Conception d'une matrice reconfigurable pour coprocesseur fortement couplé

Nicolas Brunie, Florent de Dinechin, Benoît de Dinechin

ComPAS'2013 : RenPar'21/ SympA'15 / CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Conception d'une matrice reconfigurable pour coprocesseur fortement couplé

Nicolas Brunie, Florent de Dinechin, Benoît de Dinechin

-

Résumé
Cet article étudie la conception d'un opérateur reconfigurable à grain moyen fortement couplé, intégré dans un *System on Chip* (Soc) haute performance et faible consommation. Nous présentons un environnement logiciel en cours de développement destiné à l'exploration architecturale d'une telle solution. L'architecture reconfigurable, très paramétrique, se compose d'une matrice de cellules à grain moyen plongée dans un réseau configurable orienté flot de donnée. Elle est associée à un compilateur qui génère la configuration à partir d'un programme en syntaxe C ou VHDL. Le fonctionnement de cet environnement est illustré sur 3 exemples : l'encryption AES, le corps de boucle d'une fonction de hachage (SHA-1) et un filtre à réponse impulsionnelle finie (FIR).

1. Introduction

On attend aujourd'hui des processeurs faible consommation pour systèmes embarqués qu'ils fournissent des capacités de calcul phénoménales dans des domaines très variés, allant par exemple des applications utlisant intensivement la virgule flottante à la cryptographie sur les corps de Galois. Les niveaux de performance attendus sont inaccessibles sans matériel spécifique.
Pour les application qui le justifient par leur base d'utilisateurs ou leur caractère incontournable, une solution est d'ajouter des instructions matérielles spécifiques. C'est le choix d'Intel pour l'algorithme de cryptographie AES [6]. Cette solution combine performance et efficacité énergétique, mais manque de flexibilité. Si beaucoup d'applications peuvent justifier une telle accélération, le coût en silicium peut devenir très important.
Une autre solution est d'embarquer un cœur reconfigurable similaire à ceux disponible dans les FPGAs actuels (Field Programmable Gate Array). Un tel cœur, fortement couplé avec le processeur, est conçu pour étendre le jeu d'instructions du processeur d'une façon reconfigurable, spécifique à chaque application. Idéalement, cette solution peut fournir des performances proche d'une implémentation purement matérielle, avec une flexibilité proche du logiciel.
Cependant, Kuon et Rose ont montré qu'une fonction donnée implémentée dans une puce FPGA classique nécessite environ 40 fois plus de silicium que ce dont aurait besoin une implémentation purement matérielle [11]. Si moins de 40 applications distinctes ont besoin d'être accélérées de cette manière, cette solution se révèle plus chère qu'un simple multiplexeur avec des blocs matériels spécifiques, allumés ou éteints à l'exécution.

Mais d'autres facteurs entrent en jeu, comme l'adaptabilité aux nouveaux standards, les coûts de développement d'un ASIC par rapport à l'utilisation d'un FPGA, ou la célérité de mise sur le marché *(time to market)*. De plus, le contexte et les besoins d'un processeur embarqué sont différents de ceux pour lesquels les FPGA classiques furent conçus : la conception d'un opérateur reconfigurable pour ces besoins pourrait donc mener à une architecture plus simple et plus efficace.

Le but de la recherche présentée dans cet article est donc de répondre à la question suivante : dans le contexte d'un accélérateur fortement couplé pour un processeur embarqué, peut on réduire le facteur de surface de 40 à une valeur plus acceptable ?

Pour s'attaquer à cette question nous avons besoin d'explorer les possibilités architecturales offertes par le contexte d'un processeur embarqué. Cet article présente RKC pour *Reconfigurable Kernel Compiler*, un ensemble d'outils intégrés dont l'objectif est de permettre une telle exploration architecturale.

Cet environnement logiciel prend en entrée une description paramétrique d'un opérateur reconfigurable, incluant la description de la cellule de base, celle du réseau d'interconnexion reconfigurable. La figure 1 donne un exemple d'un fichier YAML de configuration de cette matrice reconfigurable paramétrique, que nous appelons *Deeply Integrated Reconfigurable Fabric* (DIRF).

À partir de cette description, l'environnement RKC génère
– la descripion vhdl synthétisable de l'accélérateur reconfigurable,
– un compilateur pour cette architecture, et
– un simulateur logiciel.

