q \leq 128$ ($q_m = 112$)	128
Entrées de mux 32 bits additionnelles	5	5	4

TABLE 6.2 – Caractéristiques des trois types de regroupements de chaînes pour la tâche matérielle considérée.

devait commuter. On appelle ce groupement une CSU, pour Chaîne de Sérialisation Union. L'ensemble des registres à extraire, R_U, est donc $R_U = \cup_{i=1}^{c} R_{Ci}$. Le temps d'extraction est dans ce cas égal aux nombres d'éléments de R_U, c'est-à-dire $|R_U|$, et la quantité de données extraites vaut $|R_U| \times N$. Le nombre d'entrées de multiplexeur à ajouter est $n_{mux} = |\cup_{i=1}^{c} R_{Ci}|$. Pour le cas présent, on obtient un temps d'extraction de 4 cycles d'horloge, une quantité de données de $4 \times 32 = 128$ bits et un supplément d'entrées de multiplexeur de 4.

Comparaison Les résultats des trois types de regroupements de chaînes qu'il est possible de faire sur le cas concret de la Figure 6.4 sont visibles dans le Tableau 6.2. La chaîne complète, non fonctionnelle, est la plus naïve et donc la plus couteuse selon deux critères : temps d'extraction le plus long et taille des données à extraire la plus importante. Les chaînes partielles offrent un avantage du point de vue de la taille du contexte relatif au registre extrait. Par contre, elles peuvent se révéler couteuses si les réutilisations de chemins sont peu nombreuses. Pour finir, la chaîne union est un intermédiaire qui se révèle peu couteux par rapport à ses concurrents. Les expérimentations sont nécessaires pour observer plus précisément les résultats des outils de synthèse logique, placement et routage sur les différentes chaînes.

Ordonnancement dans une chaîne

Un problème lié à la création de chaîne de sérialisation subsiste : l'ordonnancement des registres au sein de cette chaîne. L'ordre des registres dans la chaîne a une importance cruciale pour la qualité des circuits produits. Par exemple, connecter directement deux registres qui sont situés à deux extrémités d'un circuit est extrêmement couteux en terme de ressource de routage (dans les FPGA comme dans les circuits de type ASIC). Pour résoudre ce problème, il faut connecter les éléments par proximité géographique et ainsi limiter la distance de fil additionnelle et donc la saturation des ressources de routage dans le FPGA. La Figure 6.5 représente la position géographique de cinq registres une fois le circuit correspondant configuré dans un FPGA (avant la phase de placement, les informations de nature géographique sont inconnues). Si la connexion est réalisée de manière aléatoire, en connectant les registres de manière indéterminée, les ressources de routage sont potentiellement

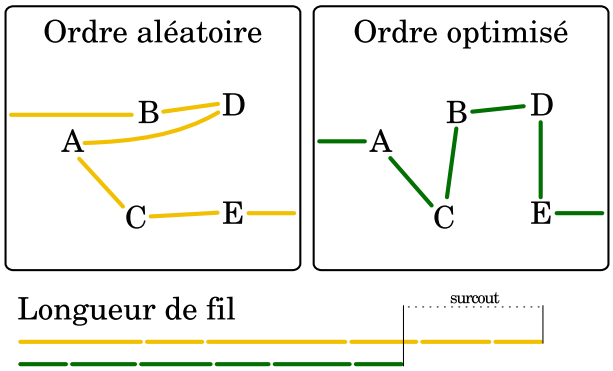


FIGURE 6.5 – Ordonnancement des registres dans une chaîne de sérialisation et surcout.

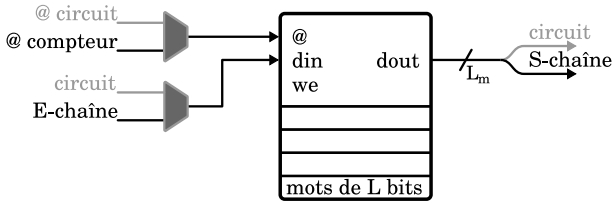


FIGURE 6.6 – Équipement d'une mémoire à sérialiser.

sur-utilisées. Dans le second cas, les connexions peuvent être faites selon plusieurs critères dont la proximité des registres entre eux par exemple. Or, la proximité géographique de registres dans la description haut niveau d'une application n'a pas de sens. Afin de répondre, partiellement, à ce problème, il est possible de s'appuyer sur la proximité algorithmique entre les registres. C'est ce qu'ont proposé ZAOURAR et al. [Zao+12], qui présentent une analyse à base de graphes pour retrouver les variables dont les connexions sont potentiellement les moins chères et ceci au niveau RTL. Le graphe construit met en évidence les relations entre registres (liens directs ou à travers des opérateurs) et permet la formalisation mathématique sous forme du problème dit « du voyageur de commerce » afin d'établir un plus court chemin parcourant tous les registres. Cette solution est envisageable à haut-niveau étant donné que la nature de la représentation intermédiaire du circuit permet une construction sous forme de graphe similaire. Pour l'heure, aucun algorithme permettant d'améliorer l'ordonnancement des registres dans les chaînes de sérialisation proposées n'est mis en place.

6.2.2 Mémoires

Équipement des mémoires

Une mémoire est un tableau dont les éléments sont accessibles via leur adresse (entrée « @ » sur la Figure 6.1b). Il n'est donc pas possible de créer une chaîne entre chacune de ses cases, comme pour l'ensemble des registres. Pour extraire le contenu d'une mémoire, il faut parcourir intégralement son espace d'adressage

et ainsi lire l'ensemble des cases de celle-ci. La solution utilisée pour équiper une mémoire est présentée sur la Figure 6.6. Elle consiste à ajouter deux multiplexeurs ou entrées de multiplexeurs existants sur les ports d'adresse et de donnée de la mémoire. Lorsqu'on souhaite extraire le contenu de cette mémoire, il faut alors utiliser l'adresse fournie par un compteur parcourant l'intégralité de l'espace d'adressage de la mémoire. Pour réintroduire le contenu précédemment extrait, il faut utiliser de la même manière le compteur d'adresses afin de parcourir l'espace d'adressage mais en plus proposer les données préalablement enregistrées sur le port d'entrée « din » à l'aide du second multiplexeur. Il faut noter qu'il est nécessaire de faire la différence entre les mémoires à lecture asynchrone (comme les LUTRAM de Xilinx) et les mémoires à lecture synchrone (comme les BRAM de Xilinx) car ces dernières possèdent un registre tampon intégré dans le bloc mémoire. Le contenu de ce registre tampon devra être extrait en tant que partie du contexte. Il devra de plus être réécrit en passant par une case mémoire car c'est le seul moyen d'accéder à ce registre. Équiper un bloc mémoire synchrone est donc plus couteux et plus complexe. Ce type d'équipement des mémoires a été proposé par WHEELER et al. [Whe+01] dans le contexte du débogage de circuit intégré sur FPGA. Les chaînes de sérialisation qui sont proposées font nécessairement un bit de large. Un mécanisme de sérialisation/désérialisation est nécessaire afin de produire des mots de la bonne largeur selon l'interface ciblée (produire des mots de 1 bit pour l'extraction, soit une sérialisation, ou produire des mots de la largeur de la mémoire pour l'écriture, soit une désérialisation). Dans notre cas, on ne considère pas comme souhaitable le découpage intra-mot mémoire à l'aide de sérialiseur/désérialiseur. En conséquence, pour une première approche, on considère que la largeur de la chaîne de sérialisation à construire est au moins de la largeur de la plus large des mémoires faisant partie du contexte de la tâche matérielle.

Organisation dans une chaîne

Dans la conception classique d'une chaîne de sérialisation, tous les éléments sont reliés et l'information à extraire circule de proche en proche. Cette méthode n'est pas appliquée telle quelle dans le mécanisme proposé pour les mémoires. Afin de simplifier le mécanisme d'extraction, celles-ci ne sont pas mises dans la même (ou les mêmes dans le cas de chaînes de sérialisation partielles) chaîne que les registres. La donnée venant d'un registre situé avant une mémoire dans une chaîne classique devrait être écrite dans une case de la mémoire avant d'être lue de nouveau lors d'un cycle supplémentaire du parcours de l'espace d'adressage. Le parcours de l'espace d'adressage lors de la réécriture dépendrait donc de la position de la mémoire dans la chaîne pour correctement prendre en compte la traversée des mémoires par les données en amont de celles-ci. De plus, séparer les deux chaînes facilite la mise en place de la non extraction de certaines mémoires (cf. 6.4). Dans le cas où les mémoires seraient intégrées dans la même chaîne que les registres, il faudrait prévoir un mécanisme de dérivation (*bypass*) de la mémoire dont l'extraction n'est pas requise. Pour finir, cette méthode présente l'avantage de ne faire aucune hypothèse sur la technologie de mémoire utilisée et notamment sur

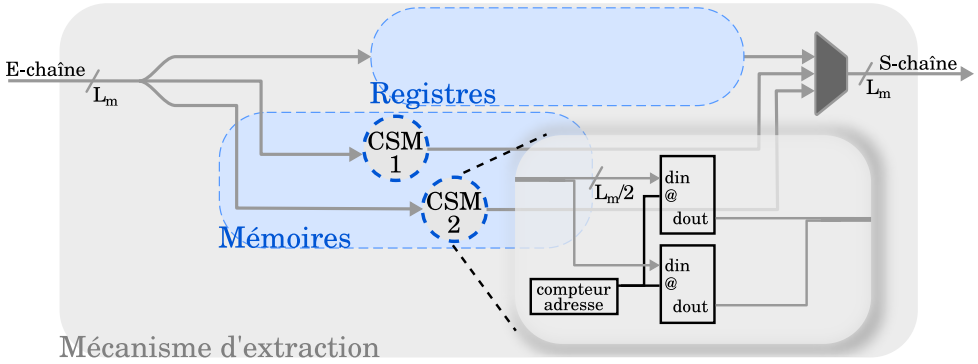


FIGURE 6.7 – Mécanisme d'extraction complet dans le chemin de données d'une tâche matérielle.

son comportement lorsqu'elle est lue et écrite à la même adresse pendant le même cycle. En effet, lectures et écritures sont effectuées dans les cas distincts que sont respectivement extraction et restauration de contexte.

Les mémoires sont donc extraites une par une, après l'extraction des registres effectuée à l'aide d'un mécanisme de chaînes partielles ou union. La Figure 6.7 présente la forme générale du mécanisme d'extraction sur le chemin de donnée d'une tâche matérielle. Il ne nous importe pas de préciser quel mécanisme est utilisé pour extraire les registres ici. Pour ce qui concerne les mémoires, une CSM, pour « Chaîne de Sérialisation Mémoire », est constituée initialement d'une seule mémoire. Tous les flux en provenance des différentes chaînes (registres et mémoires) sont ensuite multiplexés à la sortie du mécanisme d'extraction « S-chaîne ». L'entrée « E-chaîne » est distribuée à toutes les chaînes de sérialisation. Afin de limiter le temps d'extraction de toutes les mémoires, il est possible d'en placer plusieurs en parallèle dans une CSM. On parle alors d'empilement des mémoires dans une CSM. En effet, si la largeur de la chaîne d'extraction est de L_m bits de large et que deux mémoires ont une largeur de mot égale à $L_m/2$, il est possible de les extraire en même temps. C'est ce qu'illustre le bloc en bas à droite de la Figure 6.7 qui est une maximisation de la CSM 2.

6.2.3 Organisation de la machine d'état

Les changements apportés sur le chemin de données d'une tâche permettent la fonctionnalité matérielle d'extraction de contexte. Il faut encore modifier la machine d'état (FSM) de ce circuit afin de prendre en compte les demandes de préemption du système et piloter les migrations de contexte. La Figure 6.8 différencie la partie initiale de la FSM (à gauche) des états ajoutés (à droite) afin de mettre en œuvre les migrations. Le cas présenté utilise un mécanisme d'extraction partielle et on remarque donc la présence d'états activant les CSP. Dans la FSM initiale, on ajoute un branchement conditionnel à la sortie de chaque point de sauvegarde. Sur la figure est visible uniquement le branchement du point de sauvegarde 2 (« CP2 ») vers un état d'identification (« ID »). Ce branchement est effectué lorsqu'une extraction de contexte est requise. L'état d'identification permet d'analyser un registre du chemin

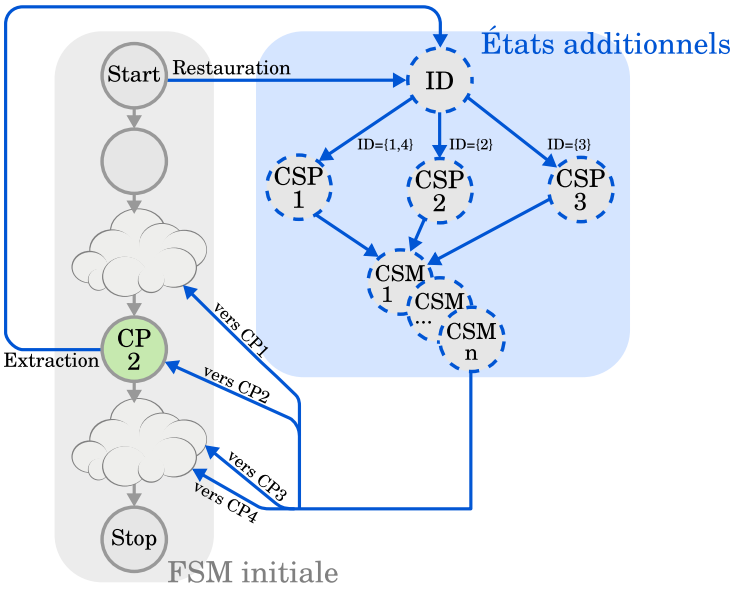


FIGURE 6.8 – Modifications apportées à la machine d'état d'une tâche matérielle. Cas avec chaînes d'extraction des registres partielles.

de données qui n'a pas encore été présenté : le registre d'identification du point de sauvegarde courant. Ce registre est rempli avec l'identifiant du point de sauvegarde rencontré afin de parcourir la CSP correspondant à son contexte. Dans le cas où le CP 2 serait rencontré lors d'une demande de préemption, c'est la CSP 2 qui est mise en œuvre afin de sauvegarder le contexte du point de sauvegarde 2. Par ailleurs, on remarque que la CSP 1 est commune aux CP 1 et 4. Après l'extraction du contexte relatif aux registres du point de sauvegarde rencontré, c'est au tour des mémoires de s'extraire à partir d'états pilotant les différentes CSM. Lorsqu'on extrait le contexte hors de la tâche matérielle, c'est ici qu'elle s'arrête. Si, au contraire, on réintroduit un contexte précédemment sauvegardé, il reste à rejoindre le point de sauvegarde correspondant à ce contexte et poursuivre l'exécution de la tâche.

6.3 Fragmentation induite

Un problème lié à l'insertion de chaînes parallèles, entrevu en 6.2.1 concerne l'introduction des registres « inutiles ». L'empilement des mémoires dans les CSM répond à un problème identique. C'est une fragmentation des données extraites, et ce à cause du mécanisme d'extraction, qu'on doit résorber. Ce paragraphe présente le problème rencontré et la réponse apportée pour les deux types d'éléments mémorisants, chacun nécessitant une approche adaptée.

6.3.1 Problème rencontré

On parle de fragmentation lorsqu'on est face à un processus qui découpe un ensemble en de multiples sous-parties. La fragmentation devient un problème

lorsqu'elle implique une dégradation des performances ou a un impact sur le cout d'un système. Ici, l'ensemble à découper en sous-parties, qu'on appellera mots, est le contexte relatif à un point de sauvegarde. La largeur des mots du contexte est bien évidemment fixe, définie lors de la construction du mécanisme d'extraction. On rappelle que « L_m » représente la largeur des chaînes du mécanisme d'extraction. Or les données à extraire, qu'elles soient enregistrées dans un registre ou une mémoire, ne sont pas malléables à merci. Fixer la valeur de L_m à une valeur trop basse défavorise la vitesse d'extraction mais évite le phénomène de fragmentation, parfois au détriment des ressources matérielles à cause d'un découpage trop important des données à extraire. Au contraire, une largeur de chaîne élevée permet une extraction très rapide des données mais la fragmentation est alors forte. Précisons la nature du problème en fonction du type d'élément mémorisant.

6.3.2 Registres fragmentés

La fragmentation des données relative au contexte lié aux registres est due à l'utilisation de chaînes parallèles. Lorsque $L_m = 1$, c'est-à-dire dans le cas de la chaîne simple vue en 6.2.1, la fragmentation est inexistante. À partir de $L_m \geq 2$, le phénomène de fragmentation peut apparaitre. Le cas est illustré par la Figure 6.3, où dès qu'une chaîne parallèle est mise en place, des registres inutiles peuvent être nécessaires. Ils évitent les décalages inter-mots entre extraction et insertion de contexte. Ajouter des registres inutiles implique non seulement un surcout matériel (des FF ainsi que des ressources de routage) mais aussi une extraction de données plus volumineuse que le seul contexte d'un point de sauvegarde.

La grandeur L_m influence donc la manière de placer les registres dans une chaîne. L'algorithme imaginé pour l'étape de construction de chaîne essaye actuellement de mettre le plus de registres possibles dans un mot afin de réduire l'ajout de registres inutiles. On rappelle qu'aucun ordonnancement des registres au sein de la chaîne n'est préalablement réalisé, et qu'il est donc possible de mettre les registres dans un ordre quelconque. L'avantage de ne pas mêler ordonnancement et construction, au delà de la plus grande simplicité de mise en œuvre de l'algorithme, est qu'on peut réduire l'ajout de registres inutiles en déplaçant des registres afin de compléter des mots. Une règle prévaut toutefois : aucune découpe de registre en vue de compléter un mot n'est possible. Cela peut potentiellement conduire à éclater des registres tout au long d'une chaîne et les conséquences pour les ressources de routage et de multiplexage seraient probablement catastrophiques. La seule découpe autorisée concerne un registre dont la taille est un multiple de L_m. Dans ce cas, seule la connexion des bits du registre sur lui-même est possible. Par exemple la Figure 6.9 illustre cette découpe lorsque $L_m = 8$ pour des registres de 16 bits. Les 8 premiers bits des registres qui en contiennent 16 sont connectés aux 8 derniers. Ceux-ci sont ensuite connectés aux 8 premiers bits du registre suivant. Ce type de découpe qui ne permet que les connexions de proche en proche limite les surcouts. Lorsque la taille des registres ne permet par leur découpe, comme dans le cas $L_m = 18$ bits avec des registres de 16 bits, et qu'aucun registre ne rentre dans l'intervalle laissé vacant, un registre inutile (« RI » sur la figure) est créé. Dans le cas $L_m = 32$ bits, le

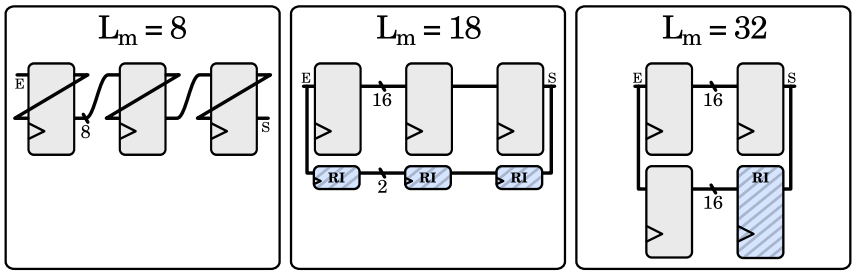


FIGURE 6.9 – Fragmentation d'une chaîne des trois registres de 16 bits pour $L_m = \{8, 18, 32\}$.

premier mot est constitué de deux registres mais pas le dernier mot car il manque un registre. C'est donc un registre inutile qui complète le mot.

L'Algorithme 3 décrit la mise en place d'une chaîne de registres et donc la règle de création de registres inutiles. Cet algorithme est naïf et ne constitue pas une contribution scientifique. Il permet néanmoins d'illustrer le phénomène de fragmentation. D'autre part, il est possible de formaliser ce problème d'ordonnancement en un problème du sac à dos (moyennant des contraintes supplémentaires), problème mathématique NP-complet. Trouver une solution exacte n'est donc pas envisageable et cet algorithme, bien que largement sous-optimal, est suffisant. Ajoutons qu'à terme, l'algorithme d'ordonnancement doit avoir des objectifs divergents avec ceux du problème du sac à dos. En effet, on souhaite appliquer une méthode comme vu dans le dernier paragraphe de 6.2.1 afin de sélectionner les registres algorithmiquement proches, et non minimiser la quantité de registres inutiles.