Le compilateur généré accepte un sous-ensembe du langage C mais aussi du langage VHDL (orienté dataflow pour le C et sans *process* pour le VHDL). Il effectue une synthèse logique, le *technology mapping* sur les cellules de bases, ainsi que le placement et le routage sur l'architecture DIRF.

Le fonctionnement de RKC est démontré sur 3 exemples représentatifs : l'encryption AES, la fonction de hachage SHA1 et les filtres FIR.

1.1. État de l'art

Des projets comme DREAM/PicoGA [14] ou PipeRench [5] ont déjà étudié des architectures à grain moyen orientées flot de données. L'originalité de notre approche est de chercher à concevoir un environnement de développement assez flexible pour permettre une exploration architecturale.

Parmi les tâches intensives en calcul que les processeurs embarqués actuels ont besoin de supporter, deux classes d'applications sont spécifiquement intéressantes pour ce travail :
– l'encodage audio et vidéo, à cause de la variété des algorithmes utilisés, du besoin de calcul à grain fin, et du parallélisme intrinsèque.
– les algorithmes cryptographiques (encryption, décryption et authentification), à cause de leur variété algorithmique également, et de leurs besoins de mélanges de bits ou d'arithmétiques non standard.

Beaucoup de solutions reconfigurables ont été proposées. Les solutions basées sur les FPGAs classiques utilisent la reconfigurabilité à grain fin (au niveau bit), possiblement renforcée par des blocs arithmétiques dédiés au traitement du signal (multiplieurs ...). Par ailleurs, des approches à grain moyen [14] voire à gros grain [19, 1, 23] ont aussi été proposées. On parle ici de la granularité des données : comme il sera détaillé dans la section 2, augmenter la granularité allège le coût de la configuration, au détriment de la flexibilité.

```
interfaces:
    inputs:
        - size: 32
          number: 4
    outputs:
        - size: 32
          number: 4
cellArray:
    rowNum: 5
    cellByRow: 64
    colByCell: 3
    cells:
        grain: 4
        inputs: 3
        outputs: 3
network:
    timingModel:
        node: 10
        jump: "lambda size: size * 0.1"
    type: benes # standard | benes
    colNumByCell: 3
    block_size: 4
    subtype: complete
```

FIGURE 1 – fichier YAML de configuration de la matrice

Le projet AMBER [16] a proposé une unité reconfigurable fortement couplée. Toutefois il se concentre sur l'extension du jeu d'instructions, plutôt qu'explorer les choix architecturaux pour l'unité fonctionnelle reconfigurable.
Le projet TCE (anciennement connu sous le nom MOVE) se concentre sur le *codesign* matériel/-logiciel [10]. Il fournit une architecture largement configurable avec un jeu d'instructions extensible, supportée par un compilateur automatisé basé sur LLVM. Cependant il s'agit d'une extension statique du jeu d'instructions : il n'est pas possible de le reconfigurer à l'exécution comme dans notre approche ou celle de DREAM.

1.2. Contexte : le processeur MPPA de Kalray

MPPA® (Multi Purpose Processor Array) est un processeur *many-core* intégré conçu par Kalray pour les systèmes embarqués, avec une attention particulière portée sur les performances et la faible consommation.
La première version de ce processeur, le MPPA256, offre 256 cœurs Kalray-1 dédiés aux calculs, plus quelques cœurs supplémentaires pour le contrôle et la gestion des données. Chaque cœur K1 implémente une architecture Very Large Instruction Word (VLIW) 5 voies avec une unité arithmétique et logique (ALU) 64 bits. Celle-ci offre des capacités étendues de manipulation de bits, parmi lesquelles une unité de *Bit Matrix Multiply* (BMM) et une unité logique *bitwise* arbitraire (BWLU) à 4 entrées et 2 sorties.
Les cœurs K1 sont regroupés dans des clusters de 4, partageant un accès à leur file de registres respectives à travers un mécanisme d'écriture distante, et un système d'évènements pour la synchronisation.