Cet algorithme prend en entrée une liste des registres à connecter (dont la taille est accessible avec l'attribut « taille »). Deux listes de travail sont utilisées : Liste_E et Liste_temp, respectivement listes des registres contenus dans le mot d'entrée et de sortie. Le premier mot d'entrée n'est pas constitué de registres. C'est la connexion à E-chaîne, l'entrée de la chaîne de sérialisation. On cherche ensuite à trouver le mot de sortie dans la boucle de la ligne 3. Tant que la liste des registres à connecter n'est pas vide, on cherche un mot à connecter (remplissage de Liste_temp, ligne 4 à 28), on le connecte au mot précédent (connexion avec Liste_E ligne 29) puis le met en entrée (ligne 30). Le processus de sélection du mot de sortie mérite plus de précisions. La première étape consiste à parcourir tous les registres (ligne 5) afin d'en sélectionner tant que mot_courant $\neq L_m$ (cf. *break* ligne 22). Si le registre a une taille dépassant L_m et qu'aucun registre n'est déjà sélectionné, on vérifie qu'on peut le découper et on procède à la connexion de ce registre sur lui-même avec la procédure CONNEXION_ET_DÉCOUPE. Si on ne peut pas le découper, une erreur est retournée enjoignant l'utilisateur à augmenter L_m. Sinon, si le registre peut entrer dans le mot courant, c'est-à-dire que mot_courant + reg.taille $\leq L_m$, alors on l'ajoute dans Liste_temp. Lorsque tous les registres ont été testés, si le mot est incomplet, il faut créer un registre inutile et l'ajouter à Liste_temp.

Le Tableau 6.3 donne les résultats de l'algorithme de construction de chaînes qui précède sur la quantité de bits de registres inutiles générés. Une différence très nette apparait entre les chaînes partielles, qui génèrent beaucoup de registres

Algorithme 3 Algorithme d'ordonnancement naïf des registres

Require: liste_registre[nombre de registres] = {nom, taille}
1: Liste_E = $\emptyset$; Liste_temp = $\emptyset$
2: Liste_E = E-chaîne ▷ l'entrée de la chaîne est le premier élément à connecter
3: **while** liste_registre n'est pas vide **do** ▷ connexion de tous les registres
4: mot_courant = 0
5: **for all** reg dans liste_registre **do** ▷ parcours de tous les registres
6: **if** reg.taille $> L_m$ et mot_courant $= 0$ **then** ▷ reg est plus grand que L_m
7: **if** reg.taille modulo $L_m = 0$ **then**
8: CONNEXION_ET_DÉCOUPE(Liste_E, reg)
9: mot_courant = L_m
10: liste_registre = liste_registre \{reg} ▷ on retire reg de liste_registre
11: Liste_temp = Liste_temp $\cup$ reg ▷ ajout de reg dans Liste_temp
12: **break**
13: **else**
14: retour erreur ▷ découpe non possible
15: **end if**
16: **else if** mot_courant + reg.taille $\leq L_m$ **then**
17: Liste_temp = Liste_temp $\cup$ reg ▷ ajout de reg dans Liste_temp
18: liste_registre = liste_registre \{reg}
19: mot_courant += reg.taille ▷ mise à jour de la taille du mot courant
20: **end if**
21: **if** mot_courant $= L_m$ **then**
22: **break**
23: **end if**
24: **end for**
25: **if** mot_courant $< L_m$ **then** ▷ création de registre inutile
26: reg_inutile = NOUVEAU_REGISTRE
27: Liste_temp = Liste_temp $\cup$ reg_inutile
28: **end if**
29: CONNEXION(Liste_E, Liste_temp) ▷ connexion des registres entre eux
30: Liste_E = Liste_temp ; Liste_temp = $\emptyset$
31: **end while**
32: Liste_temp = S-chaîne ▷ connexion à la sortie
33: CONNEXION(Liste_E, Liste_temp)

inutiles, et la chaîne union, qui en génère moins. Les cases *n.d*, pour « non défini », illustrent l'incapacité de l'algorithme à découper un ou plusieurs registres faisant partie d'une chaîne. Cela correspond à la ligne 14 de l'Algorithme 3. Le surcout en FF est directement calculable à partir de cette donnée brute de la quantité de registres inutiles ajoutés. Par contre, il n'est pas possible de trouver un modèle simple capable de prédire l'impact sur les autres ressources (LUT pour les multiplexeurs et routage) de cet ajout. Pour avoir une connaissance précise du surcout des registres inutiles, il faut donc continuer le flot de conception après la synthèse de haut niveau afin

	Chaînes partielles			Chaîne union		
L_m	16	32	64	16	32	64
adpcm	n.d	671	1055	n.d	66	130
aes	19	51	275	19	51	83
blowfish	21	53	245	21	53	85
dfadd	28	60	92	28	60	92
dfdiv	43	59	155	28	44	108
dfmul	14	30	62	14	30	62
dfsin	57	89	281	26	42	106
gsm	11	171	491	11	27	59
idct	23	55	119	23	55	87
jpeg	534	1110	2390	22	54	118
mips	13	29	125	13	29	61
mjpeg	304	816	1584	22	54	118
mpeg2	440	984	1560	16	48	80
sha	n.d	29	253	n.d	29	93

TABLE 6.3 – Quantité de registres inutiles en bit pour différents L_m.

d'atteindre un niveau d'information fiable.

6.3.3 Mémoires fragmentées

La fragmentation des mémoires provient, comme pour les registres, de l'utilisation de chaînes parallèles. Or, celles-ci sont nécessaires car on ne souhaite pas utiliser de mécanisme de sérialisation/désérialisation autour des mémoires. La Figure 6.10 met en évidence le problème de fragmentation des données extraites. Ce cas particulier montre le profil d'extraction de deux mémoires sur une chaîne de sérialisation parallèle caractérisée par $L_m = 32$ bits. Ces deux mémoires possèdent respectivement une largeur de mot de 32 bits (la mémoire A) et de 16 bits (la mémoire B) et ont chacune 64 mots. Étant donné que l'extraction d'une mémoire s'effectue exclusivement sur l'intégralité de son contenu, le contexte relatif aux mémoires est ici de $32 \times 64 + 16 \times 64 = 3072$ bits. Or la largeur du mécanisme d'extraction est $L_m = 32$ bits afin d'extraire la plus large des mémoires. C'est donc plus qu'il n'est nécessaire à la mémoire B. On observe, après l'extraction de la mémoire A qui occupe l'intégralité de la largeur de la chaîne de sérialisation, que l'extraction de la mémoire B n'en occupe que la moitié. Pourtant, les mots reçus par le système extérieur sont de 32 bits. Une quantité de 1024 bits ne faisant partie d'aucun contexte utile est donc stockée.

De la même manière que pour les registres, on recueille les chiffres concernant la fragmentation du contexte lié aux mémoires sur les applications du banc de test. On évite de présenter les applications dfadd, dfdiv, dfmul et dfsin étant donné qu'aucune mémoire n'y est présente. Le Tableau 6.4 rassemble les données qui nous intéressent afin de mesurer l'impact de cette fragmentation. La première colonne du tableau recense la taille totale des éléments mémorisants placés en mémoire (donnée précédemment vue dans le Tableau 6.1). La seconde partie, divisée en deux

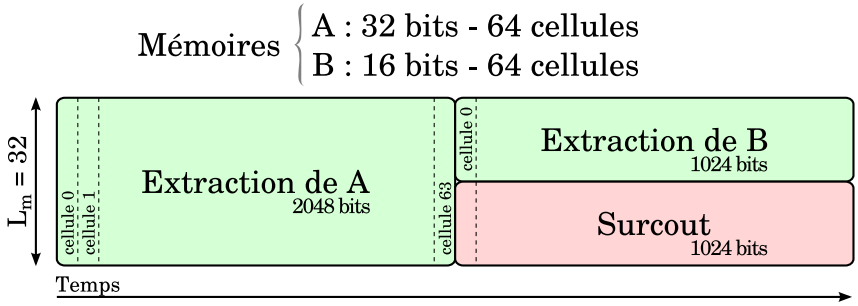


FIGURE 6.10 – Profil d'extraction de deux mémoires de largeur différente pour $L_m = 32$ bits et surcout associé.

		Extraction (cycles)		Fragmentation		% de frag.	
	taille (bit)	32	64	32	64	32	64
adpcm	12608	394	235	0	2432	0%	16%
aes	20608	644	576	0	16256	0%	44%
blowfish	34112	1082	1042	512	32576	1%	49%
gsm	3408	186	169	2544	7408	43%	68%
idct	2560	128	64	1536	1536	38%	38%
jpeg	371177	28302	26055	534487	1296343	59%	78%
mips	3072	96	64	0	1024	0%	25%
mjpeg	291600	14443	13008	170576	540912	37%	65%
mpeg2	16768	2060	2052	49152	114560	75%	87%
sha	3232	101	85	0	2208	0%	41%
Moyenne						25%	51%

TABLE 6.4 – Fragmentation du contexte relatif aux mémoires pour $L_m = \{32, 64\}$.

colonnes pour $L_m = 32$ bits puis $L_m = 64$ bits, présente le temps d'extraction du contexte relatif aux mémoires selon le schéma précédemment proposé. On rappelle que plusieurs mémoires peuvent être concaténées dans un même mot (cf. 6.2.2). La fragmentation est alors calculée à l'aide de la formule $L_m \times temps\_extraction - taille$ dans les deux colonnes suivantes. On obtient une grandeur en bit correspondant au « trop plein » d'informations extraites. Les deux dernières colonnes illustrent cette part de données inutiles extraites dans le contexte relatif aux mémoires, ou le pourcentage de fragmentation, c'est-à-dire la valeur de $\frac{fragmentation}{L_m \times temps\_extraction}$. Pour un mécanisme dont $L_m = 32$ bits, la part des données extraites inutiles n'est pas négligeable : en moyenne un quart des données extraites sont liées au problème de fragmentation. Pour $L_m = 64$ bits, le surcout moyen est inacceptable. On double les données extraites dans ce dernier cas.

Il est possible de se débarrasser de la plupart de la fragmentation en adoptant une technique de découpe des mémoires. Le concept consiste à pouvoir accéder à deux mots d'une mémoire à la fois afin de compléter la largeur de la chaîne de sérialisation disponible (cf. Figure 6.11). La part la plus importante de la fragmentation dans notre banc de test est en général originaire d'une seule mémoire de taille très importante comparée aux autres. Par exemple, dans le cas du mpeg2, on observe la présence d'une mémoire de 2048 mots de 8 bits, ce qui représente plus de 97% de

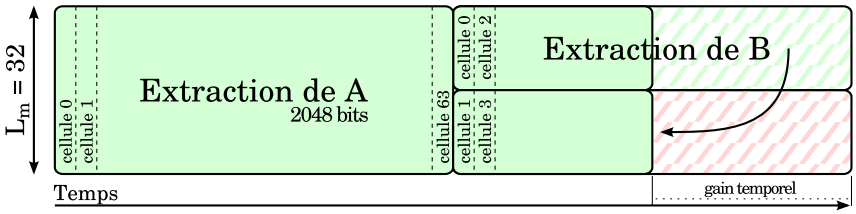
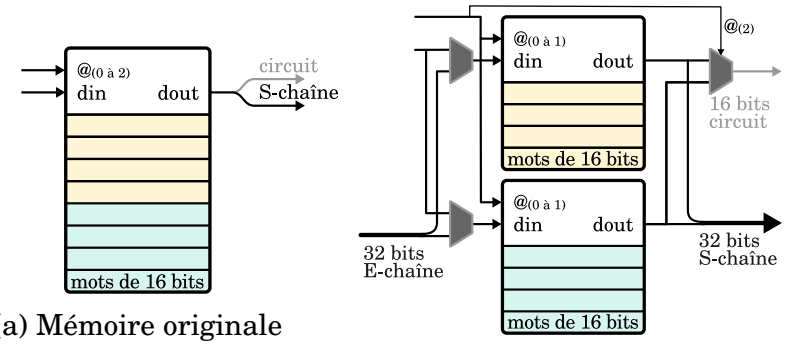


FIGURE 6.11 – Profil d'extraction de deux mémoires dont une découpée pour $L_m = 32$ bits et gain temporel associé.



(a) Mémoire originale

(b) Mémoire découpée

FIGURE 6.12 – Découpe d'une mémoire de 8 cases de 16 bits pour $L_m = 32$ bits.

la taille totale du contexte relatif aux mémoires. Aucune autre mémoire ne peut s'additionner à celle-ci afin de réduire la fragmentation et la méthode d'empilement des mémoires dans les CSM n'est donc pas suffisante. Permettre l'accès en parallèle à plusieurs mots de cette mémoire permet de réduire radicalement la fragmentation. On peut s'en convaincre avec le Tableau A.1 visible en Annexe A qui présente les résultats d'une stratégie de découpage sur les applications de notre banc de test.

Le principe matériel de la technique de découpe des mémoires est illustré par la Figure 6.12. Dans le cas d'une mémoire de 16 bits de large et de 8 mots (cf. Figure 6.12a), une chaîne de sérialisation de 32 bits entraine une fragmentation du contexte extrait si aucun empilement n'est possible avec une autre mémoire de la tâche matérielle. Or, il est possible de lire deux mots de 16 bits en parallèle afin de compléter exactement la largeur de la chaîne de sérialisation. On découpe donc la mémoire en deux sous-ensembles dont la largeur de mot est inchangée. Il suffit d'utiliser le bit d'adressage supérieur ou inférieur afin de choisir quel mot envoyer au circuit dans le cycle de fonctionnement normal. Lors d'une extraction, les deux mots lus, à la même adresse mais dans deux mémoires différentes, sont concaténés pour former un mot de 32 bits envoyé sur S-chaîne. De même lors de la restauration de contexte, un multiplexeur est chargé de sélectionner soit une entrée venant du circuit, soit une portion de E-chaîne à réécrire dans une mémoire.

6.4 Ouverture : mémoires et analyse de la durée de vie

L'un des derniers problèmes abordé dans ce manuscrit, bien que de manière incomplète, concerne l'analyse de la durée de vie des variables en mémoire. On a justifié la seule analyse de la durée de vie des variables situées dans des registres par le plus grand impact de l'équipement de celles-ci sur le surcout du mécanisme. Il n'est toutefois pas sans intérêt de proposer une solution au moins partielle afin de réduire la taille du contexte relatif aux mémoires. Ce paragraphe présente le problème d'analyse de la durée de vie des variables en mémoire ainsi qu'une première solution simple qui a été mise en œuvre. Enfin, on propose des pistes permettant de raffiner la méthode.

6.4.1 Problème rencontré

L'accès à une variable située dans un registre du FPGA est direct. Il est donc peu complexe de définir la plage d'utilisation d'un registre à partir de la représentation intermédiaire d'un outil de HLS. Au contraire, une mémoire nécessite l'information supplémentaire d'un index afin de retourner la valeur qu'elle contient dans cette case. Cet index est d'une difficulté d'analyse variable, qui va du simple itérateur de boucle à une valeur dépendante des vecteurs d'entrée de la tâche. Dans ce dernier cas, il est impossible de prévoir la valeur de l'index et donc de définir lors de la synthèse quelles sont les cases « vivantes » de la mémoire. Le Verbatim 6.1 est un extrait de code en C pouvant être donné en entrée d'un flot de conception de haut niveau. L'analyse de la durée de vie des variables `a` et `b` est immédiate : `a` est vivante à partir de la ligne 5 et le reste tout au long de la tâche, tandis que `b` est vivante à partir de la ligne 10 et jusqu'à une ligne suivante. Cet effort n'est pas aussi simple à produire pour `mem`. On voit qu'on accède à deux valeurs distinctes de `mem` dans la boucle infinie (aux index `a` et `a-1`). À chaque itération, ces deux valeurs évoluent et on utilise les résultats calculés à l'itération n lors de l'itération $n+1$. L'état des variables contenues dans `mem` dépend donc directement de la valeur de `a`, qui dépend elle-même de l'itération dans laquelle se trouve la tâche. Il n'est donc pas possible de définir lors de l'étape de synthèse une plage fixe de durée de vie pour les variables contenues dans `mem`.

De plus, on n'analyse pas la durée de vie des mémoires dans l'algorithme de recherche des points de sauvegarde car le mécanisme d'extraction d'une mémoire est simple et relativement peu couteux par rapport aux registres. Complexifier l'extraction des données en mémoire en spécifiant des valeurs ou une plage d'adresse à extraire peut rapidement devenir excessivement couteux. Il faut donc trouver un équilibre entre l'analyse fine de la durée de vie des variables, dans les rares cas où cela est possible, et la complexité du mécanisme à mettre en place afin d'extraire seulement la partie vivante d'une mémoire.

```
1  int main(void)
2  {
3      int mem[10];
4      mem[0] = 1586;
5      int a = 0;
6      int b;
7      while(true){
8          a++;
9          mem[a] = fifo_read();
10         b = mem[a - 1];
11         fifo_write(b);
12         if(a == 9) a = 0;
13     }
14 }
```

Verbatim 6.1 – Complexité de l'analyse de la durée de vie d'une mémoire.

6.4.2 Solution intermédiaire

Une tâche matérielle créée à des fins d'accélération présente généralement un profil classique visible sur le Verbatim 6.1. Dans la fonction `main`, après l'instanciation de différentes variables, une boucle infinie (ligne 7) contient le traitement à effectuer. Celui-ci commence en général la lecture de données d'entrées, à l'aide ici de la fonction `fifo_read`. Enfin, après le traitement, la tâche communique les données obtenues au système. C'est le rôle de `fifo_write`, chargé d'écrire sur le bus de sortie la variable `b` obtenue après calcul. Une hypothèse permet de faire une simplification de l'analyse de la durée de vie d'une mémoire et ainsi ouvre la porte à une première solution. Celle-ci est la suivante : si une mémoire n'est pas explicitement marquée comme nécessaire entre les itérations de la boucle principale d'exécution, alors c'est qu'on peut considérer la mémoire comme non vivante entre deux itérations. Il faut donc trouver, dans la boucle principale d'exécution, le premier endroit où la mémoire est écrite. C'est à partir de cet état que la mémoire est considérée comme vivante. Une fois que la mémoire n'est plus lue, on peut la considérer comme hors du contexte. Cette hypothèse n'est pas immédiate à vérifier à l'aide de la représentation interne d'un outil de HLS. Comme dit précédemment, analyser tous les index d'accès à la mémoire afin de vérifier un potentiel parcours de lecture/écriture dénotant une utilisation restreinte à la boucle principale est complexe. Il n'est donc pas prévu d'automatiser la vérification de cette hypothèse. D'un autre côté, le développeur d'application est nécessairement conscient de l'utilisation qui est faite des mémoires qu'il instancie : est-ce une mémoire tampon qui ne sert à produire qu'un seul jeu de vecteurs de sortie ou est-ce une mémoire dont les itérations suivantes dépendent (typiquement dans le cas d'un décodeur) ? Dans cette optique, c'est le développeur d'application qui fournit l'information via un commentaire dans le code source. Le Verbatim 6.2 est doté de ce commentaire à la ligne 1. Si ce commentaire n'est pas présent, aucune hypothèse n'est faite et l'analyse de la durée de vie simpliste n'est pas effectuée. Le cas présenté possède le commentaire disant qu'aucune mé-

```
1  //PRAGMA mémoires vivantes = aucune
2  int main(void)
3  {
4      int mem[10];
5      int a = 0;
6      int b = 0;
7      while(true){
8          a++;
9          for(int i = 0; i < 10; i++){
10             mem[i] = fifo_read();
11         }
12         b = mem[a - 1];
13         fifo_write(b);
14         if(a == 10) a = 0;
15     }
16 }
```

Verbatim 6.2 – Ajout d'information au code source pour valider l'hypothèse de non réutilisation.

moire n'est réutilisée entres les itérations principales. C'est donc qu'on suppose l'hypothèse valide pour la mémoire `mem`. On remarque d'ailleurs que l'application démarre, après l'incrément de `a`, par une écriture complète de la mémoire. Dans ce cas simple et visuel, l'hypothèse précédente est effectivement validée. La durée de vie de la mémoire est ici limitée aux lignes 9 à 12. Avant, la mémoire n'est pas écrite. Après, la mémoire n'est plus lue. Comme un point de sauvegarde est placé dans la boucle d'attente potentiellement infinie liée à l'échange avec le système (`fifo_write`), celui-ci pourra éviter de passer par l'état CSM relatif à l'extraction de `mem` et ainsi économiser l'extraction de 10 mots de 32 bits.

6.4.3 Autres solutions envisagées

Deux solutions additionnelles sont envisagées afin d'améliorer l'analyse de la durée de vie des variables en mémoire. La première étend l'analyse qu'il est possible de faire lors de la synthèse et la seconde ajoute un mécanisme d'espionnage au circuit.

Analyse statique poussée

La première amélioration qu'il est possible d'apporter concernant l'analyse de la durée de vie des variables en mémoire est d'automatiser la validation de l'hypothèse précédemment présentée. En effet, une méthode ne faisant pas intervenir le développeur d'application est préférable. Pour ceci, il faut mettre en place une analyse des accès effectués sur une mémoire. Connaissant les schémas d'accès à la mémoire, on peut déterminer si celle-ci possède des plages d'utilisation différentes entre les itérations principales. Déterminer à chaque accès à la mémoire quel est l'index accédé est complexe et non envisageable actuellement.

Une approche similaire mais d'une portée plus faible consiste à faire l'analyse des accès dans l'optique de réduire la durée de vie sur un ensemble restreint d'états. Si l'analyse exhaustive des index d'accès à la mémoire est difficilement envisageable, on peut par contre réaliser des analyses plus simples comme celle de la boucle située ligne 9 du Verbatim 6.2. Sachant le parcours complet de la mémoire, on est certain que celle-ci n'est pas vivante avant cette boucle et ce jusqu'à la dernière lecture qui en est faite. On parvient donc à trouver une zone dans laquelle la mémoire ne fait pas partie du contexte.

Analyse dynamique

Étant donné la difficulté d'analyse de la durée de vie des mémoires lors de la synthèse de haut niveau, ou analyse statique, on envisage une solution basée sur l'analyse des accès aux index embarquée dans le circuit. C'est donc une analyse dynamique, « à chaud », faite pendant le fonctionnement du circuit qu'on imagine. L'ajout de logique dédiée à l'analyse des accès serait placé entre la mémoire et ses signaux de pilotage. Équiper toutes les mémoires de modules d'analyse n'est pas raisonnable compte tenu des surcouts potentiels. Par contre, dans le cas de mémoires dont la taille est grande par rapport au contexte de la tâche, un équipement est envisageable.

L'idée consiste à ajouter deux registres espions en relation étroite avec la mémoire. Ces deux registres (« min » et « max ») sont chargés d'analyser la plage mémoire accédée en écriture lors du déroulement de la tâche matérielle. Il est possible d'adopter deux stratégies. La première consiste à extraire la valeur de ces deux registres puis le contenu de la mémoire dans la plage min-max. À la restauration, il faut réintégrer les valeurs de min et max avant de restaurer le contenu de la mémoire. Un défaut de cette approche est que les deux registres ne sont jamais réinitialisés et on aboutit probablement rapidement à une plage min-max couvrant la totalité des index mémoire. Une seconde stratégie consiste à réinitialiser min et max à la restauration. Il faut au préalable extraire l'intégralité de la mémoire au premier changement de contexte. Pour les changements de contexte qui suivront, il faudra faire sur le contenu de la mémoire, un différentiel entre l'image de la mémoire stockée précédemment et l'image extraite qui correspond à la plage min-max ayant évolué. C'est cette image mise à jour à chaque changement de contexte qu'il faudra réintégrer. Dans ce cas, lors d'une restauration, l'intégralité de la mémoire doit transiter par le mécanisme de sérialisation. Du côté positif, les registres min et max sont réinitialisés et l'extraction suivante est réduite. Le gain est donc observé à l'extraction mais pas à la restauration.

L'implémentation et l'analyse de ces techniques n'ont pas pu être abordées dans ce travail de thèse.

Chapitre 7

Mise en œuvre et résultats

Ce chapitre présente l'outil développé ainsi que les expériences menées afin de valider la méthode développée dans cette thèse. On étudie ensuite l'impact des différentes variations de mise en œuvre de la méthode.

Sommaire

7.1 CP3 : *CheckPoint PinPoint*

La méthode de conception de circuits commutables s'insère au cœur d'un logiciel de synthèse de haut niveau. On commence par présenter l'outil sélectionné puis le greffon réalisé permettant l'insertion de la fonctionnalité. Enfin, deux environnements permettant le test et la validation des circuits produits sont illustrés.

7.1.1 Logiciel hôte

Le logiciel hôte choisi pour supporter la fonctionnalité de production des circuits commutables est l'outil AUGH développé par PROST-BOUCLE et al. [PB+14], gratuit et sous licence libre. Ce logiciel a été spécifiquement conçu pour générer automatiquement des accélérateurs matériels sous contrainte de ressources. AUGH accepte en entrée un sous-ensemble du C ANSI et produit des circuits en VHDL. L'utilisateur n'est pas investi dans la transformation du circuit en vue de sa mise en œuvre matérielle, il ne fait pas de choix architecturaux. C'est le logiciel AUGH, à qui l'utilisateur fournit une contrainte en ressource et une contrainte en fréquence, qui décide des transformations à appliquer au circuit. Les circuits produits ne dépendront pas de la capacité qu'a l'utilisateur à optimiser son circuit pour la cible FPGA choisie. Cette indépendance du flot de conception vis-à-vis des connaissances de l'utilisateur est importante pour l'introduction de la fonctionnalité de commutation. Il est, dans une certaine mesure, quand même possible de piloter l'outil pour des étapes de conceptions qui ne relèveraient pas d'un algorithme : définition du niveau hiérarchique supérieur du circuit, types d'entrées/sorties et interfaces de communications, paramétrage de certains algorithmes d'élaboration et transformation, etc.

7.1.2 Développement d'un greffon

Afin d'ajouter dans le flot de développement de haut niveau les étapes nécessaires à la création de circuits commutables, un greffon a été écrit pour le logiciel hôte AUGH. Ce greffon s'appelle CP3, pour *CheckPoint PinPoint*. Il est développé en langage C et est intégré au dépôt de développement de AUGH. Il est possible d'accéder au code source de CP3 par le biais du système de gestion de version de AUGH accessible sur la page internet du projet [PB13].

Le greffon CP3 utilise la représentation intermédiaire de AUGH. À l'aide de ces informations, il est capable de mettre en œuvre les deux étapes de construction présentée précédemment (recherche des points de sauvegarde et insertion du mécanisme d'extraction de contexte) à partir de paramètres additionnels donnés par l'utilisateur. Ceux-ci sont propres à la création d'un circuit commutable et ne sont utilisés que par le greffon. Les deux paramètres principaux sont t_{lat}, la latence d'extraction du contexte de la tâche ainsi que L_m, la largeur du mécanisme d'extraction. On peut aussi préciser le type de chaîne d'extraction relative aux registres qu'on souhaite : chaînes partielles ou chaîne union. Il est possible d'activer ou désactiver le découpage des mémoires afin de limiter la fragmentation du contexte relatif aux

mémoires. Pour finir, on peut activer ou désactiver l'analyse simple de la durée de vie des mémoires.

7.1.3 Environnements de test

Deux environnements de test et de validation ont été construits. L'un permet une validation rapide en simulation et éventuellement un passage sur FPGA. Le second est plus complet et implique un système entier et exclu donc la simulation.

Plateforme programmable

La première plateforme répond à des besoins de validation par simulation fonctionnelle. En outre, la simulation n'ayant pas la valeur d'une validation sur FPGA, cette plateforme est prévue pour être synthétisable. Cette plateforme est développée en VHDL et son schéma hiérarchique est visible sur la Figure 7.1. Elle est statique dans le sens ou il n'est pas possible de changer le type de tâche matérielle que cette plateforme peut mettre en œuvre une fois celle-ci synthétisée. Une fois le circuit testé choisi, les signaux comme par exemple ceux connectant « E » et « E-chaînes » sont configurés pour avoir la taille correspondante. Cette plateforme est programmable. Un processeur miniature, « μCPU », lit les instructions se trouvant dans le bloc « programme » et les exécute. Les instructions disponibles sont `Rst, Sig_start, Ack_data, Run, Waitfor, Cp_search, Cp_save, Idle, Restore, Stop`. Cet ensemble permet de piloter le circuit lors de toutes les étapes de son fonctionnement, de la fourniture des vecteurs d'entrée (`Ack_data`), à l'attente d'un certain nombre de vecteurs de sortie (`Waitfor`) en passant bien sur par la demande de préemption (`Cp_search`) et la restauration d'un contexte préalablement enregistré (`Restore`). Le bloc de gestion des entrées et sorties sert le circuit à l'aide de valeurs préenregistrées. À la réception de données de la part du circuit, il compare les valeurs obtenues avec des valeurs de référence. Le bloc « contexte » est un simple banc de mémoire contenant quatre emplacements de stockage de contextes. C'est ce bloc qui est connecté aux ports de restauration et d'extraction et de contexte du circuit (resp. « E-chaîne » et « S-chaîne »). La synthèse de cette plateforme sur cible FPGA est possible. On a pu vérifier le fonctionnement de certaines applications du banc de test. Les avantages de cette plateforme sont sa rapidité de simulation, sa flexibilité et sa relative simplicité.

VALZY

La plateforme précédente, efficace pour tester rapidement les circuits produits, n'est pas un démonstrateur complet dans le sens où il ne permet pas de mettre en œuvre l'étendue des possibilités ajoutées par le mécanisme de commutation. La plateforme VALZY permet une utilisation beaucoup plus intensive du changement de contexte. Pour ce faire, on utilise un système de pilotage plus complet ainsi qu'une architecture plus flexible. On définit le nombre de cibles reconfigurables, ou zones reconfigurables capables d'accueillir une tâche matérielle. Chacune de ces

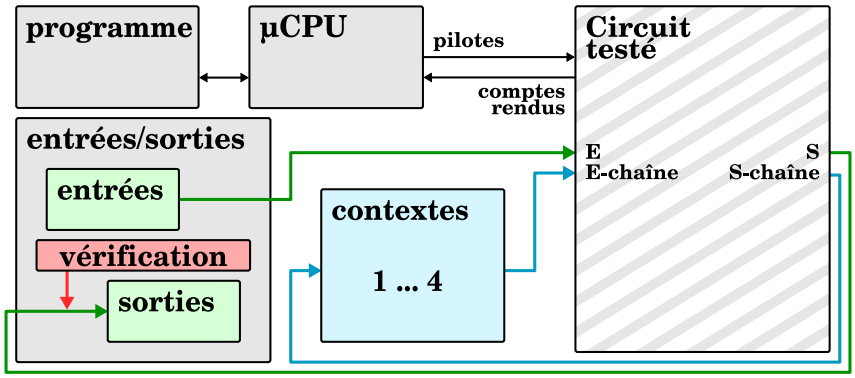
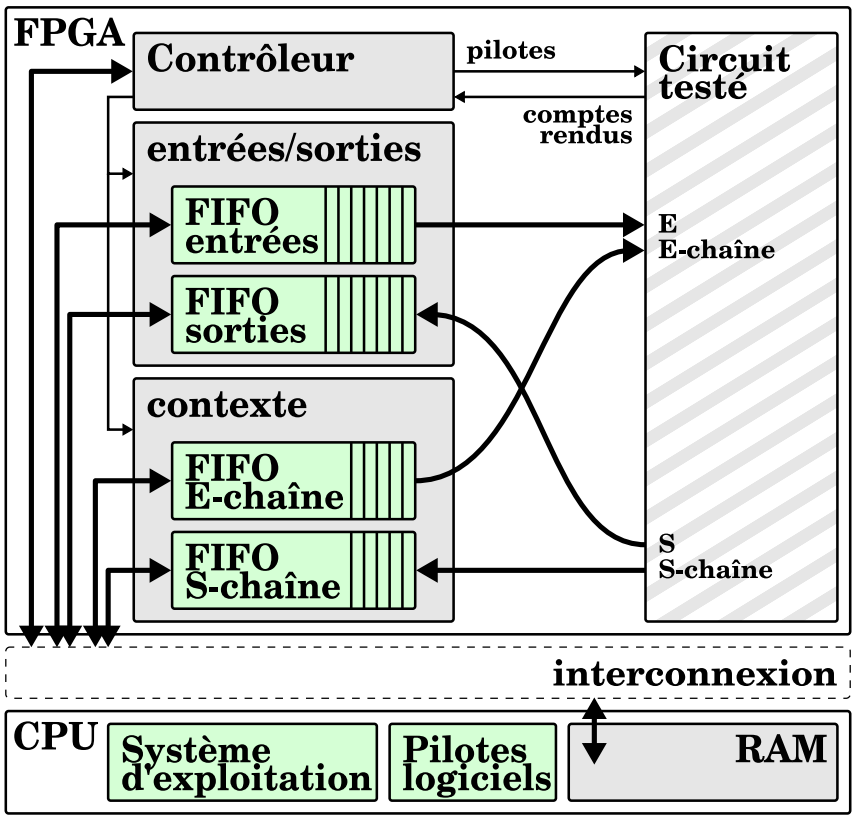


FIGURE 7.1 – Banc de test programmable et synthétisable.

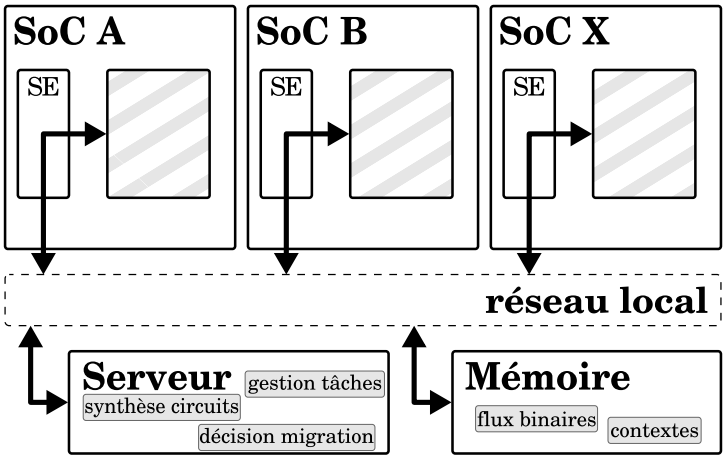
zones se voit attribuer un processeur exécutant un système d'exploitation, ce qui est sous-optimal. Cet élément de base, qui est la partie fonctionnelle du système, est piloté depuis un serveur de contrôle afin de démarrer, interrompre, migrer des tâches matérielles.

La configuration actuelle prévoit uniquement l'utilisation de cartes de type *System On Chip* ou SoC, c'est-à-dire associant un système complet (CPU, mémoire et interfaces) à une zone reconfigurable. Cette dernière est utilisée pour une seule tâche matérielle à la fois et le processeur pilote est celui présent dans le SoC. On observe l'architecture bas niveau de VALZY, c'est-à-dire au niveau d'un SoC, sur la Figure 7.2a. La partie supérieure (« FPGA ») est la partie reconfigurable du SoC. Celle-ci contient bien sûr le circuit testé mais aussi quatre blocs FIFO (« First In First Out ») chargés d'échanger des données avec le bus d'interconnexion. Ces FIFO servent d'interface entre les entrées/sorties du circuit et le bus d'interconnexion. Un bloc de contrôle est chargé de commander tous les blocs de la partie reconfigurable. Ce bloc reçoit des instructions du système d'exploitation (SE) à partir du même bus d'interconnexion. Des pilotes logiciels développés par ailleurs sont utilisés par le SE afin d'envoyer des requêtes au bloc de contrôle de la partie reconfigurable. Au niveau système, cf. Figure 7.2b, on observe la mise en réseau de plusieurs SoC. Un serveur connecté sur ce même réseau est chargé de la gestion des tâches (sélection, démarrage, vérification des vecteurs de sortie) et décide de leur préemption. En outre, il est responsable de la construction des flux binaires de configuration des SoC. Ces derniers sont mis à disposition dans une mémoire visible sur le réseau. Lorsqu'un SoC reçoit l'ordre de charger un flux, le SE le rapatrie à partir de cet espace partagé, l'écrit dans la mémoire RAM du SoC et démarre la reconfiguration. Lorsqu'une migration est demandée, le contexte extrait hors d'un circuit testé est stocké sur cette mémoire partagée. La cible chargée de redémarrer la tâche utilise alors le flot de configuration correspondant à son modèle de SoC (si nécessaire, c'est-à-dire si la tâche change de SoC hôte) et récupère puis recharge le contexte précédemment sauvegardé.

Pour l'heure, on supporte cinq cartes munies de SoC : Zybo (carte Digilent, SoC Xilinx Zynq xc7z010), Zedboard (carte AVNET, SoC Xilinx Zynq xc7z020), VC706 (carte Xilinx, SoC Xilinx Zynq xc7z045), DE1 SoC (carte Terasic, SoC Altera Cyclone



(a) Niveau SoC



(b) Niveau système

FIGURE 7.2 – Système VALZY.

V 5CSEMA5F31C6) et Arria V SoC Development Kit (carte Altera, SoC Altera Arria V 5ASTFD5K3F40I3N). Toutes les applications du banc de test ont déjà fonctionné sur au moins deux de ces plateformes, c'est-à-dire qu'elles ont effectué une migration avec succès, exception faite des dfadd, dfdiv, dfmul et dfsin. Hors du banc de test, deux applications plus graphiques ont été utilisées pour des démonstrations lors d'une conférence : un jeu de type *pong* ainsi qu'une application de traitement d'image (filtre de Sobel) [Wic+16]. Par ailleurs, la plateforme VALZY sert de base aux développements des travaux qui poursuivent cette thèse et qui sont en cours.

7.2 Résultats de la recherche des points de sauvegarde

Dans ce paragraphe, on s'intéresse aux résultats de l'algorithme de recherche des points de sauvegarde. On voit comment la latence n'est pas une donnée cruciale. Ensuite, on regarde le temps moyen d'extraction obtenu ainsi que les bénéfices de la sélection sur les objectifs d'optimisation.

7.2.1 Impact de la latence

L'un des paramètres à fournir obligatoirement au greffon CP3 est la latence maximale d'une réponse à une demande de préemption (t_{lat}). On analyse ici l'impact de ce paramètre sur l'algorithme de recherche des points de sauvegarde. En particulier, on cherche à savoir si c'est un facteur impactant la qualité des circuits produits.

Si t_{lat} est trop faible, l'algorithme de recherche des points de sauvegarde ne converge pas vers une solution car certains états sont impossibles a couvrir. Autrement dit, il est impossible d'extraire le contexte de ces états ni d'assurer que d'autres états dont le contexte est suffisamment petit les couvrent. En théorie, plus la contrainte de latence t_{lat} est relâchée, c'est-à-dire que le temps disponible pour l'extraction est élevé, moins il y aura de points de sauvegarde car ceux-ci couvriront plus d'états. Ce phénomène s'observe en effet sur la Figure 7.3. Sur cette figure est représenté le nombre de points de sauvegarde de l'application dfdiv en fonction de t_{lat}. Hormis certains minima locaux, le nombre de points de sauvegarde diminue avec l'augmentation de la latence. Après une certaine valeur de latence (200 cycles ici), on constate une stagnation du nombre de points de sauvegarde. La sélection de points de sauvegarde « forcés » est à l'origine de ce phénomène. À la fin du paragraphe 5.3.1 traitant de l'identification de la matrice $\boldsymbol{A}$, on a indiqué que le pire cas devait être pris en compte dans l'identification des couvertures. Or, une boucle d'attente de vecteurs d'entrée de type « poignée de main » empêche toute évaluation de couverture. En effet, c'est une boucle bloquante donc la condition de sortie dépend de conditions extérieures. Il est aussi possible qu'un itérateur de boucle dépende de données d'entrée ou de calculs. Dans ces deux cas, un point de sauvegarde est sélectionné à l'intérieur de cette boucle afin de satisfaire la contrainte de latence. Ce qu'on observe à partir de $t_{lat} = 200$ est donc le nombre minimal de points de

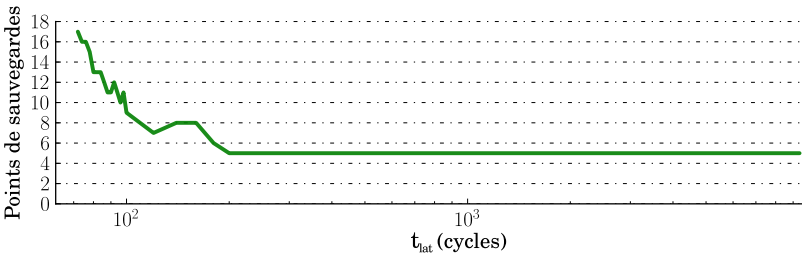


FIGURE 7.3 – Évolution du nombre de points de sauvegarde en fonction de t_{lat} pour l'application dfdiv.

sauvegarde qu'il est possible de sélectionner. La contrainte de latence n'a plus d'effet au delà. Pour l'exemple de dfdiv, qu'on ait $t_{lat} = 200$ cycles ou $t_{lat} = 2000$ cycles, le nombre de points de sauvegarde est de cinq. Le temps d'extraction maximal de l'application est dans les deux cas de 200 cycles, le même ensemble de points de sauvegarde produisant la même couverture (les cinq points de sauvegarde sont identiques à $t_{lat} = 200$ et $t_{lat} = 2000$ cycles). Ce phénomène se reproduit à l'identique pour chacune des applications du banc de test, mais avec des valeurs plancher différentes.