2. Choix architecturaux pour l'intégration au MPPA d'une matrice reconfigurable

Cet article s'inscrit dans un travail plus général dont la finalité est d'étudier l'utilité d'intégrer un coprocesseur reconfigurable fortement couplé dans le MPPA®. Ce coprocesseur complémente un cluster de 4 processeurs. Il apparaît comme une unité fonctionnelle à chaque cœur, qui peut l'accéder de manière indépendante ou synchronisée. Les cœurs sont la seule manière d'accéder au coprocesseur, contrairement à DREAM [14], qui inclut un générateur d'adresses mémoires dans le coprocesseur pour un accès direct à la mémoire, ou contrairement à [3] qui inclut un contrôle reconfigurable basé sur un réseau de Petri.
Nous présentons maintenant quelque choix importants dans ce contexte, et plus spécifiquement dictés par le besoin de :
– réduire le coût de configuration (nombre de bits de configuration et coût de la mémoire correspondante),
– permettre à la matrice reconfigurable de fonctionner à la même fréquence que les cœurs l'entourant.
– simplifier la compilation pour cet opérateur, le sujet de la section 3.

2.1. Granularité moyenne

La matrice reconfigurable et le réseau de routage opèrent sur une granularité de 4 à 8 bits. Le principal avantage de cette granularité moyenne est de coûter moins de bits de configuration pour le réseau d'interconnexion : les données routées sont des mots de 4 bits. Le principal inconvénient est qu'augmenter la granularité réduit la flexibilité, et augmente le coût des *look-up tables* (LUT) correspondantes. La LUT est le principal élément des cellules de bases d'une matrice reconfigurable de type FPGA. Nous n'avons pas encore exploré la modification de cette granularité dans notre environnement RKC : dans tout l'article, cette granularité est fixée à 4.

Cependant, RKC supporte la modification paramétrique de la granularité, ce qui permettra dans un avenir proche d'explorer quantitativement l'impact de la granularité sur la surface de l'architecture.

2.2. Aucun contrôle

Contrairement aux solutions présentées dans l'état de l'art, nous avons choisi de laisser la totalité du contrôle aux cœurs entourant le coprocesseur. La matrice reconfigurable n'inclut donc aucun automate de contrôle, et doit se limiter aux calculs pipelinés. Cette solution a déjà été en partie envisagée par [13]. Cela simplifie grandement le réseau d'interconnexion qui peut devenir unidirectionnel : du haut vers le bas, des entrées vers les sorties. Ceci réduit les coûts, en particulier le nombre de bits de configuration. Nous avons aussi choisi de ne pas implémenter de connexion horizontale dans nos étages de pipeline, à l'exception d'une chaîne de retenue généralisée décrite plus bas. Pour résumer, la matrice reconfigurable est optimisée pour des opérateurs à très grande profondeur de pipeline (de l'ordre d'une dizaine de cycle), qui seront déployés sous la forme d'un étage de pipeline par ligne de cellules de base dans la matrice reconfigurable.

2.3. Architecture déclenchée par le transport (*Transport-triggered architecture*, TTA)

Le DIRF est une architecture dirigé par le transport : présenter une nouvelle donnée à l'entrée du pipeline déclenche l'avancement d'un cycle de chaque étage. Quand autant de données que le nombre d'étages ont été insérées, le premier résultat peut être récupéré, et ainsi de suite. Ainsi, tous les déplacements de données entre le DIRF et un processeur sont explicites.
Cette solution est souvent envisagée pour les opérateurs reconfigurables ou les extensions de jeu d'instructions (TCE) [10], parce qu'elle ne donne à voir qu'une simple instruction MOVE au processeur en lui cachant toute la complexité interne de l'opérateur. Elle peut donc être simplément intégrée dans n'importe quel processus de compilation ou de simulation sans tenir compte directement du comportement de l'instruction.

2.4. Pas de routage horizontal, mais une chaîne de retenue généralisée

Pour s'assurer qu'un simple additionneur linéaire n'a pas besoin d'un nombre linéaire de niveaux de cellules, une chaîne de retenue rapide horizontale, inspirée des FPGA actuels, est implémentée dans chaque ligne. Cette chaîne est unidirectionnelle et constitue l'unique connexion horizontale possible. À part la propagation de retenues, elle supporte d'autres fonctionnalités comme la réduction et la diffusion [7].