Dans la suite des expérimentations, on se place au dessus de cette limite afin de s'assurer que le plus petit nombre de points de sauvegarde est sélectionné. La valeur de t_{lat} est donc choisie suffisamment grande pour que ce plancher soit atteint pour toutes les applications, même les plus exigeantes en terme de temps d'extraction.

7.2.2 Premiers bénéfices

Avant même l'insertion du mécanisme d'extraction, on peut commenter les résultats de l'algorithme de recherche des points de sauvegarde. Le Tableau 7.1 rassemble les données observables après sélection. On utilise les sigles suivants, dont certains ont déjà été vus : CSP pour Chaînes de Sérialisation Partielle, qui désigne le mécanisme d'extraction différentié par point de sauvegarde ; CSC pour Chaîne de Sérialisation Complète, mécanisme utilisé uniquement pour comparaison, il contient l'intégralité des registres du circuit ; CSU pour Chaîne de Sérialisation Union, chaîne contenant l'union de tous les registres vivant dans au moins un des point de sauvegarde. Tous les résultats sont obtenus après synthèse par AUGH et CP3. Les contraintes de ressources de AUGH ont été débridées. En effet, dans un premier temps, on souhaite observer le comportement de CP3 sur des circuits dont la taille n'est pas contrainte par une cible matérielle. De plus, on choisit t_{lat} très grand afin de se situer sur la zone plancher du nombre de points de sauvegarde pour toutes les applications. Pour satisfaire les critères d'extraction de l'application jpeg, on prend donc $t_{lat} = 30000$. On constatera effectivement dans les paragraphes suivants que cette valeur est très largement surestimée pour toutes les autres applications du banc de test.

La première colonne de ce tableau contient trois nombres dont le nombre de chaînes partielles du mécanisme de type CSP, le nombre de points de sauvegarde

	# CSP /pt. de sauv./ états	ratio pt. de sauv.	CSC (bit)	CSU (bit)	gain / CSC	max CSP	moy. CSP	gain / CSC
adpcm	24/31/154	20.1%	3545	2809	21%	1336	942	73%
aes	13/24/237	10.1%	664	360	46%	128	58	91%
blowfish	7/8/123	6.5%	632	424	33%	224	113	82%
dfadd	6/7/97	7.2%	1766	673	62%	433	185	90%
dfdiv	5/5/335	1.5%	2155	721	67%	401	160	93%
dfmul	3/3/51	5.9%	885	128	86%	64	42	95%
dfsin	9/10/553	1.8%	4671	1426	69%	593	325	93%
gsm	16/23/206	11.2%	752	512	32%	160	57	92%
idct	2/2/231	0.9%	1000	40	96%	40	36	96%
jpeg	61/120/1703	7.0%	3433	2179	37%	579	320	91%
mips	5/8/42	19.0%	320	256	20%	160	120	63%
mjpeg	45/107/899	11.9%	2623	1859	29%	719	415	84%
mpeg2	38/102/356	28.7%	919	809	12%	361	283	69%
sha	13/29/106	27.4%	510	478	6%	288	152	70%
Moyennes geo.		6%			32%			73%

TABLE 7.1 – Résultats de l'algorithme de sélection des points de sauvegardes

trouvés à l'issue de l'algorithme et le nombre d'états de la FSM du circuit. La colonne suivante est le ratio du nombre de points de sauvegarde sur le nombre d'états constitutifs d'une tâche. Ce ratio peut-être très petit (pour l'idct, seulement $0,9\%$ des états sont des points de sauvegarde) mais est toujours inférieur à 30%. Les colonnes suivantes traitent du contexte relatif aux registres car c'est sur ces éléments mémorisants qu'est effectuée l'analyse de la durée de vie. On constate qu'une CSU permet de réduire la taille du contexte de 32% en moyenne. En utilisant la taille moyenne d'une CSP et en supposant que les points de sauvegarde sont rencontrés de manière homogène, on obtient une réduction moyenne de 73% pour les CSP. La colonne « max CSP » est la taille de la plus grande CSP. Le chapitre précédent indique que ces données ne suffisent pas à caractériser le circuit obtenu. En effet, les différents types de chaînes qu'on choisit par la suite impactent fortement les effets du mécanisme sur le circuit.

7.2.3 Temps moyen d'extraction

L'extraction d'un contexte hors d'une tâche matérielle est réalisée sous contrainte de latence. Le temps que met une ressource matérielle à être libérée dépend en partie de cette latence sauf à partir de la zone plancher définie dans le paragraphe précédent. On place donc le curseur t_{lat} dans cette zone pour toutes les applications du banc de test et on observe le temps moyen que met une application avant de libérer la ressource matérielle, c'est-à-dire la somme des temps t_c et t_s, respectivement temps nécessaire pour atteindre un point de sauvegarde et temps nécessaire à la sauvegarde du contexte propre au point de sauvegarde.

Afin d'obtenir une estimation de ce temps, on fait l'expérience suivante. Pour obtenir une donnée pertinente sur le temps d'extraction, on effectue une demande de préemption de cette tâche à chacun des cycles de son exécution. Pour cela, on utilise la plateforme VALZY. La tâche matérielle est démarrée, ses vecteurs d'entrée lui sont fournis, le système effectue une demande de préemption à un cycle déterminé et on enregistre le temps nécessaire jusqu'à la fin de l'extraction du contexte de la tâche. On répète l'opération en décalant la demande de préemption d'un cycle et ainsi de suite jusqu'à la durée complète d'exécution de la tâche. On obtient ainsi, pour un jeu de vecteurs d'entrée, les données permettant de calculer un temps moyen d'extraction. Pour obtenir une valeur sans biais, il faudrait répéter l'opération sur un nombre conséquent de jeux de vecteurs d'entrée.

L'expérience a été menée sur une carte VC706 de Xilinx. Les applications sur virgule flottante ne font pas partie de l'expérience car le bus utilisé actuellement par VALZY (AXI esclave) est de 32 bits de large. Or les vecteurs d'entrée de ces applications font 64 bits de large et aucun sérialiseur désérialiseur n'a été développé.

Les résultats de cette expérience sur les applications du banc de test sont visibles sur le Tableau 7.2. On remarque que pour la plupart des applications t_c est petit devant t_s. Le contexte sauvegardé se compose, en nombre de cycles d'extraction, du contexte relatif aux registres (colonne CSP) et du contexte relatif aux mémoires (colonne CSM). La remarque précédente est confirmée par l'examen de la dernière colonne. On note en effet que la plupart du temps d'extraction est consacré à la sauvegarde du contexte plutôt qu'à l'attente de rencontre d'un point de sauvegarde. Cela illustre le degré de liberté finalement assez faible qu'il est possible d'obtenir lors de la sélection des points de sauvegarde : ce sont les nids de boucles, représentant la plupart du temps d'exécution d'une tâche matérielle, dont le point de sauvegarde aura été forcé afin de pallier le pire des cas, qui sont le plus souvent rencontrés. On observe le phénomène suivant : la recherche des points de sauvegarde n'a qu'un impact faible dans la réduction du temps d'extraction de contexte de la tâche matérielle. Néanmoins, son impact positif sur les autres objectifs d'optimisation est vu dans les paragraphes suivants. On note que le temps d'extraction a été calculé sur un groupement de type CSP. C'est donc une moyenne de temps d'extraction du contexte relatif aux registres qu'on obtient, et qui est calculée grossièrement à partir des données du Tableau 7.1. Un chiffre remarquable concerne le ratio du mpeg2, supérieur à 100% du temps moyen d'extraction. On explique simplement ce phénomène car la chaîne partielle empruntée le plus souvent possède une taille supérieur à la moyenne des CSP de cette tâche.

7.3 Surcouts

Ce paragraphe traite des surcouts divers liés à l'insertion du mécanisme d'extraction de contexte et de ses options. On sélectionne définitivement le type de groupement de chaînes à réaliser. On étudie ensuite les surcouts incrémentaux liés aux différents composants du mécanisme (CSU et CSM) puis ceux liés à l'option de découpage des mémoires.

	t_c moy.	t_s moy.	$t_c + t_s$ moy.	CSP	CSM	total	ratio
adpcm	3	441	444	30	394	424	95%
aes	13	653	666	2	644	646	97%
blowfish	14	1091	1105	4	1082	1086	98%
gsm	5	193	198	2	186	188	95%
idct	47	132	179	2	128	130	73%
jpeg	3	28330	28333	10	28302	28312	100%
mips	3	103	106	4	96	100	94%
mjpeg	10	14481	14491	13	14443	14456	100%
mpeg2	2	2066	2068	9	2060	2069	100%+
sha	2	111	113	5	101	106	94%

TABLE 7.2 – Temps d'extraction moyen obtenu avec $L_m = 32$ et t_{lat} grand. Toutes les valeurs sont exprimées en nombre de cycles.

7.3.1 Comparaison entre CSP et CSU

Une fois la sélection des points de sauvegarde effectuée, il faut décider du mécanisme à mettre en place et plus particulièrement quel type de groupement effectuer entre les variables propres aux contextes liés aux registres de tous les points de sauvegarde. On a évoqué la possibilité d'adopter un mécanisme de chaînes parallèles, qui n'a pas d'incidence sur le groupement à choisir. Dans ce paragraphe, on fixe le parallélisme du mécanisme d'extraction à $L_m = 32$ bits. On étudiera dans un prochain paragraphe l'impact de ce paramètre car on considère qu'il ne favorise aucun groupement. On s'intéresse donc plus particulièrement aux deux groupements présentés en 6.2.1, c'est-à-dire aux Chaînes de Sérialisation Partielles (CSP) et à la Chaîne de Sérialisation Union (CSU).

Afin d'obtenir les données relatives au surcout des deux types de groupement, on effectue un traitement sur toutes les applications du banc de test. Pour chaque application, le circuit de référence est construit avec AUGH sans ajout de la part de CP3 et suivi d'une synthèse à l'aide de l'outil ISE 14.7 du constructeur Xilinx. Ensuite, on utilise le flot complet AUGH et CP3 une fois avec l'option permettant de réaliser le groupement en CSU et une autre fois avec le groupement CSP. On obtient les résultats présentés sur la Figure 7.4. Ces premiers résultats montrent donc non seulement la différence entre CSU et CSP mais aussi la différence entre un circuit « nu » et un circuit doté de la capacité de changement de contexte.

Toutes les valeurs sont normalisées par rapport au nombre de LUT de l'application sans capacité de commutation, ceci afin de pouvoir comparer les applications entres elles. La quantité de FF est inférieure à la quantité de LUT. C'est l'outil AUGH qui a tendance à créer ce déséquilibre. Sachant que la plupart des constructeurs de FPGA proposent sinon plus de FF que de LUT, en général au moins autant de l'une que de l'autre, on peut considérer que la variation du nombre de FF n'est pas une donnée cruciale dans le cas présent. La plupart des applications possèdent un surcout situé entre $1,0$ et $1,5$ fois le nombre de LUT du circuit original. L'exception

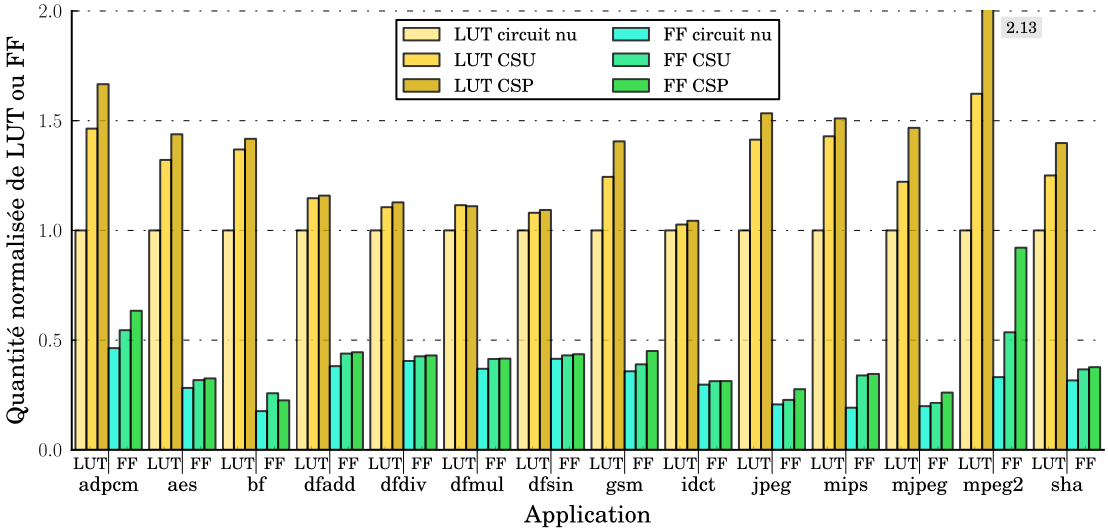


FIGURE 7.4 – Mise en évidence des surcouts liés aux CSU et CSP.

	LUT					FF				
	CSC	CSU	%	CSP	%	CSC	CSU	%	CSP	%
adpcm	9997	9556	−4%	10941	9%	3853	3659	−5%	4263	11%
aes	3597	3277	−9%	3951	10%	1047	884	−16%	926	−12%
blowfish	4384	4291	−2%	4393	0%	920	913	−1%	945	3%
dfadd	5184	4556	−12%	4603	−11%	1954	1743	−11%	1767	−10%
dfmul	2716	2113	−22%	2104	−23%	981	785	−20%	788	−20%
gsm	3365	3247	−4%	3649	8%	1086	1036	−5%	1203	11%
idct	5392	4908	−9%	4864	−10%	1508	1460	−3%	1471	−2%
jpeg	29365	31421	7%	34435	17%	5646	5301	−6%	6470	15%
mips	1626	1602	−1%	1711	5%	432	369	−15%	377	−13%
mjpeg	20638	20717	0%	23311	13%	3819	3791	−1%	4635	21%
mpeg2	3479	4047	16%	5389	55%	1359	1347	−1%	2345	73%
sha	2223	2318	4%	2640	19%	679	683	1%	703	4%
Moyenne			**–3,0%**		**7,8%**			**–6,8%**		**6,7%**

TABLE 7.3 – Comparaison entre les chaînes de sérialisation CSC, CSU et CSP pour $L_m = 32$ bits.

à cette observation concerne l'application mpeg2 dont le nombre de LUT est $2,13$ fois plus élevé pour le circuit dont les chaînes sont de type CSP. D'autre part, ce qui avait été envisagé à l'aide du Tableau 6.2 quant aux couts relatifs entre CSU et CSP se confirme : mis à part pour l'application dfmul, regrouper les chaînes de sérialisation dans des CSP se révèle toujours plus couteux que de les regrouper dans une CSU.

Pour finir, on différencie les trois techniques de groupement de chaînes c'est-à-dire CSU et CSP, qu'on vient de comparer, mais aussi la technique de la CSC, pour Chaîne de Sérialisation complète. On observe ces trois mécanismes sans extraction des mémoires afin de se focaliser sur les chaînes de sérialisation du contexte relatif au registre. Enfin, on choisit $L_m = 32$ bits. Les circuits construits ne sont pas fonctionnels du point de vue de l'extraction de contexte, la présence des CSM étant indispensable. Le Tableau 7.3 recense les données obtenues après synthèse avec ISE. Les circuits dfdiv et dfsin ne sont pas présents car le groupement CSC n'est pas réalisable (cf. Algorithme 3 ligne 14). Très logiquement, c'est la CSU qui est la moins chère des solutions avec en moyenne 3% de LUT en moins et $6,8\%$ de FF en moins qu'une CSC. La CSC, justement, est en position intermédiaire. C'est la CSP qui est le plus couteux des groupements avec en moyenne un surcout de $7,8\%$ de FF et $6,7\%$ de LUT par rapport à une CSC. Malgré une sélection des registres, rendre les chaînes spécifiques à chaque point de sauvegarde rend le mécanisme plus couteux qu'une chaîne de sérialisation complète. Les CSP possèdent toujours l'avantage d'extraire un contexte relatif aux registres minimaliste.

Compte tenu du gain finalement assez faible que représente les CSP sur la quantité de données à extraire (cf. Tableau 7.1), nous considérons comme plus adapté à la réalisation de nos objectifs le groupement de type CSU. C'est donc ce groupement que nous utilisons dans la suite des expériences.

7.3.2 Surcout matériel incrémental

Étant donné l'utilisation d'un mécanisme d'extraction de contexte dont le mécanisme d'extraction lié aux registres est groupé sous forme d'une CSU. Ce paragraphe cherche à mettre en évidence les couts incrémentaux associés aux différentes techniques nécessaires à l'extraction de contexte ainsi qu'à l'amélioration de ce mécanisme. On compare ici de manière plus fine les différences entre un circuit nu et un circuit capable de commuter. La valeur de référence pour tous les ajouts est bien entendu le circuit nu. On construit ensuite, de la même manière que dans le paragraphe précédent, trois circuits avec des capacités incrémentales. Les deux premiers circuits, non fonctionnels du point de vue de l'extraction de contexte, sont construits sans mécanisme d'extraction du contexte relatif aux mémoires. Le premier adopte une chaîne de sérialisation simple dont $L_m = 1$. Le second met en œuvre une chaîne de sérialisation parallèle avec $L_m = 32$. Le dernier circuit est fonctionnel du point de vue de la commutation. En effet, il intègre un mécanisme d'extraction de contexte relatif aux mémoires et une chaîne de sérialisation parallèle avec $L_m = 32$.