3. L'environnement logiciel RKC

Des projets comparables [13] demandent au programmeur de décomposer manuellement les noyaux de calculs en élements basiques configurables, et de se charger du placement et du routage. Nous pensons qu'un opérateur reconfigurable devrait être directement disponible dans le flot de compilation classique du MPPA. Pour ce faire, le but de notre Reconfigurable Kernel Compiler (RKC) est de permettre aux programmeurs d'utiliser une syntaxe proche du C pour programmer le DIRF. De tels programmes devront toutefois être limités à une description *dataflow*, ce qui se traduit dans le sous-ensemble du langage C accepté.
Une interface VHDL (restreinte de la même manière à une description structurellement *dataflow*) est aussi utile pour faciliter le portage d'architectures exsistantes écrites en VHDL vers des configurations pour le DIRF, puisque ce dernier se rapproche par certains aspects d'un opérateur matériel. Cela nous permet aussi, dans cet article, de comparer la configuration DIRF et la syn-

thèse ASIC obtenues à partir du même VHDL.
Ce processus de compilation est intermédiaire entre de la compilation logicielle et la synthèse matérielle. D'un côté, il n'est pas simple de recibler un compilateur standard (gcc, llvm, ...). D'un autre côté , beaucoup de solutions C vers VHDL sont destinées aux FPGAs, comme par exemple ROCCC [22], et des outils FPGA pour l'aspect back-end sont aussi disponibles, le plus connu étant sans doute VPR [12]. Cependant, ces outils n'ont pas été conçus pour notre matrice reconfigurable unidirectionnelle, orientée dataflow et ne sont pas donc réellement adaptés. Par exemple, le difficile problème de placement en 2 dimensions peut être remplacé par deux problèmes en une dimension beaucoup plus simples : commencer par construire les lignes de cellules, puis ensuite placer les cellules à l'intérieur de ces lignes. Chaque phase peut même être exprimée comme un problème de programmation linéaire en nombre entiers (et même variables booléenes, mais les détails dépassent le champ de cet article). C'est pour ces raisons que nous avons choisi d'implémenter RKC en grande partie en partant de zéro.
La figure 2 illustre le processus de compilation de RKC :
– Un graphe de flot de donnée (DFG pour *data-flow graph*) est construit à partir du code source.
– Un mélange de l'algorithme *flow-map* [2] et de reconnaissance de modèle est utilisé pour décomposer le graphe initial en sous-nœuds correspondants aux cellules de bases matérielles.
– Après placement et routage, cette décomposition ainsi que la configuration du réseau d'interconnexion sont traduits en un fichier binaire *(bitstream)* de configuration.

À moyen terme, RKC sera utilisé comme compilateur de production pour le DIRF. Cependant, à ce stade du projet, son principal intérêt est d'être paramétrisable à de multiples niveaux (indiqués par les encadrés grisés de la figure 2 qui représentent les différents fichiers de configuration utilisés lors de la compilation).

1. Un niveau configure l'exécution de RKC (choix de l'algorithme de placement et routage, limitation du contexte de recherche ...).
2. Les *decomposition rules* configurent comment RKC divise le DFG en ensembles de cellules physiques (elles guident la reconnaissance de modèle d'opération).
3. les *array-level rules* configurent les dimensions de la matrice reconfigurable (nombre de lignes, architecture du réseau d'interconnexion ...).
4. le *cell-level rules* configurent la génération du *bitstream* de configuration, en reliant la configuration des cellules de bases à la valeur de leur registre de configuration physique.

Ces fichiers de configuration sont aussi utilisés par le génrateur VHDL qui produit la description VHDL du DIRF.
Une fois optimisé correctement, la paramétrisation variable du DIRF permettra facilement d'explorer de multiples solutions architecturales. Pour une application donnée, nous pourrons isoler les paramètres de configuration du DIRF qui donnent le meilleur compromis coût/performance. Des exemples seront données dans la section 5.
Cependant, un des principaux défis, pour un logiciel si peu mature, est de déterminer dans quelle mesure nous pouvons nous fier aux résultats de synthèses rendus par le compilateur (aire utilisée, nombre de niveaux logiques, ...) : sont-elles des limites dures ou juste le résultat d'un manque d'optimisation ?

4. La matrice reconfigurable en détail

La figure 3 réprésente l'architecture de la matrice reconfigurable (RA pour *reconfigurable array*). Les entrées se situent en haut et les sorties en bas. La chemin de donnée est orienté du haut vers le bas, avec des lignes de cellules logiques reconfigurables (RLC).

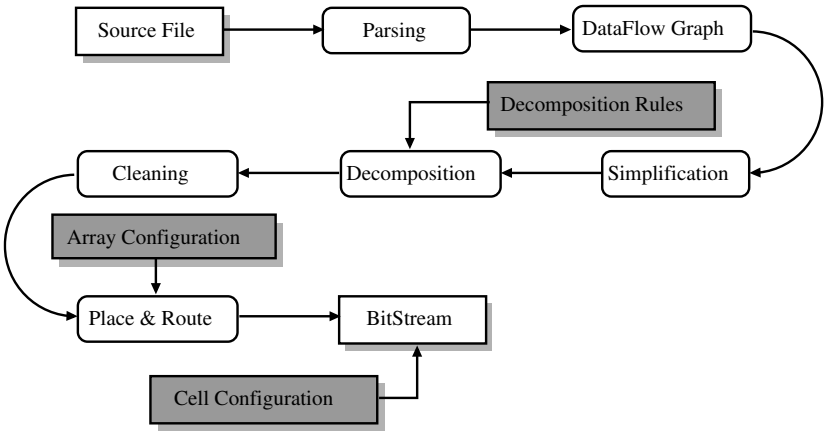

FIGURE 2 – Étape de compilation de RKC

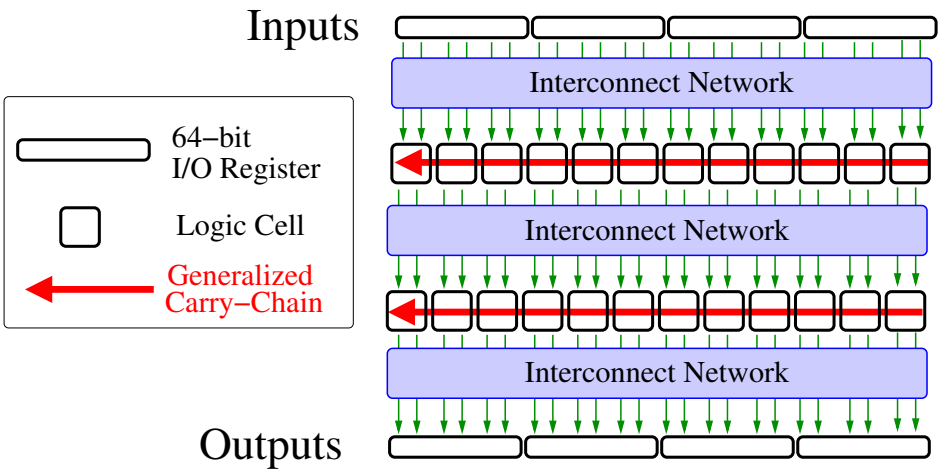

FIGURE 3 – Architecture de la matrice reconfigurable

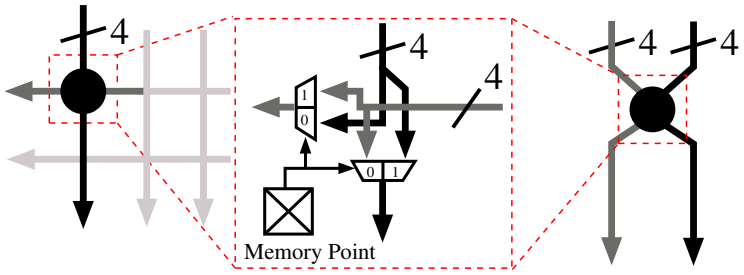

FIGURE 4 – Jonction du réseau d'interconnexion

Chaque cellule sur une ligne ne peut communiquer avec une cellule de la même ligne que via la chaîne de retenue rapide généralisée.
Nous nous attelons maintenant à décrire le réseau d'interconnexion qui relie une ligne de cellule à la suivante comme un des exemples d'exploration architecturale apportés par l'ensemblre DIRF/RKC.

4.1. Réseau d'interconnexion

Deux sortes de réseaux d'interconnexion ont été étudiées : une inspirée par les crossbars des FPGAs classiques et un réseau de permutation.
Actuellement, les deux sont basés sur une jonction élémentaire montrée par la figure 4 (appelée β-élément dans la littérature). Nous pouvons d'ores et déjà noter que ce choix, qui réduit le coût de configuration à un bit par jonction, ne permettra pas la duplication d'un signal. Nous lèverons cette restriction ultérieurement. Par ailleurs, les signaux peuvent être dédoublés en utilisant les sorties multiples des cellules de bases, mais cela augmente la pression locale sur le routage.