Les résultats en surface de ces quatre circuits sont visibles sur la Figure 7.5. De la même manière que pour le paragraphe précédent, toutes les valeurs en nombre

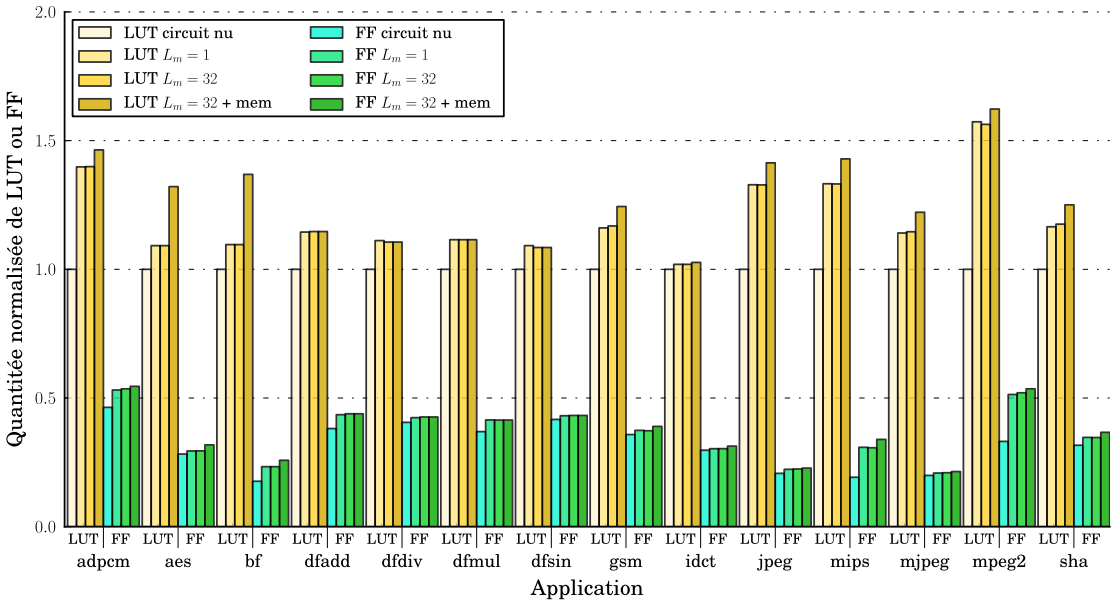


FIGURE 7.5 – Mise en évidence des surcouts liés au différentes composantes du mécanisme d'extraction de contexte.

de LUT et FF sont normalisées par rapport à la quantité de LUT d'un circuit nu. Les valeurs obtenues pour le nombre de FF sont aussi d'une importance moindre. L'évolution de l'équipement d'un circuit est généralement associée à une surface matérielle croissante tant en LUT qu'en FF. Le blowfish, abrégé en *bf* sur la figure, ainsi que l'aes présentent un pic lié à l'insertion des mémoires dans le mécanisme d'extraction. On explique ce surcout de la manière suivante : lors de la synthèse, l'outil (ici ISE) peut procéder à des simplifications qui ne sont pas mises en œuvre dans AUGH. Dans le cas du blowfish et de l'aes, c'est une mémoire dont la taille est réduite car l'outil de synthèse détecte une plage d'adresses non utilisée. Lors de la synthèse de haut niveau, première étape du flot de conception, AUGH ne détecte pas la non utilisation de cette plage et laisse la mémoire entière. Lors de l'insertion d'un mécanisme d'extraction ne traitant que les registres, les mémoires sont laissées telles quelles et l'outil de synthèse peut procéder à la simplification. Lorsque CP3 met en place le mécanisme d'extraction des mémoires, l'outil de synthèse est cette fois dans l'incapacité de simplifier la mémoire non utilisée, sa plage d'adresses étant dans ce cas parcourue au moins une fois dans son intégralité lors de son extraction. La solution à ce problème est d'implémenter un mécanisme de simplification des mémoires dès la synthèse de haut niveau dans l'outil hôte, mais comme on l'a vu précédemment, c'est une procédure complexe. Sauf ces cas particulier (bf et aes), c'est bien le mécanisme d'extraction lié aux registres qui représente la majeure partie du surcout en LUT. De plus, équiper le circuit d'une chaîne de sérialisation simple ($L_m = 1$) ou parallèle ($L_m = 32$) est pratiquement équivalent du point de vue du surcout en LUT.

	mémoires	taille totale (en bit)	mémoires découpées	mémoire découpées (en bit)	ratio
adpcm	14	12608	0	0	0%
aes	6	20608	0	0	0%
bf	6	34112	4	768	2%
gsm	6	3408	5	3120	92%
idct	2	2560	1	512	20%
jpeg	29	371177	4	255472	69%
mips	2	3072	0	0	0%
mjpeg	27	291600	7	68992	24%
mpeg2	3	16768	1	16384	98%
sha	3	3232	0	0	0%

TABLE 7.4 – Résultat des découpes sur l'ensemble des applications possédant des mémoires pour $L_m = 32$ bits.

7.3.3 Découpage de la chaîne de mémoires

La technique d'empilement des mémoires vue en 6.2.2 n'est pas suffisante afin de réduire de façon admissible la fragmentation du contexte relatif aux mémoires. La technique de découpe des mémoires permet de résoudre ce problème et on obtient grâce à elle un circuit presque exempt de contexte mémoire fragmenté (cf. Tableau A.1 visible en annexe). Le Tableau 7.4 recense les résultats de la méthode de découpe sur tous les circuits du banc de test qui possèdent une ou plusieurs mémoires. On reporte dans cette table le nombre de mémoires et leur taille totale. On applique une politique de découpe des mémoires dans le cas où le mécanisme a pour caractéristique $L_m = 32$ bits. Les trois dernières colonnes représentent le nombre de mémoires ayant subi une découpe, la quantité de mémoire que concerne cette découpe et enfin le ratio de la mémoire faisant parti d'une mémoire découpée par rapport à la mémoire n'ayant pas été découpée. Les applications n'ayant subi aucune découpe (adpcm, aes, mips et sha) ont des mémoires dont la largeur est de 32 bits. Les découper n'est donc pas utile. C'est sur les applications restantes qu'on étudie le surcout matériel lié à la technique.

La Figure 7.6 propose une mise en évidence des résultats de synthèse des circuits dont les mémoires ont subi une découpe. On compare des circuits construits avec AUGH sans apport de CP3, puis synthétisé avec l'outil ISE 14.7 du constructeur Xilinx, avec des circuits ayant subi une découpe des mémoires (pas de sélection de points de sauvegarde ni de construction de mécanisme d'extraction de contexte) puis une synthèse avec le même outil sur la Figure 7.6a. On procède à une comparaison similaire sur la Figure 7.6b sauf que les circuits subissent l'ajout d'un mécanisme d'extraction de contexte ainsi qu'une sélection de points de sauvegarde dans les deux cas. Toutes les valeurs sont normalisées par rapport au nombre de LUT de l'application sans découpe, ceci afin de pouvoir comparer les applications entres

elles. On remarque tout d'abord que les surcouts en LUT sont situés entre -5% et $+16\%$. L'accroissement du nombre de LUT est très faible voire négatif dans le cas de l'idct. On remarque qu'il est le plus conséquent pour le gsm. On explique cela par le fait que cette application est particulièrement petite. La découpe des mémoires ajoutant un nombre même faible de LUT s'en ressent. En mettant en relation la dernière colonne du Tableau A.1 visible en annexe et les surcouts, on observe que la quantité de mémoires impliquée dans une découpe est très peu corrélée au surcout final.

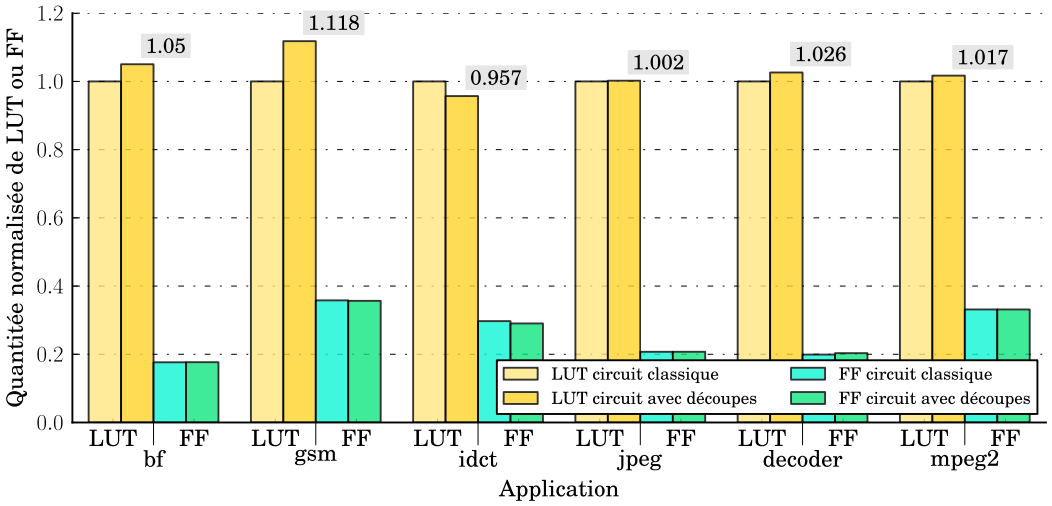
7.4 Performances

Cette partie traite des performances des circuits produits ainsi que des performances de la méthode de production. On étudie en premier lieu les résultats en fréquence des circuits. Enfin, on comparera les résultats obtenus par la méthode proposée aux plateformes existantes dans l'état de l'art.

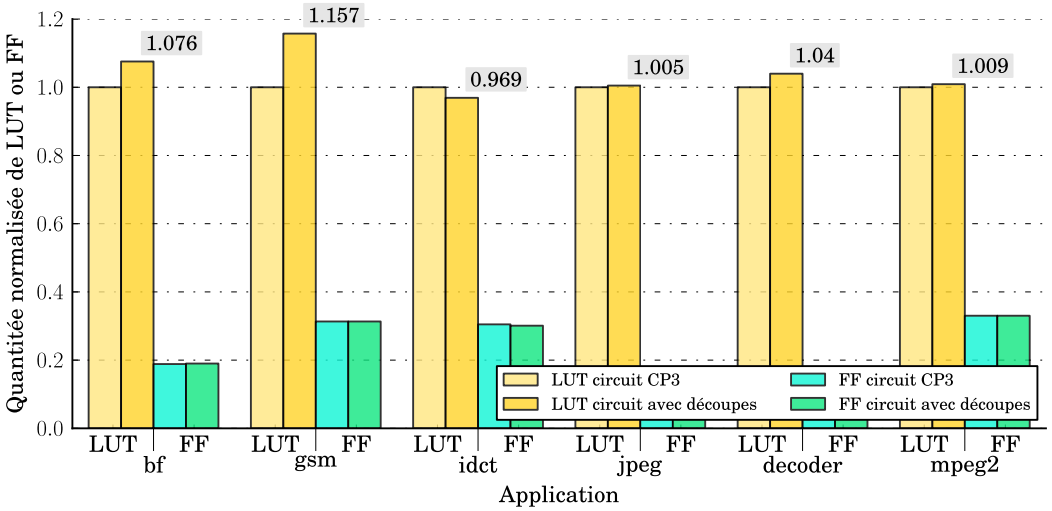
7.4.1 Fréquence des circuits obtenus

Le principal indice de performance des circuits produits par le flot de synthèse de haut niveau est la fréquence de fonctionnement finale. La latence du circuit, autre indice de performance majeur, n'est pas impactée par l'utilisation du flot modifié. En effet, les transformations appliquées par AUGH réduisant la latence du circuit à l'aide de parallélisation par exemple, le sont avant la recherche des points de sauvegarde et l'ajout du mécanisme d'extraction de contexte. Seul l'ajout du mécanisme d'extraction de contexte impacte physiquement le circuit produit et plus particulièrement son chemin critique. D'une part, l'ajout de matériel au sein d'un circuit induit, sauf cas particuliers, une plus grande difficulté de résolution de l'algorithme de placement et routage. Cela implique généralement une distance accrue entre les éléments logiques constitutifs d'un circuit et donc un chemin critique potentiellement plus grand. D'autre part, si l'ajout de matériel se situe sur le chemin critique, cela implique un allongement du temps de propagation des signaux parcourant ce chemin. Ces deux phénomènes s'additionnent et on émet donc l'hypothèse que l'ajout du mécanisme d'extraction de contexte ne peut que diminuer la fréquence de fonctionnement du circuit.

Les résultats en terme de fréquence sont visibles sur le Tableau 7.5. Ces résultats sont obtenus après placement et routage avec l'outil ISE. On remarque immédiatement que l'hypothèse formulée précédemment n'est pas valide : la fréquence de fonctionnement de quatre circuits est améliorée (pour l'adpcm, le gsm, le jpeg et le mpeg2). En réalité, les outils de synthèse sont capables de comprendre les circuits générés par AUGH mais de manière basique. Pour déterminer le chemin critique, l'outil de synthèse détermine le plus long chemin combinatoire existant entre deux registres. Ce chemin peut n'être jamais utilisé en pratique dans le circuit. Illustrons ce cas à l'aide de la Figure 7.7. On observe un composant de multiplication partagé entre quatre registres. En pratique, seule la multiplication des registres ● entre



(a) Sans mécanisme d'extraction de contexte



(b) Avec mécanisme d'extraction de contexte

FIGURE 7.6 – Surcouts matériels en LUT et FF liés à la découpe des mémoires.

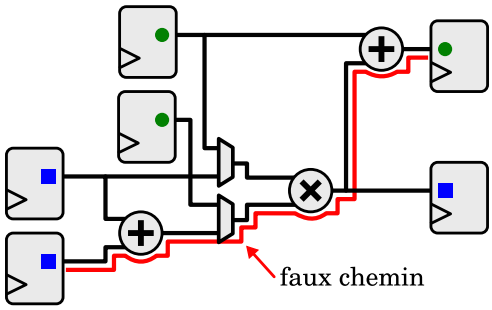


FIGURE 7.7 – Illustration des faux chemins pris en compte par l'outil de synthèse.

eux et des registres ■ entre eux est permise : la machine d'état contrôle les multiplexeurs nourrissant le multiplieur et le registre chargé de recevoir le résultat. Le chemin représenté, reliant un registre ■ à un registre ●, ne peut donc jamais être utilisé lorsque le circuit fonctionne. C'est un faux chemin combinatoire. Cette information, qu'il est possible d'extraire de la machine d'état, n'est plus disponible après la synthèse logique. L'outil chargé d'évaluer les temps de propagation est donc obligé de prendre en compte ce cas en pratique impossible : le chemin critique est surestimé.

Pour obtenir des circuits dont la fréquence estimée est meilleure, il faut éviter la production de faux chemins. Il est possible, grâce à AUGH, d'interdire le partage d'opérateurs tel qu'il est pratiqué pour le multiplieur de la Figure 7.7. Interdire ce partage couvre une partie des cas d'apparition de faux chemins. Il reste toutefois possible d'en créer avec des mémoires asynchrones, celles-ci qui pouvant être partagées de la même façon que les opérateurs. Le Tableau 7.5 présente donc les résultats obtenus avec cette option. On peut comparer les résultats avec le Tableau C.1 visible en annexe pour se convaincre de la disparition d'au moins une partie des faux chemins. Certains circuits ne sont pas impactés par l'ajout d'un mécanisme d'extraction (blowfish, dfmul, dfsin, idct). L'équipement du aux chaînes d'extraction n'est donc pas situé sur le chemin critique identifié par l'outil de synthèse. À l'inverse, les cas extrêmes comme celui du mjpeg montrent que l'ajout du mécanisme d'extraction est probablement très pénalisant pour le chemin critique (réduction de moitié de la fréquence de fonctionnement du mjpeg par exemple). Les cas dans lesquels la fréquence est améliorée sont une illustration de la variabilité des résultats produits par les outils de synthèse pour FPGA : les circuits équipés à l'aide de CP3 peuvent bénéficier d'optimisations, ou le placement routage s'effectue de manière plus avantageuse.

7.4.2 Rapidité de l'obtention des circuits

Un des avantages, qui n'a pas encore été cité, de l'utilisation d'un flot de HLS est la rapidité d'obtention de circuits fonctionnels et la plus grande facilité d'exploration architecturale complexe possible. AUGH est un outil de HLS rapide. L'addition des étapes de construction propres à CP3, c'est-à-dire la recherche des points de sauvegarde, l'insertion du mécanisme d'extraction de contexte et pourquoi pas, l'analyse

	Chem. crit. (ns)		Fréquence max. (MHz)		
	nu	CSU	nu	CSU	diff.
adpcm	12.9	12.5	77.6	80.2	3%
aes	5.2	5.2	192.1	191.1	−1%
bf	5.8	5.8	171.9	171.9	0%
dfadd	8.1	8.6	122.8	115.7	−6%
dfdiv	n.d	n.d	n.d	n.d	n.d
dfmul	9.7	9.7	102.8	102.8	0%
dfsin	11.1	11.1	89.8	89.8	0%
gsm	13.9	13.3	71.9	75.4	5%
idct	8.8	8.8	113.9	113.9	0%
jpeg	31.6	29.2	31.6	34.3	8%
mips	5.8	7.6	173.0	131.7	−24%
mjpeg	15.7	29.6	63.9	33.8	−47%
mpeg2	26.9	23.7	37.1	42.2	14%
sha	7.8	8.1	128.7	124.1	−4%

TABLE 7.5 – Chemin critique et fréquence maximale obtenues avec et sans CP3, avec option limitant le partage d'opérateur.

simple de la durée de vie des mémoires, n'impactent que de manière raisonnable le temps de synthèse des circuits. On peut le vérifier grâce à la Figure 7.8. Dans la plupart des cas, le temps d'exécution de CP3 n'excède pas 20% du temps total passé dans le flot de HLS. L'exécution de AUGH seule représente entre 70% et 80% du temps d'exécution en général. On remarque l'exception du jpeg, dominé par l'analyse simple de la durée de vie des mémoires.

En plus de ne nécessiter qu'un faible effort de la part du développeur d'application dans le cas ou celui-ci souhaite une analyse simple de la durée de vie des mémoires (insertion du pragma), l'exécution de CP3 impacte de manière faible le flot de conception HLS propre à AUGH. Dans tous les cas, le temps d'exécution du logiciel AUGH, dont le temps d'exécution hors CP3 est visible en annotation de la Figure 7.8, reste très nettement inférieur au temps d'exécution des outils de synthèse.

7.4.3 Comparaison du mécanisme avec l'état de l'art

Le dernier point abordé dans cette partie concernant les performances est comparatif. Afin de placer les résultats obtenus par rapport à l'état de l'art, on regarde dans un premier temps comment on se situe par rapport aux plateformes citées en Section 3. Ensuite, on observe plus précisément une référence mettant en œuvre un mécanisme proche du nôtre.

La Tableau 7.6 présente les précédentes propositions de plateformes dont l'objectif principal est d'obtenir des circuits commutables, même si les motivations sont diverses. Toutes ces références proposent donc un système fonctionnel et relativement complet, parfois assorti d'un démonstrateur. Les données présentées concernent la technique utilisée afin d'extraire le contexte d'une tâche matérielle, les pertes de performances engendrées ainsi que le surcout matériel subi et le temps nécessaire afin de sauvegarder le contexte d'une tâche. Quelques remarques com-

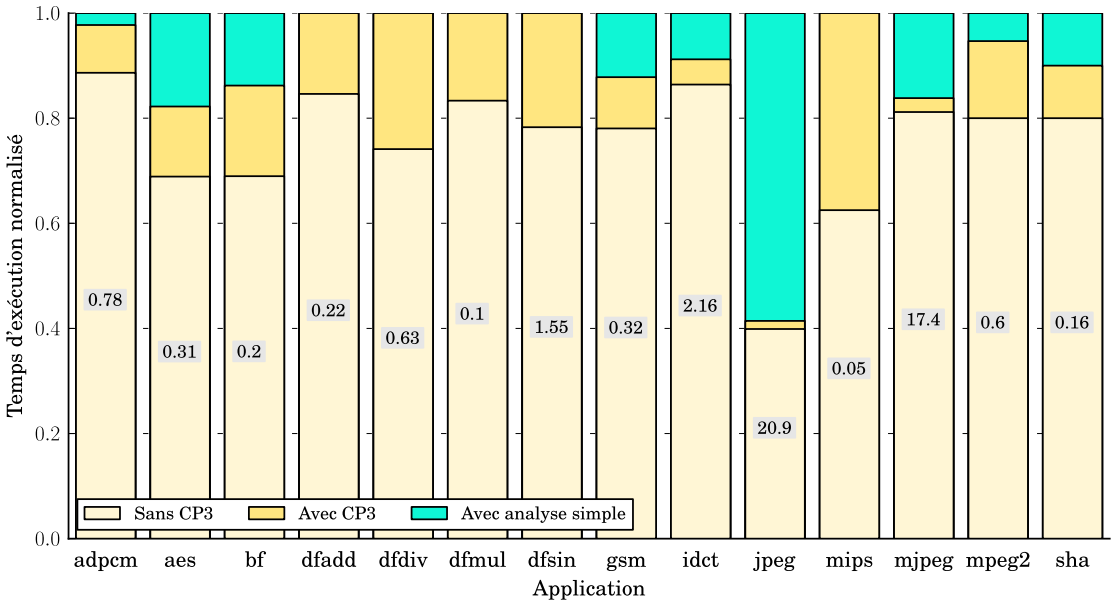


FIGURE 7.8 – Temps d'exécution normalisés du logiciel AUGH et du greffon CP3 avec annotation du temps d'exécution (en s) de AUGH seul.

plètent les informations disponibles. Les plateformes dont la technologie d'extraction est embarquée dans la description de la tâche ont les temps d'extraction (t_s) les plus faibles et de très loin. Le surcout matériel et la dégradation des performances sont en revanche nuls pour les technologies à relecture du flot de configuration. Les technologies embarquées, qui sont les plus facilement comparables à notre méthode, présentent des surcouts matériels similaires, bien que les applications utilisées par ces auteurs ne présentent généralement pas la complexité des applications du banc de test CHStone. La dégradation de la fréquence de fonctionnement n'est soit pas communiquée (« N.C ») soit située entre 10% et 30%. Dans [Jov+07], l'usage d'une FF spécifique à l'extraction de contexte pénalise notablement les performances du circuit.