Interconnexion de type crossbar

Cette approche est décrite par la figure 5. Elle est basée sur des lignes unidirectionnelles (vers la droite ou vers la gauche). Chaque ligne est définie dans le fichier de configuration par des paramètres *step* et *offset*. Le premier inidique le nombre de colonnes traversées sans interaction entre deux jonctions, et le second indique l'indice de la colonne de départ pour la ligne. En fait, les lignes avec un *step* de k, sont présentes par groupe de k. Par exemple, dans la figure 5, la première ligne a un offet=0 et step=1, les deux lignes suivantes ont step=2.
Dans notre environnement d'exploration, cette structure est complètement paramétrisable. Un choix naturel (par exemple pour l'implémentation dynamique de décalage de bit) est d'avoir des steps qui sont des puissances de 2 croissantes, mais nous conservons la possibilité de les complémenter avec d'autres types de lignes pour faciliter le routage. Chaque ligne a un nombre de bits de configuration égal aux nombres de colonnes croisées. Notre but est d'obtenir un nombre minimal de ligne tout en étant capable de configurer le réseau d'interconnexion de manière à faire fonctionner les applications visées.

Interconnexion de type réseau de permutation

La famille des réseaux de permutations de type Benes [17] ont une profondeur de $\log(n)$, et peuvent être configurés avec $n \times \log(n)$ bits, ce qui est le nombre optimal pour permettre une permutation arbitraire. L'exemple des réseaux de Benes inclut les réseaux *butterfly*, *shuffle-base* ou *omega-flip* [9]. Dans notre cas, nous utilisons un réseau de Benes à n entrées/sorties où n

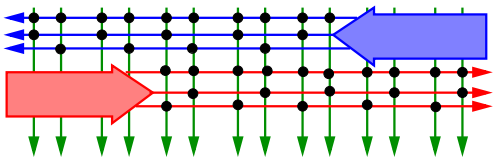

FIGURE 5 – Un exemple de réseau en crossbar

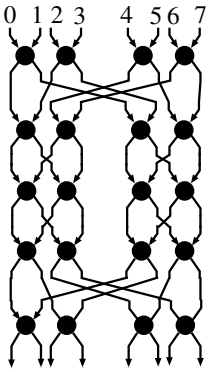

FIGURE 6 – Un exemple de réseau de Benes

n'est pas nécessairement une puissance de 2 [17].

4.2. Cellule de base

Chaque cellule a un nombre paramétrisable d'entrées/sorties à grain moyen. Dans la suite, nous utilisons par exemple 3 entrées (nommées A, B et C) et 2 sorties de 4 bits.
RKC nous permet d'explorer plusieurs variations de cette cellule. Le compromis typique est qu'une cellule plus efficace aura besoin de plus de bits de configurations et plus de silicium, mais une application complète en nécessitera un nombre plus faible. Ces variations peuvent être quantitative (combien d'entrées et de sorties) mais aussi qualitative (devrions-nous intégrer un multiplieur 4 bits à la cellule ?).

4.2.1. Capacité de la cellule de base

La cellule supporte l'arithmétique de base à travers un additionneur 4 bits (connecté à la chaîne de retenue intra-ligne), de la logique *bitwise* à travers une *look-up table* (LUT) et un décalage de 4 bits pouvant diffuser sa commande via la chaîne de retenue.
Dans son implémentation actuelle cette cellule de base est constuite autour de 4 LUTs 8x2 (l'une d'elles étant présentée figure 4.2.1), un additionneur 4 bits (qui peut avoir une de ses entrées inversée) et plusieurs multiplexeurs. L'additionneur 4 bits et la chaîne de retenue sont aussi utilisés pour le calcul des réductions logiques (AND, OR ou XOR d'une ligne).
Les adresses de la LUT peuvent provenir des trois entrées de la cellule selon une des deux configuration (bitwise et mémoire) décrites par la figure 4.2.1. Le mode *bitwise* permet d'appliquer une fonction arbitraire à 3 entrées de manière bit à bit. Le mode mémoire permet d'implémenter n'importe quelle fonction arbitraire à 4 bits d'entrées et 4 bits de sorties sur l'entrée A.
Comme exemple d'exploration architecturale, nous envisagerons dans la suite d'ajouter à cette

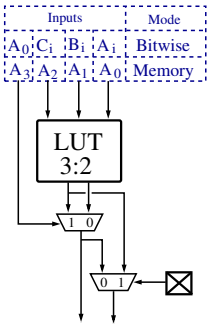

FIGURE 7 – La 8x2 LUT d'indice i d'une cellule, avec ses deux modes d'opération

cellule de base un multiplieur dans $GF(2^4)$ pour le support des fonctions cryptographiques.