Il est possible de faire une comparaison plus précise mais aussi plus équitable vis à vis de [Joz+12]. En effet, ce dernier a mis en œuvre (entre autre) un des mécanismes présentés par [Koc+07] qui est le mécanisme embarqué nommé *Memory Map* (MM) ainsi qu'un mécanisme de *readback*. De plus, trois applications sont communes entre notre banc de test et le leur ce qui rend la comparaison plus cohérente. Il reste néanmoins que leurs circuits sont construits à la main ce qui fatalement leur donne un avantage. On va donc comparer ces deux mécanismes au nôtre. Les circuits construits à l'aide de notre méthode sont relativement simples : $L_m = 32$ bits, on groupe les chaînes dans une CSU et les mémoires ne sont ni analysées, ni découpées. C'est donc un cas peu favorable du point de vue de la rapidité d'extraction du contexte. On retrouve dans le Tableau 7.7 des données concernant les applications gsm, idct et sha. On observe plus particulièrement la taille du contexte à extraire lors d'une préemption ainsi que t_s, le temps de

Référence	tech.	perf.	surcout	t_s	remarques
[Sim+00]	readback		aucun	$7,5$ à $14,4$ ms	Filtrage du flux de config. Cible Xilinx XCV400
[Lan+02]	readback		aucun	407 ms	Filtrage du flux de config. Cible SLAAC-V1
[Ull+04]	readback		aucun	118 kB via ICAP	reconfiguration partielle, bus CAN
[Hua+07]	embarqué	N.C	$\sim 30\%$	$O(n)$ $n =$nb. FF	Modification manuelles du code source, pas de gestion des mémoires
[Jov+07]	embarqué	-30%	$\sim 30\%$	$O(n)$ $n =$nb. FF	Usage d'une *preemptive FF*, pas de gestion des mémoires, manuel
[Joz+12]	embarqué	-10%	$\sim 100\%$	~ 1 ms	Usage d'un *Memory Map*, automatique
[MV+13]	readback		aucun	5 à 13 ms	Extraction par colonne, cible XUPV5-LX110T
[Hap+15]	readback		aucun	10 à 16 ms	Intégré dans ReconOS, cible Virtex 6 ML605

TABLE 7.6 – Comparaison à l'état de l'art.

		[Joz+12] *readback*	[Joz+12] *memory map*	Notre méthode
Taille contexte	gsm	99.7 kbit	5.37 kbit	6.50 kbit
	idct	49.6 kbit	1.25 kbit	4.20 kbit
	sha	47.5 kbit	1.65 kbit	3.74 kbit
t_s	gsm	299 µs	n/a	2.7 µs (203 cycles @ 75 MHz)
	idct	151 µs	266 µs	1.2 µs (131 cycles @ 113 MHz)
	sha	144 µs	351 µs	0.94 µs (117 cycles @ 124 MHz)

TABLE 7.7 – Comparaison avec [Joz+12]

sauvegarde de ce contexte. La taille du contexte des applications construites avec CP3 a été obtenue en ajoutant la taille des CSM à la taille de la CSU. Sans conteste, c'est la méthode de relecture qui possède le contexte le plus conséquent malgré la mise en place d'une relecture partielle. Les contextes des techniques MM et de la nôtre sont comparables. Ceux-ci ont une taille de l'ordre de quelques kilobits. Les contextes identifiés avec notre méthode sont plus importants mais cela s'explique par la construction automatique des circuits. Les registres de travail ne sont ni économisés ni partagés dans l'état actuel de AUGH. Le point le plus désavantageux de la méthode MM est sa lenteur d'extraction. Malgré un contexte environ cinquante fois moins volumineux, il est plus long à l'extraction que lorsque la technique de relecture est utilisée ! Les circuits construits avec CP3 sont deux ordres de grandeurs plus rapides à l'extraction de contexte. L'objectif d'optimisation impactant le système (rapidité d'extraction, petite taille de contexte) est donc atteint.

7.5 Analyse des mémoires

Ce dernier paragraphe présente les résultats des expériences menées sur le mécanisme d'analyse simple des mémoires. On observe une métrique permettant d'estimer le gain obtenu. Enfin, on calcule le cout que représente l'extraction des mémoires différentiées selon le point de sauvegarde.

7.5.1 Gain de l'analyse simple

L'analyse simple de la durée de vie des mémoires implique de la part du développeur de valider l'hypothèse de non réutilisation évoquée en 6.4.2. On peut estimer grossièrement la plage d'utilisation des mémoires remplissant ce critère. Il est donc possible de retirer du contexte de certains points de sauvegarde des mémoires sous condition. Afin d'estimer le gain apporté par la suppression des mémoires non vivantes, on fait l'addition de la taille du contexte de chacun des points de sauvegarde dans le cas où on n'analyse pas les mémoires et dans le cas où on pratique une analyse simple de la durée de vie des mémoires. Ce sont les données présentées par la Figure 7.9. Ces résultats sont à analyser avec prudence, car ils ne favorisent pas les points de sauvegarde dont la probabilité de rencontre est la plus importante. Ceux-ci, situés au cœur des boucles de calcul, sont non seulement empruntés en moyenne plus souvent mais ils nécessitent en général l'extraction d'un plus grand contexte, la plupart des mémoires étant alors vivantes. Néanmoins, on observe que la taille moyenne des points de sauvegarde diminue (le nombre de points de sauvegarde étant constant malgré l'analyse). Pour obtenir des données plus précises, il faut reproduire une expérience similaire à celle menée en 7.2.3. Cette expérience étant couteuse en temps, elle n'a pas été faite.

Le gain est notable dans le cas d'application dont la taille cumulée des contextes est conséquente : jpeg, mjpeg, mpeg2. Pour d'autres application, on peut considérer l'analyse comme n'ayant qu'une influence faible (blowfish, sha).

7.5.2 Cout de l'analyse simple

Le cout de l'analyse simple de la durée de vie des mémoires sur le circuit produit à l'issue du flot de synthèse est observé dans ce paragraphe. Analyser les mémoires implique qu'il faille adapter le mécanisme des CSM. Initialement, toutes les CSM sont utilisées à chaque changement de contexte (extraction comme restauration). L'analyse de la durée de vie des mémoires écarte certaines CSM des chaînes à extraire pour certains points de sauvegarde. Le surcout de l'analyse simple est donc issu de la logique supplémentaire qu'il faut ajouter au circuit pour sélectionner les CSM correspondant à l'extraction ou restauration propre à un point de sauvegarde. De la même manière que le groupement de type CSP a un cout supplémentaire en logique dans la machine d'état, n'extraire qu'un sous-ensemble des CSM induit une utilisation de logique plus importante.

La Figure 7.10 expose le surcout engendré par cette analyse. On compare les circuits qui ont subi l'ajout du mécanisme d'extraction sans analyse simple de la

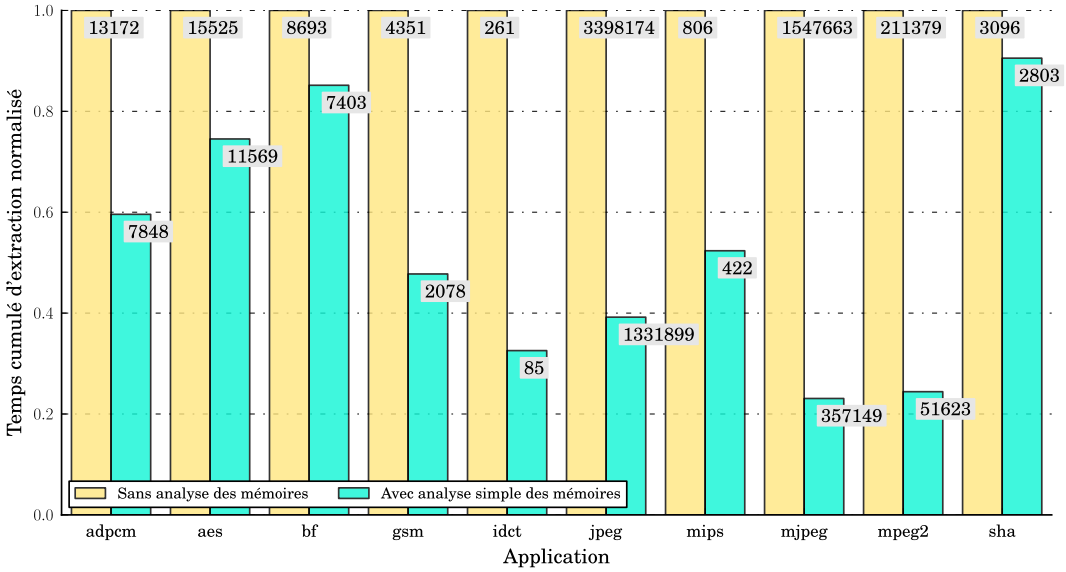


FIGURE 7.9 – Gain de l'analyse mémoire.

durée de vie des mémoires à ceux ayant effectué celle-ci. On recueille les résultats de l'outil de synthèse ISE en terme de quantité de LUT et FF comme précédemment. De manière prévisible, les circuits ayant eu recours à l'analyse sont plus complexes et sont donc plus couteux. Toutefois, le surcout engendré par la sélection des CSM reste inférieur à 5% dans tous les cas. Compte tenu du gain potentiel engendré par cette analyse, on peut considérer son adoption comme bénéfique.

7.6 Synthèse des résultats

Dans ce paragraphe, on fait une synthèse des résultats produits dans ce chapitre. La méthode présentée est intégrée dans un logiciel, greffon d'un logiciel de HLS préexistant (AUGH). Les expériences et mesures sont réalisées sur carte et au sein d'un système complet. Du point de vue des résultats on note en premier lieu que la sélection des points de sauvegarde permet effectivement de réduire la taille du contexte relatif aux registres. Le temps moyen d'extraction est néanmoins dominé par l'extraction du contexte et non la recherche des points de sauvegarde. Pour ce qui est du cout du mécanisme, l'avantage est au groupement CSU, qui est donc sélectionné pour les expériences. Découper les mémoires afin de se débarrasser de la fragmentation associée aux mémoires est peu couteux. De plus, les performances des circuits sont impactées de manière limitée bien que difficilement prévisibles compte tenu du problème des faux chemins. On vérifie que le mécanisme obtenu est effectivement situé correctement par rapport à l'état de l'art, la méthode apportant de véritables arguments en faveur de son adoption. Pour finir, on contrôle la technique d'analyse simple des mémoires : celle-ci présente un ratio bénéfice/cout *a priori* favorable.

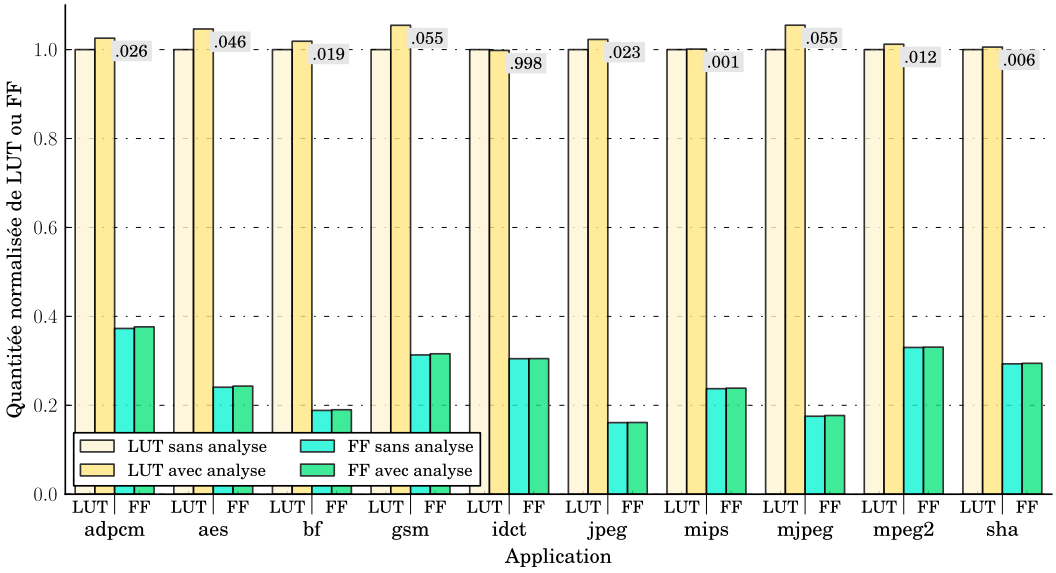


FIGURE 7.10 – Surcouts matériels en LUT et FF liés à l'analyse simple de la durée de vie des mémoires.

Chapitre 8

Conclusion et perspectives

Imaginer et réaliser une architecture de circuit commutable appartient à l'état de l'art depuis les premières publications des années 1990. Dans cette thèse, cette technique avantageuse est remise au gout du jour, améliorée grâce à plusieurs contributions et ses différentes facettes analysées sous un angle nouveau. On présente un flot de conception permettant l'insertion d'une brique de base technologique dans des circuits fonctionnant sur cible reconfigurable.

L'adoption d'un mécanisme de changement de contexte à l'usage de tâches matérielles s'exécutant sur une cible reconfigurable passe par une flexibilité des flots de conception jusqu'ici peu assurée. On propose d'utiliser un outil de synthèse de haut niveau afin d'ajouter la capacité souhaitée dans la description matérielle des circuits. On obtient non seulement l'**automatisation** de la construction de tâches matérielles dotées d'un mécanisme d'extraction de contexte mais aussi la **portabilité** de leur description sur des cibles reconfigurables hétérogènes. Le développeur d'application n'est pas impliqué dans la construction du mécanisme. Afin de maximiser les chances de son adoption, le mécanisme ajouté au circuit doit répondre à des critères de qualité rigoureux. Nous avons identifié en amont de nos travaux une **contrainte** forte, afin de répondre aux besoins des ordonnanceurs existants, assortie de trois objectifs d'optimisation. La réponse d'un circuit à un ordre d'extraction de contexte doit être bornée en temps, et c'est cette contrainte qu'est chargée d'intégrer la première étape du flot de conception : la sélection des points de sauvegarde. Cette étape ainsi que la seconde et dernière de notre flot de conception, qui consiste à additionner le circuit synthétisé d'un mécanisme d'extraction de son contexte, doivent tenir compte du triple objectif d'optimisation. Tout au long du manuscrit, on a étudié la faisabilité et l'impact de l'ajout de différents mécanismes d'extraction sur le circuit, ses performances, sa nature ainsi que son comportement au sein d'un système. La sélection des points de sauvegarde au cœur de la machine d'état permet une préservation du surcout matériel lié à la méthode et une préservation des performances. Le mécanisme adopté, la CSU parallèle assortie des CSM, est aussi construit dans ce but. On propose ensuite deux contributions spécifiques aux mémoires embarquées dans les circuits synthétisés. La découpe des mémoires pour faciliter leur extraction limite l'impact sur le système de la phase de sauvegarde de contexte en réduisant la quantité de données à extraire et donc à faire transiter.

Enfin, l'analyse simple de la durée de vie des mémoires, bien que n'étant pas encore automatisée intégralement, réduit de la même manière la taille du contexte à extraire. Toutes les contributions présentées dans ce manuscrit ont été testées et ont subi mesures et analyses sur deux plateformes matérielles concrètes. La plateforme VALZY, mettant en œuvre des composants complexes et une architecture de système complet, est par ailleurs le support de recherches en cours dans le prolongement de ce travail. À la fin de ces travaux, une démonstration de migration de tâche matérielle sur des cibles hétérogènes (entre des technologies Xilinx et Altera) a été réalisée sur VALZY.

De nombreuses pistes restent ouvertes afin d'améliorer encore les contributions de cette thèse. On a déjà mentionné la mise en application des travaux de ZAOURAR et al. [Zao+12] afin d'ordonnancer la chaîne de sérialisation du contexte propre aux registres à l'aide d'une métrique faisant sens (section 6.2.1, page 62). On a aussi proposé sommairement une méthode d'analyse embarquée des accès aux index des mémoires des circuits afin de réduire la quantité de données à extraire lors d'une extraction de contexte (section 6.4.3, page 76). Cette méthode prometteuse est en cours d'investigation.

Les travaux futurs que l'on a pas encore abordés se retrouvent à chaque étape du flot de conception. Dans la sélection des points de sauvegarde, il est possible d'améliorer l'algorithme d'identification de la matrice $\boldsymbol{A}$ pour éviter certaines situations actuellement résolus en « pire cas ». Toutefois, l'existence du plancher de points de sauvegarde indique que trouver plus rapidement les points de sauvegarde les plus intéressants n'est finalement pas prioritaire. Afin de tirer parti d'une latence de préemption potentiellement laxiste (t_{lat} grand), il serait aussi envisageable de ne pas emprunter le premier point de sauvegarde rencontré mais de continuer le plus longtemps possible l'exécution de la tâche et de n'extraire le contexte qu'au dernier moment. Dans ce cas, il faut embarquer dans le circuit des informations telles que le temps maximum nécessaire pour atteindre un prochain point de sauvegarde depuis un état. Le cout d'une telle option est peu aisé à estimer sans étude préalable.

Réduire l'impact du mécanisme sur les performances du circuit est crucial. Prévoir l'évolution du chemin critique lors de l'insertion d'un point de sauvegarde est une information précieuse et il est possible d'en obtenir une estimation à l'aide de AUGH. Avec cette information, on peut ajouter un critère de sélection des points de sauvegardes en fonction de leur impact sur le chemin critique. D'autre part, la résolution du problème des faux chemins dans AUGH via l'export des chemins vers les outils en aval ouvre des perspectives intéressantes. Il est possible de contraindre le flot de synthèse avec des fréquences de fonctionnement différentes selon que le circuit fonctionne en mode normal ou en mode extraction/récupération de contexte. Réduire la fréquence du mode contexte permet de limiter l'impact de son ajout sur la fréquence de fonctionnement du circuit normal. À l'inverse, les chemins d'extraction de données ne contenant aucune logique combinatoire, il est possible que la fréquence de fonctionnement de cette sous-partie du circuit soit plus élevée que celle du circuit global. Il est dans ce cas possible de faire fonctionner le circuit à une fréquence plus élevée lors de l'extraction du contexte.

Il est enfin nécessaire de se poser la question des communications du circuit

avec son environnement. Plusieurs problèmes se posent. D'une part, il faut prévoir les interfaces capables d'absorber l'extraction d'un contexte pesant jusqu'à 1 Mbit (cas de l'application jpeg) à un débit atteignant aisément le gigabit par seconde (interface de 32 bits de large fonctionnant à au moins 30 MHz). D'autre part, il faut être capable de prendre en charge de manière générique la migration du contexte « indirect » de la tâche matérielle, c'est-à-dire par exemple les données produites par la tâche mais pas encore consommées par le système récepteur ou encore les données en attente d'être consommées par la tâche matérielle préemptée. Ces données ne font pas partie du contexte de la tâche matérielle car elles n'y transitent pas encore ou plus. Elles sont toutefois indispensable au fonctionnement correct d'un système tirant parti de la fonctionnalité de changement de contexte.

AUGH et CP3 sont libres sous licence AGPLv3 et leur sources sont accessibles sur le site internet du projet [Bou16].

Glossaire

ASIC Circuit intégré dédié à une application spécifique. Sigle issu de l'anglais *Application-Specific Integrated Circuit*.

BRAM Bloc de mémoire permettant de lire et d'écrire des valeurs par adresse. Sigle issu de l'anglais *Block of Random Access Memory*.

CLB Élément constitutif d'un réseau logique programmable (FPGA). Il organise et contient des LUT et des FF. Sigle issu de l'anglais *Configurable Logic Block*.

CPU Représente le microprocesseur principal d'un ordinateur. Sigle issu de l'anglais *Central Processing Unit*.

CSP CSP est le sigle de Chaîne de Sérialisation Partielle. Les CSP sont des chaînes d'extraction propres au contexte relatif au registre d'un ou plusieurs points de sauvegardes partageant un contexte identique.

CSU CSU est le sigle de Chaîne de Sérialisation Union. Une CSU est un groupement de toutes les variables appartenant au contexte lié aux registres d'au moins un point de sauvegarde.

DPGA Circuit dérivé du FPGA. Présente la particularité de pouvoir permuter des configurations différentes de la puce dans des temps très courts (de l'ordre du cycle d'horloge). Sigle issu de l'anglais *Dynamically Programmed Gate Arrays*.

DSP Composant permettant d'effectuer des calculs du type traitement du signal (couramment addition, soustraction, multiplication, etc). Sigle issu de l'anglais *Digital Signal Processor*.

FF Registre élémentaire permettant de mémoriser un bit, couramment implémenté sous la forme d'une bascule D. Sigle issu de l'anglais *Flip-Flop*.

FPGA Circuit reconfigurable capable d'adopter le comportement de circuits combinatoires ou séquentiels variés. Sigle issu de l'anglais *Field Programmable Gate Array*.

GPU Représente un composant de calcul dédié au traitement et à l'accélération des graphismes. Sigle issu de l'anglais *Graphical Processing Unit*.

HDL Se dit d'un langage permettant la description de circuits électroniques, souvent numériques. Sigle issu de l'anglais *Hardware Description Language*.

HLS Domaine de la conception de circuits numériques utilisant des langages de haut niveau (tel le C) comme description d'entrée. Sigle issu de l'anglais *High-Level Synthesis*.

HTG Graphe hiérarchique de tâches, servant de représentation interne aux logiciels de CAO. Sigle issu de l'anglais *Hierarchical Task Graph*.

LUT Table de correspondance permettant de générer une valeur booléenne à partir d'une combinaison de valeurs d'entrées. Peut aussi servir de composant de mémoire de petite taille. Sigle issu de l'anglais *Look-Up Table*.

ORGA Circuit dérivé du FPGA. L'état de l'étage reconfigurable est piloté par des photodiodes. Sigle issu de l'anglais *Optically Reconfigurable Gate Array*.

SE Système d'Exploitation. Ensemble de programmes s'éxécutant sur une machine et dont l'objectif est double. D'une part, il présente une abstraction des ressources matérielles du système ainsi que des méthodes pour accéder à ces ressources. D'autre part, il permet le multiplexage de ces ressources entre les utilisateurs du système.

SoC Circuit intégré contenant des composants de natures différentes, qui assemblés forment un système complexe. Par exemple un SoC peut contenire un processeur, de la mémoire ainsi que des périphériques d'entrée/sortie. Sigle issu de l'anglais *System on Chip*.

Bibliographie

[Asa+05] Ghazanfar ASADI et Mehdi B TAHOORI. “Soft error rate estimation and mitigation for SRAM-based FPGAs”. In : *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. ACM. 2005, p. 149–160.

[Asa+09] S. ASANO, T. MARUYAMA et Y. YAMAGUCHI. “Performance comparison of FPGA, GPU and CPU in image processing”. In : *2009 International Conference on Field Programmable Logic and Applications*. 2009, p. 126–131. DOI : 10.1109/FPL.2009.5272532.

[Blo+03a] Brandon BLODGET, Scott MCMILLAN et Patrick LYSAGHT. “A lightweight approach for embedded reconfiguration of FPGAs”. In : *DATE, 2003*. IEEE. 2003, p. 399–400.

[Blo+03b] Brandon BLODGET, Philip JAMES-ROXBY, Eric KELLER, Scott MCMILLAN et al. “A Self-reconfiguring Platform”. In : *Field Programmable Logic and Application : 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003 Proceedings*. Sous la dir. de Peter Y. K. CHEUNG et George A. CONSTANTINIDES. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003, p. 565–574. ISBN : 978-3-540-45234-8. DOI : 10.1007/978-3-540-45234-8_55. URL : http://dx.doi.org/10.1007/978-3-540-45234-8_55.

[Bol+16] Théotime BOLLENGIE, Mohamad NAJEM, Jean-Christophe Le LANN et Loïc LAGADEC. “Zeff : Une plateforme pour l’intégration d’architectures overlay dans le Cloud”. In : *Conférence d’informatique en Parallélisme, Architecture et Système (COMPAS 2016)*. Lorient, France, juil. 2016.

[Bou16] Alban BOURGE. *CP3, AUGH plugin*. 2016. URL : http://tima.imag.fr/sls/research-projects/cp3/.

[Bra+12] Alexander BRANT et Guy GF LEMIEUX. “ZUMA : An open FPGA overlay architecture”. In : *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE. 2012, p. 93–96.

[But+13] G.C. BUTTAZZO, M. BERTOGNA et Gang YAO. “Limited Preemptive Scheduling for Real-Time Systems. A Survey”. In : *Industrial Informatics, IEEE Transactions on* 9.1 (2013), p. 3–15. ISSN : 1551-3203.

[Can+12] Fabio CANCARE, Davide B BARTOLINI, Matteo CARMINATI, Donatella SCIUTO et al. “On the evolution of hardware circuits via reconfigurable architectures”. In : *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 5.4 (2012), p. 22.

[Cha+97] Douglas CHANG et Malgorzata MAREK-SADOWSKA. “Buffer Minimization and Time-multiplexed I/O on Dynamically Reconfigurable FPGAs”. In : *Proceedings of the 1997 ACM Fifth International Symposium on Field-programmable Gate Arrays*. FPGA ’97. Monterey, California, USA : ACM, 1997, p. 142–148. ISBN : 0-89791-801-0. DOI : 10.1145/258305.258331. URL : http://doi.acm.org/10.1145/258305.258331.

[Che+92] D. C. CHEN et J. M. RABAEY. “A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths”. In : *IEEE Journal of Solid-State Circuits* 27.12 (1992), p. 1895–1904. ISSN : 0018-9200. DOI : 10.1109/4.173120.

[Chv79] Vasek CHVATAL. “A greedy heuristic for the set-covering problem”. In : *Mathematics of operations research* 4.3 (1979), p. 233–235.

[Cou+10] Philippe COUSSY et Adam MORAWIEC. *High-level synthesis*. Springer, 2010.

[Dav+02] R. DAVID, D. CHILLET, S. PILLEMENT et O. SENTIEYS. “DART : a dynamically reconfigurable architecture dealing with future mobile telecommunications constr”. In : *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002*. 2002, p. 156 –163. DOI : 10.1109/IPDPS.2002.1016554.

[Dav+07] Nirav DAVE, Kermin FLEMING, Myron KING, Michael PELLAUER et al. “Hardware acceleration of matrix multiplication on a xilinx fpga”. In : *Formal Methods and Models for Codesign, 2007. MEMOCODE 2007. 5th IEEE/ACM International Conference on*. IEEE. 2007, p. 97–100.

[DeH94] Andre DEHON. “DPGA-coupled microprocessors : Commodity ICs for the early 21st century”. In : *FCCM, 1994.* IEEE. 1994, p. 31–39.

[DeH96] André DEHON. “DPGA utilization and application”. In : *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*. ACM. 1996, p. 115–121.

[EA+09] Esam EL-ARABY, Ivan GONZALEZ et Tarek EL-GHAZAWI. “Exploiting partial runtime reconfiguration for high-performance reconfigurable computing”. In : *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 1.4 (2009), p. 21.

[EG+08] Tarek EL-GHAZAWI, Esam EL-ARABY, Miaoqing HUANG, Kris GAJ et al. “The promise of high-performance reconfigurable computing”. In : *Computer* 2 (2008), p. 69–76.

[Egw+13] IfeanyiP. EGWUTUOHA, David LEVY, Bran SELIC et Shiping CHEN. "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems". In : *The Journal of Supercomputing* 65.3 (2013), p. 1302–1326. ISSN : 0920-8542. DOI : 10.1007/s11227-013-0884-0. URL : http://dx.doi.org/10.1007/s11227-013-0884-0.

[Eln+02] Elmootazbellah Nabil ELNOZAHY, Lorenzo ALVISI, Yi-Min WANG et David B JOHNSON. "A survey of rollback-recovery protocols in message-passing systems". In : *ACM Computing Surveys (CSUR)* 34.3 (2002), p. 375–408.

[Fas01] Jean-Philippe FASSINO. "THINK : vers une architecture de systèmes flexibles". Theses. Télécom ParisTech, déc. 2001. URL : https://pastel.archives-ouvertes.fr/tel-00005776.

[Fek+12] Sándor P FEKETE, Tom KAMPHANS, Nils SCHWEER, Christopher TESSARS et al. "Dynamic defragmentation of reconfigurable devices". In : *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 5.2 (2012), p. 8.

[Gar+07] Samuel GARCIA, Jean-Christophe PRÉVOTET et Bertrand GRANADO. "Hardware task context management for fine grained dynamically recon gurable architecture". In : *Workshop on Design and Architectures for Signal and Image Processing*. 2007.

[Gar+09] Samuel GARCIA et Bertrand GRANADO. "OLLAF : a fine grained dynamically reconfigurable architecture for os support". In : *EURASIP journal on embedded systems* 2009 (2009), p. 10.

[Gir+94] Milind GIRKAR et Constantine D POLYCHRONOPOULOS. "The hierarchical task graph as a universal intermediate representation". In : *International Journal of Parallel Programming* 22.5 (1994), p. 519–551.

[Gol+00] S. C. GOLDSTEIN, H. SCHMIT, M. BUDIU, S. CADAMBI et al. "PipeRench : a reconfigurable architecture and compiler". In : *Computer* 33.4 (2000), p. 70–77. ISSN : 0018-9162. DOI : 10.1109/2.839324.

[Gua+08] Nan GUAN, Qingxu DENG, Zonghua GU, Wenyao XU et al. "Schedulability Analysis of Preemptive and Nonpreemptive EDF on Partial Runtime-reconfigurable FPGAs". In : *ACM Trans. Des. Autom. Electron. Syst.* 13.4 (oct. 2008), 56 :1–56 :43. ISSN : 1084-4309. DOI : 10.1145/1391962.1391964.

[Hal10] Tom R HALFHILL. "Tabula's time machine". In : *Microprocessor Report* 131 (2010), p. 0–0.

[Han+11] Simen Gimle HANSEN, Dirk KOCH et Jim TORRESEN. "High speed partial run-time reconfiguration using enhanced ICAP hard macro". In : *IPDPSW, 2011*. IEEE. 2011, p. 174–180.

[Hap+15] Markus HAPPE, Andreas TRABER et Ariane KELLER. "Preemptive Hardware Multitasking in ReconOS". In : *Applied Reconfigurable Computing : 11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings*. Sous la dir. de Kentaro SANO, Dimitrios SOUDRIS, Michael HÜBNER et C. Pedro DINIZ. Cham : Springer International Publishing, 2015, p. 79–90. ISBN : 978-3-319-16214-0. DOI : 10.1007/978-3-319-16214-0_7. URL : http://dx.doi.org/10.1007/978-3-319-16214-0_7.

[Har+09] Yuko HARA, Hiroyuki TOMIYAMA, Shinya HONDA et Hiroaki TAKADA. "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis". In : *Information and Media Technologies* 4.4 (2009), p. 740–752.

[Hau+10] Scott HAUCK et Andre DEHON. *Reconfigurable Computing : The Theory and Practice of FPGA-Based Computation*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2010. ISBN : 0123705223, 9780080556017, 9780123705228.

[Her+07] Martin C HERBORDT, Tom VANCOURT, Yongfeng GU, Bharat SUKHWANI et al. "Achieving high performance with FPGA-based computing". In : *Computer* 40.3 (2007), p. 50.

[Hua+07] Chun-Hsian HUANG, Kai-Jung SHIH, Chao-Sheng LIN, Shih-Shiue CHANG et al. "Dynamically swappable hardware design in partially reconfigurable systems". In : *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*. IEEE. 2007, p. 2742–2745.

[Jon+95] D. JONES et D. M. LEWIS. "A time-multiplexed FPGA architecture for logic emulation". In : *Custom Integrated Circuits Conference, 1995., Proceedings of the IEEE 1995*. 1995, p. 495–498. DOI : 10.1109/CICC.1995.518231.

[Jov+07] Slavisa JOVANOVIC, Camel TANOUGAST et Serge WEBER. "A hardware preemptive multitasking mechanism based on scan-path register structure for FPGA-based reconfigurable systems". In : *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*. IEEE. 2007, p. 358–364.

[Joz+10] K. JOZWIK, H. TOMIYAMA, S. HONDA et H. TAKADA. "A Novel Mechanism for Effective Hardware Task Preemption in Dynamically Reconfigurable Systems". In : *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. 2010, p. 352–355. DOI : 10.1109/FPL.2010.76.

[Joz+12] Krzysztof JOZWIK, Hiroyuki TOMIYAMA, Masato EDAHIRO, Shinya HONDA et al. "Comparison of preemption schemes for partially reconfigurable FPGAs". In : *Embedded Systems Letters, IEEE* 4.2 (2012), p. 45–48.

[Kal+05] Heiko KALTE et Mario PORRMANN. “Context saving and restoring for multitasking in reconfigurable systems”. In : *FPL, 2005*. IEEE. 2005.

[Kal+06] Heiko KALTE et Mario PORRMANN. “REPLICA2Pro : Task relocation by bitstream manipulation in Virtex-II/Pro FPGAs”. In : *Proceedings of the 3rd conference on Computing frontiers*. ACM. 2006, p. 403–412.

[Kin+06] Volodymyr KINDRATENKO et David POINTER. “A case study in porting a production scientific supercomputing application to a reconfigurable computer”. In : *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*. IEEE. 2006, p. 13–22.

[Koc+07] Dirk KOCH, Christian HAUBELT et Jürgen TEICH. “Efficient hardware checkpointing : concepts, overhead analysis, and implementation”. In : *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*. ACM. 2007.

[Koc12] D. KOCH. *Partial Reconfiguration on FPGAs : Architectures, Tools and Applications*. T. 153. Lecture Notes in Electrical Engineering 1. Springer, 2012. ISBN : 9781461412250.

[Lag+01] Löic LAGADEC, Dominique LAVENIER, Erwan FABIANI et Bernard POTTIER. “Placing, routing, and editing virtual FPGAs”. In : *International Conference on Field Programmable Logic and Applications*. Springer Berlin Heidelberg. 2001, p. 357–366.

[Lan+02] Wesley J. LANDAKER, Michael J. WIRTHLIN et Brad L. HUTCHINGS. “Multitasking Hardware on the SLAAC1-V Reconfigurable Computing System”. In : *Field-Programmable Logic and Applications : Reconfigurable Computing Is Going Mainstream : 12th International Conference, FPL 2002 Montpellier, France, September 2–4, 2002 Proceedings*. Sous la dir. de Manfred GLESNER, Peter ZIPF et Michel RENOVELL. Berlin, Heidelberg : Springer Berlin Heidelberg, 2002, p. 806–815. ISBN : 978-3-540-46117-3. DOI : 10.1007/3-540-46117-5_83. URL : http://dx.doi.org/10.1007/3-540-46117-5_83.

[Li+07] Chuanpeng LI, Chen DING et Kai SHEN. “Quantifying the cost of context switch”. In : *ExpCS 2007*. ACM. 2007, p. 2.

[Lyk+15] J.C. LYKE, C.G. CHRISTODOULOU, G.A. VERA et A.H. EDWARDS. “An Introduction to Reconfigurable Systems”. In : *Proceedings of the IEEE* 103.3 (2015), p. 291–317. ISSN : 0018-9219. DOI : 10.1109/JPROC.2015.2397832.

[Lys+05] Roman LYSECKY, Kris MILLER, Frank VAHID et Kees VISSERS. “Firm-core Virtual FPGA for Just-in-Time FPGA Compilation (Abstract Only)”. In : *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*. FPGA '05. Monterey, California, USA : ACM, 2005, p. 271–271. ISBN : 1-59593-029-9. DOI :

10.1145/1046192.1046247. URL : http://doi.acm.org/10.1145/1046192.1046247.

[Lüb+09] Enno LÜBBERS et Marco PLATZNER. "ReconOS : Multithreaded programming for reconfigurable computers". In : *ACM Transactions on Embedded Computing Systems (TECS)* 9.1 (2009), p. 8.

[Mei+03] Bingfeng MEI, Serge VERNALDE, Diederik VERKEST, Hugo DE MAN et al. "ADRES : An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix". In : *Field Programmable Logic and Application : 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003 Proceedings*. Sous la dir. de Peter Y. K. CHEUNG et George A. CONSTANTINIDES. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003, p. 61–70. ISBN : 978-3-540-45234-8. DOI : 10.1007/978-3-540-45234-8_7. URL : http://dx.doi.org/10.1007/978-3-540-45234-8_7.

[Mig+03] J-Y MIGNOLET, Vincent NOLLET, Paul COENE, Diederik VERKEST et al. "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip". In : *DATE, 2003*. IEEE. 2003, p. 986–991.

[Min83] Michel MINOUX. *Programmation mathématique : théorie et algorithmes*. Sous la dir. de Tec & DOC. T. 1. Dunod Paris, 1983.

[Mir+96] E. MIRSKY et A. DEHON. "MATRIX : a reconfigurable computing architecture with configurable instruction distribution and deployable resources". In : *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*. 1996, p. 157–166. DOI : 10.1109/FPGA.1996.564808.

[MV+13] A. MORALES-VILLANUEVA et A. GORDON-ROSS. "On-chip Context Save and Restore of Hardware Tasks on Partially Reconfigurable FPGAs". In : *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. 2013, p. 61–64. DOI : 10.1109/FCCM.2013.13.

[Nak+09] Mao NAKAJIMA et Minoru WATANABE. "A Four-Context Optically Differential Reconfigurable Gate Array". In : *J. Lightwave Technol.* 27.20 (2009), p. 4460–4470. URL : http://jlt.osa.org/abstract.cfm?URI=jlt-27-20-4460.

[Ols+12] Corey B OLSON, Maria KIM, Cooper CLAUSON, Boris KOGON et al. "Hardware acceleration of short read mapping". In : *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE. 2012, p. 161–168.

[Owe+08] John D OWENS, Mike HOUSTON, David LUEBKE, Simon GREEN et al. "GPU computing". In : *Proceedings of the IEEE* 96.5 (2008), p. 879–899.

[Pap+11] Kyprianos PAPADIMITRIOU, Apostolos DOLLAS et Scott HAUCK. "Performance of partial reconfiguration in FPGA systems : A survey and a cost model". In : *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 4.4 (2011), p. 36.

[Par+15] Hyunseok PARK, Shreel VIJAYVARGIYA et André DEHON. "Energy minimization in the time-space continuum". In : *Field Programmable Technology (FPT), 2015 International Conference on*. 2015, p. 64–71. DOI : 10.1109/FPT.2015.7393131.

[PB+14] Adrien PROST-BOUCLE, Olivier MULLER et Frédéric ROUSSEAU. "Fast and standalone Design Space Exploration for High-Level Synthesis under resource constraints". In : *Journal of Systems Architecture* 60 (2014), p. 79–93.

[PB13] Adrien PROST-BOUCLE. *AUGH project*. 2013. URL : http://tima.imag.fr/sls/research-projects/augh/.

[Pha+13] H. M. PHAM, S. PILLEMENT et S. J. PIESTRAK. "Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor". In : *IEEE Transactions on Computers* 62.6 (2013), p. 1179–1192. ISSN : 0018-9340. DOI : 10.1109/TC.2012.55.

[Poc+13] K. POCEK, R. TESSIER et A. DEHON. "Birth and adolescence of reconfigurable computing : a survey of the first 20 years of field-programmable custom computing machines". In : *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. 2013, p. 1–17. DOI : 10.1109/FPGA.2013.6882273.

[Reo+09] M Sonza REORDA, Massimo VIOLANTE, Cristina MEINHARDT et Ricardo REIS. "A low-cost SEE mitigation solution for soft-processors embedded in systems on programmable chips". In : *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design et Automation Association. 2009, p. 352–357.

[Res+05] Javier RESANO, Daniel MOZOS, Diederik VERKEST et Francky CATTHOOR. "A reconfiguration manager for dynamically reconfigurable hardware". In : *IEEE Design & Test* 22.5 (2005), p. 452–460.

[Rup+09] Kyle RUPNOW, Wenyin FU et Katherine COMPTON. "Block, drop or roll (back) : Alternative preemption methods for RH multi-tasking". In : *FCCM'09*. IEEE. 2009, p. 63–70.

[Sca+98] Stephen M SCALERA et Jóse R VAZQUEZ. "The design and implementation of a context switching FPGA". In : *FCCM, 1998*. IEEE. 1998, p. 78–85.

[Sch+11] Andrew G SCHMIDT, Bin HUANG, Ron SASS et Matthew FRENCH. "Checkpoint/restart and beyond : resilient high performance computing with FPGAs". In : *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE. 2011, p. 162–169.

[Sed+06] Pete SEDCOLE, Brandon BLODGET, Tobias BECKER, James ANDERSON et al. "Modular dynamic reconfiguration in Virtex FPGAs". In : *Computers and Digital Techniques, IEE Proceedings*. IET. 2006.

[Set+08] D. SETO et M. WATANABE. "A Dynamic Optically Reconfigurable Gate Array - Perfect Emulation". In : *Quantum Electronics, IEEE Journal of* 44.5 (2008), p. 493–500. ISSN : 0018-9197. DOI : 10.1109/JQE.2008.916705.

[Sil+98] Abraham SILBERSCHATZ, Peter B GALVIN et Greg GAGNE. *Operating system concepts*. 6e éd. T. 4. Addison-Wesley Reading, 1998.

[Sim+00] H. SIMMLER, L. LEVINSON et R. MÄNNER. "Multitasking on FPGA Coprocessors". In : *Field-Programmable Logic and Applications : The Roadmap to Reconfigurable Computing : 10th International Conference, FPL 2000 Villach, Austria, August 27–30, 2000 Proceedings*. Sous la dir. de Reiner W. HARTENSTEIN et Herbert GRÜNBACHER. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000, p. 121–130. ISBN : 978-3-540-44614-9. DOI : 10.1007/3-540-44614-1_13. URL : http://dx.doi.org/10.1007/3-540-44614-1_13.

[Sin+00] H. SINGH, Ming-Hau LEE, Guangming LU, F. J. KURDAHI et al. "MorphoSys : an integrated reconfigurable system for data-parallel and computation-intensive applications". In : *IEEE Transactions on Computers* 49.5 (2000), p. 465–481. ISSN : 0018-9340. DOI : 10.1109/12.859540.

[Tan01] Andrew S. TANENBAUM. *Modern operating systems*. 2nd ed. Prentice Hall, 2001.

[Tau+95] Edward TAU, Derrick CHEN, Ian ESLICK et Jeremy BROWN. "A First Generation DPGA Implementation". In : *In Proceedings of the Third Canadian Workshop on Field-Programmable Devices*. 1995, p. 138–143.

[Tes+15] R. TESSIER, K. POCEK et A. DEHON. "Reconfigurable Computing Architectures". In : *Proceedings of the IEEE* 103.3 (2015), p. 332–354. ISSN : 0018-9219. DOI : 10.1109/JPROC.2014.2386883.

[Tod+05] T. J. TODMAN, G. A. CONSTANTINIDES, S. J. E. WILTON, O. MENCER et al. "Reconfigurable computing : architectures and design methods". In : *IEE Proceedings - Computers and Digital Techniques* 152.2 (2005), p. 193–207. ISSN : 1350-2387. DOI : 10.1049/ip-cdt:20045086.

[Ton+13] Da TONG, Lu SUN, Kiran MATAM et Viktor PRASANNA. "High throughput and programmable online trafficclassifier on FPGA". In : *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2013, p. 255–264.

[Tri+97] Steven TRIMBERGER, Dean CARBERRY, Anders JOHNSON et Jennifer WONG. "A time-multiplexed FPGA". In : *FCCM, 1997*. IEEE. 1997, p. 22–28.

[Ull+04] Michael ULLMANN, Michael HÜBNER, Björn GRIMM et Jürgen BECKER. "An FPGA run-time system for dynamical on-demand reconfiguration". In : *IPDPS, 2004*. IEEE. 2004, p. 135.

[Vah+08] Frank VAHID, Greg STITT et Roman LYSECKY. "Warp processing : Dynamic translation of binaries to FPGA circuits". In : *Computer* 7 (2008), p. 40–46.

[Wat+11] Matthew A. WATKINS et David H. ALBONESI. "ReMAP : A Reconfigurable Architecture for Chip Multiprocessors". In : *IEEE Micro* 31.1 (2011), p. 65–77. ISSN : 0272-1732. DOI : http://doi.ieeecomputersociety.org/10.1109/MM.2011.14.

[Whe+01] Timothy WHEELER, Paul GRAHAM, Brent NELSON et Brad HUTCHINGS. "Using design-level scan to improve FPGA design observability and controllability for functional verification". In : *FPL, 2001*. Springer. 2001, p. 483–492.

[Wic+16] Arief WICAKSANA, Alban BOURGE, Olivier MULLER et Frédéric ROUSSEAU. "Demonstration of a Context-Switch Method for Heterogeneous Reconfigurable Systems". In : *2016 26th International Conference on Field Programmable Logic and Applications (FPL), Demo Night*. IEEE. 2016.

[Xil10] XILINX. *Partial Reconfiguration User Guide*. UG702. 2010.

[Xil14] XILINX. *7 Series FPGAs Configurable Logic Block*. UG474. 2014.

[Zao+12] Lilia ZAOURAR, Yann KIEFFER et Chouki AKTOUF. "A graph-based approach to optimal scan chain stitching using RTL design descriptions". In : *VLSI Design* 2012 (2012), p. 3.

[Zha+15a] Chen ZHANG, Peng LI, Guangyu SUN, Yijin GUAN et al. "Optimizing fpga-based accelerator design for deep convolutional neural networks". In : *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2015, p. 161–170.

[Zha+15b] Hongyan ZHANG, M. A. KOCHTE, E. SCHNEIDER, L. BAUER et al. "STRAP : Stress-aware placement for aging mitigation in runtime reconfigurable architectures". In : *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*. 2015, p. 38–45. DOI : 10.1109/ICCAD.2015.7372547.

Publications et communications de l'auteur

Alban BOURGE, Alexandre GHITI, Olivier MULLER et Frédéric ROUSSEAU. "Méthode de sélection de checkpoint matériel avec outil de synthèse de haut niveau". In : *Journées Nationales du Réseau Doctoral en Microélectronique (JNRDM'14)*. Lille, France, mai 2014, p. 4. URL : https://hal.archives-ouvertes.fr/hal-01089685.

Alban BOURGE, Olivier MULLER et Frederic ROUSSEAU. "A Novel Method for Enabling FPGA Context-Switch (Abstract Only)". In : *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, CA, USA : ACM, fév. 2015, p. 261–261. DOI : 10.1145/2684746.2689096. URL : https://hal.archives-ouvertes.fr/hal-01353496.

Alban BOURGE, Olivier MULLER et Frédéric ROUSSEAU. "Automatic High-Level Hardware Checkpoint Selection for Reconfigurable Systems". In : *Field-Programmable Custom Computing Machines (FCCM'15)*. Vancouver, Canada, mai 2015. URL : https://hal.archives-ouvertes.fr/hal-01164923.

Alban BOURGE, Olivier MULLER et Frédéric ROUSSEAU. "Flot de conception automatique pour circuits commutables". In : *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS 2016)*. Lorient, France, juil. 2016. URL : https://hal.archives-ouvertes.fr/hal-01353512.

Alban BOURGE, Olivier MULLER et Frédéric ROUSSEAU. "Generating Efficient Context-Switch Capable Circuits Through Autonomous Design Flow". In : *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 0.0 (2016). Accepté pour publication, à paraître, p. 1–24. URL : https://hal.archives-ouvertes.fr/hal-01367798.

Alban BOURGE, Olivier MULLER et Frederic ROUSSEAU. "La synthèse de haut niveau au service du changement de contexte matériel." In : *Colloque National GDR SoC-SiP*. Nantes, France, 2016. URL : https://hal.archives-ouvertes.fr/hal-01353497.

Tristan GROLEAT, Matthieu ARZEL, Sandrine VATON, Alban BOURGE et al. "Flexible, extensible, open-source and affordable FPGA-based traffic generator". In : *HPDC 2013 : 22nd International ACM Symposium on High Performance Parallel and Distributed Computing*. New-York, United States, juin 2013. URL : https://hal.archives-ouvertes.fr/hal-00859291.

Annexes

Annexe A

Découpage des mémoires

		Extraction (cycles)		Fragmentation		% de frag.	
	taille (bit)	32	64	32	64	32	64
adpcm	12608	394	198	0	64	0%	1%
aes	20608	644	322	0	0	0%	0%
blowfish	34112	1066	539	0	384	0%	1%
gsm	3408	108	56	48	176	1%	5%
idct	2560	80	40	0	0	0%	0%
jpeg	371177	11710	5858	3543	3735	1%	1%
mips	3072	96	48	0	0	0%	0%
decoder	291600	9114	4559	48	176	0%	0%
mpeg2	16768	524	262	0	0	0%	0%
sha	3232	101	51	0	32	0%	1%
Moyenne						0%	1%

TABLE A.1 – Fragmentation du contexte relatif aux mémoires pour $L_m = \{32, 64\}$ avec mécanisme de découpe des mémoires.

Annexe B

Résultats en surface des circuits obtenus

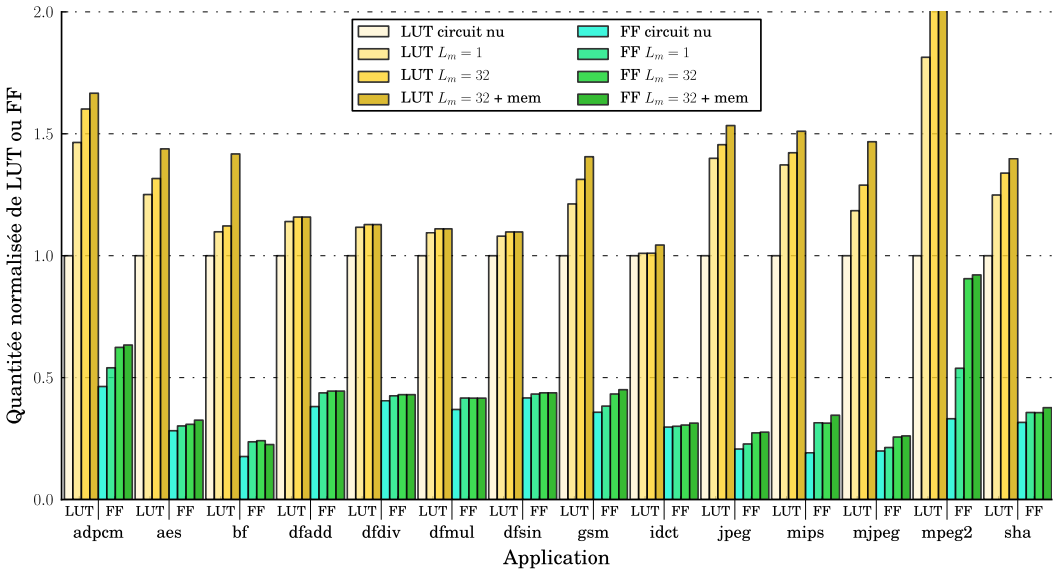


FIGURE B.1 – Cout en surface d'un mécanisme CSP obtenu de manière incrémentale.

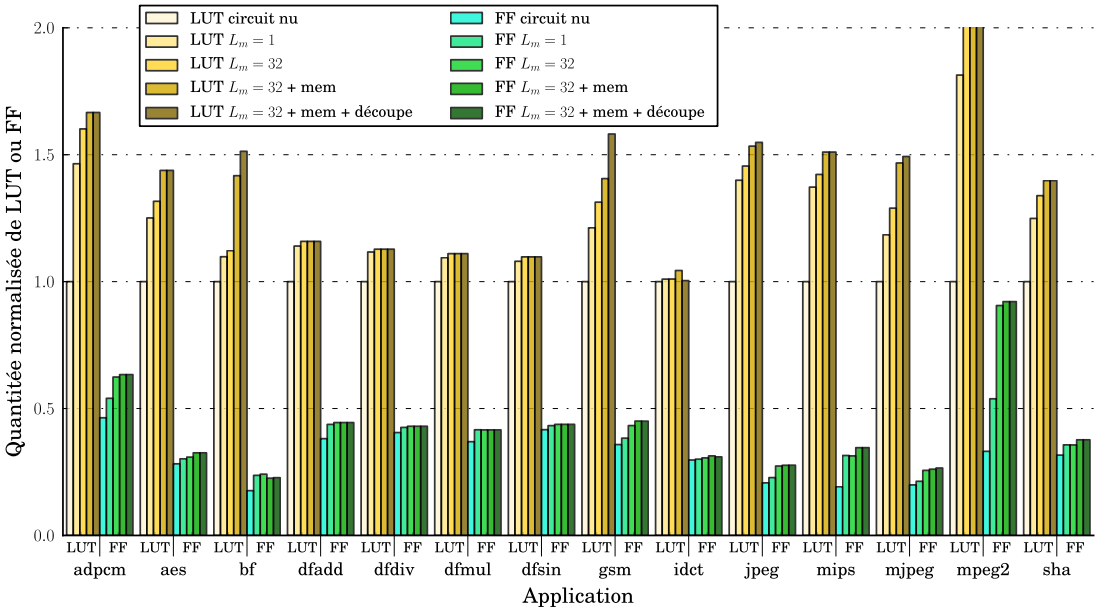


FIGURE B.2 – Cout en surface d'un mécanisme CSP avec découpe des mémoires obtenu de manière incrémentale.

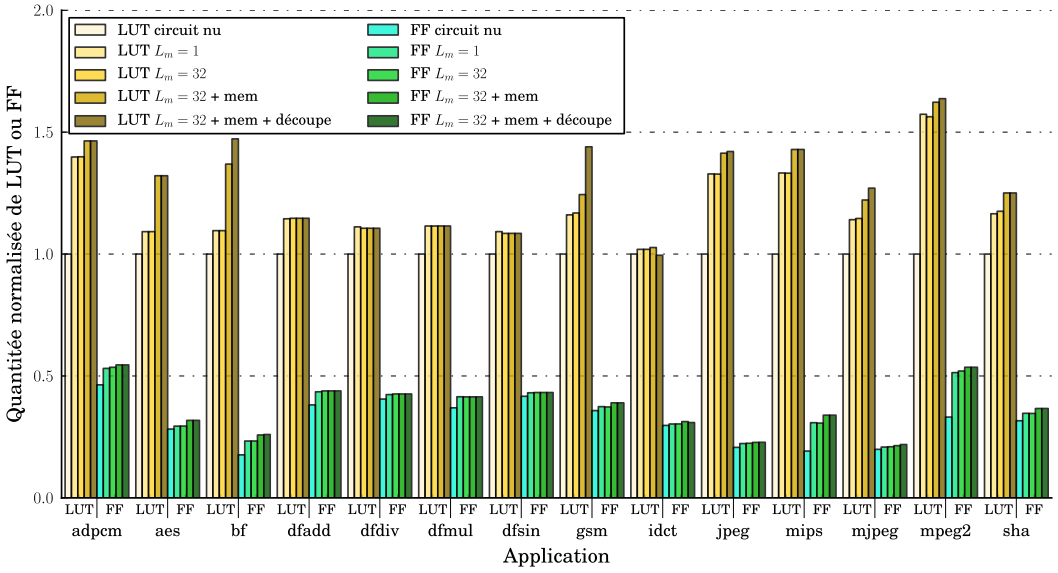


FIGURE B.3 – Cout en surface d'un mécanisme CSU avec découpe des mémoires obtenu de manière incrémentale.

Annexe C

Fréquence des circuits obtenus

	Chem. crit. (ns)		Fréquence max. (MHz)		
	nu	CSU	nu	CSU	diff.
adpcm	13.8	16.8	72.6	59.6	−18%
aes	8.1	7.4	123.2	136.0	10%
bf	6.2	6.2	160.8	160.5	0%
dfadd	7.2	7.7	139.3	130.5	−6%
dfdiv	63.2	62.0	15.8	16.1	2%
dfmul	6.7	6.8	148.5	148.1	0%
dfsin	86.6	179.3	11.5	5.6	−52%
gsm	16.0	19.7	62.6	50.7	−19%
idct	16.2	16.5	61.8	60.6	−2%
jpeg	48.8	45.5	20.5	22.0	7%
mips	5.9	7.6	170.1	132.2	−22%
mjpeg	55.5	48.6	18.0	20.6	14%
mpeg2	13.7	14.7	73.2	67.9	−7%
sha	9.3	9.5	107.0	105.0	−2%

TABLE C.1 – Chemin critique et fréquence maximale obtenues avec et sans CP3, sans option limitant le partage d'opérateur.

Notes

Résumé

Les accélérateurs matériels occupent un rôle déterminant dans l'informatique actuelle. Leur mission est d'assurer des calculs spécifiques trop complexes pour les systèmes génériques basés sur des processeurs. Une famille de composants électroniques dits « reconfigurables », sont depuis des années considérés comme des candidats idéaux pour assurer l'accélération matérielle dans de nombreux cas : production à faible volume, besoin de mise à jour régulière, besoins d'utilisation flexible etc. En pratique, ils ne sont pas utilisés à la hauteur des gains qu'ils pourraient apporter. Afin de faciliter leur adoption on cherche à rendre l'utilisation d'une telle technologie plus flexible. Dans cette thèse, on propose donc d'étudier et d'améliorer la capacité des puces reconfigurables à être partagées. Pour partager une ressource reconfigurable, il faut prévoir la commutation des tâches s'y déroulant. Cette technique, la commutation de tâches matérielles sur cible reconfigurable, n'est pas nouvelle et appartient à l'état de l'art. On propose dans ces travaux d'utiliser cette technique avantageuse en conjonction avec un flot de conception actuel dit de synthèse de haut niveau.

Grâce au flot de synthèse de haut niveau, on peut automatiser la génération de circuits commutables portables car non spécifiques à une architecture de puce reconfigurable. Deux propositions viennent compléter la méthode. Celle-ci visent à tirer parti du niveau de manipulation des circuits afin d'améliorer les performances d'un système utilisant des tâches commutables. Dans un premier temps, on sélectionne un ensemble de points de sauvegarde lors desquels la commutation est autorisée. On additionne ensuite un mécanisme d'extraction à la description matérielle de la tâche. Grâce à ces deux contributions ainsi que leur utilisation à haut niveau, on parvient à automatiser la génération de circuits flexibles et ayant un surcout limité compte tenu des caractéristiques additionnelles obtenues. Un démonstrateur utilisant plusieurs technologies de FPGA est présenté. Sa mise en œuvre permet de vérifier la fonctionnalité voulue ainsi que les mesures et caractérisations in situ.

Mots-clés : FPGA, changement de contexte matériel, accélérateur matériel, HLS

Abstract

Current informatics greatly rely on hardware accelerators. Their mission is to accelerate domain-specific computation often too complex for a general purpose processor. A certain family of electronic devices, as known as reconfigurables, has been considered for accelerating computation in some cases : low volume production, need for recurrent updates, flexible use cases etc. However, they are still not being used enough compared with the benefits they offer. In order to facilitate a wider adoption, we propose to give more flexibility when using this kind of technology. This thesis presents and studies the capacity of reconfigurable resources to be shared. In order to do that, one should be able to switch between tasks executing on the resource. This technique is called context-switch, and has been in the literature for quite some time. In this work we propose to use a high-level synthesis flow in conjunction with this scheme.

Thanks to the high-level synthesis flow, context-switchable circuit construction is automated. The produced circuits are portable because generically built. In addition, we add two steps in the high-level flow in order to give better characteristics to the final circuits. One checkpoint selection limits the states where context-switch is allowed. Moreover, an embedded mechanism is built. Thanks to these two steps, the produced circuits overheads and performances are better than circuits with naive mechanisms. A demonstration platform using FPGAs from different vendors is presented. It allows to test functionally and qualitatively the produced circuit used within a complex system.

Keywords : FPGA, hardware context-switch, hardware accelerator, HLS

ISBN : 978-2-11-129219-2