5. Études de cas

5.1. Application

Durant cette phase de développement initial du DIRF et de RKC, nous avons utilisées les applications suivantes en étude de cas.

AES

Le *Advanced Encrypion Standard* utilise l'algorithme Rijndael, et se base sur l'arithmétique des corps finis sur $GF(2^8)$ (qui peut être décomposée en une arithmétique sur $GF(2^4)$ [18, 20]). Les détails peuvent être trouvés dans [21]. Ce standard largement répandu a été étudié en détail sur une large variété de cible, du logiciel aux implémentations purement matérielles (ASIC). Par exemple, certains processeurs Intel [6] disposent d'une nouvelle instruction pour le calcul d'un tour *(round)* AES. Plus proche de nos travaux, le projet DREAM a aussi proposé son implémentation de l'AES [15], en ajoutant un support direct pour des sous-opérations de l'AES (e.g : multiplieur dans $GF(2^4)$) dans la matrice reconfigurable PicoGA.
Au moment d'obtenir les résultats reportés ci-après, nous n'avions pas encore implémenté le round AES complet sur notre matrice reconfigurable : la traduction entre $GF(2^8)$ et $GF(2^4)$ été faite à travers une BMM [8, 20], un opérateur déjà présent dans le jeu d'instructions du K1. Nous avons depuis mis au point un algorithme pour traduire simplement les BMM sur des vecteurs dynamiques par des matrices constantes sur notre architecture, mais ce n'est pas l'objet de cet article.

SHA1

Le Standard Hash Algorithm 1 (SHA-1) [4] est un autre algorithme de cryptographie convenant bien à une telle implémentation : il consiste en un nid de boucle avec un cœur très intensif en calcul qui implémente $T = \mathrm{ROTL}(a, 5) + ((b \wedge c) \oplus ((\neg b) \wedge d)) + e + k_i + w_i$ ainsi qu'un autre rotation de 30 bits, toutes les variables étant des entiers 32 bits. Ici nous avons implémenté le corps de boucle complet.

type d'interconnexion	profondeur obtenue	# cellules utilisées	largeur max. d'un niveau
Benes	11	760	100
Crossbar	12	764	96

TABLE 1 – Comparaison sur AES de l'efficacité de deux réseaux d'interconnexion. Il faut noter toutefois que, à capacité de routage comparable, le crossbar est 3,6 fois plus coûteux en surface de silicium que le Benes.

application	profondeur	# cell. utilisées	max. largeur niveau
AES : avec multiplieur et cellule 3 sorties	10	615	84
AES : avec multiplieur et cellule 2 sorties	13	818	95
AES : avec cellule 3 sorties et sans multiplieur	17	1024	128

TABLE 2 – exploration des paramètres de la cellule

FIR

Le denier exemple est un filtre à réponse impulsionnelle finie (FIR) à deux étapes, sur des donnée de 32 bits.

5.2. Exploration architecturale

Le tableau 1 compare l'efficacité de nos deux types de réseau d'interconnexion sur l'implémentation de l'AES.
Le tableau 2 compare plusieurs choix pour la structure de la cellule de base. Par exemple, en réduisant le nombre de sorties dupliquées on obtient une petite réduction de la largeur de la matrice du DIRF, mais cela implique une augmentation plus importante de la profondeur du pipeline , c'est à dire une latence plus grande pour l'opération ainsi qu'un nombre total de cellules utilisées plus grand. Un autre exemple est que notre cellule de base standard contient un multiplieur dans $GF(2^4)$ comme [15], en le supprimant on ne réduit que de 5% l'aire de la cellule mais on augmente dramatiquement la taille de la matrice nécessaire pour AES.

6. Conclusion et perspectives

Nous sommes encore loin de fournir une réponse satisfaisante à la question soulevée dans l'introduction : peut-on réduire le surcoût du calcul reconfigurable à un facteur inférieur à 40 ? La tableau 3 compare la surface du plus petit DIRF capable de contenir une fonction donnée à l'aire de la même fonction, synthétisée directement comme un ASIC. Pour notre meilleur application, SHA1, nous avons encore un surcoût d'un facteur 120.
Bien sûr, ce travail n'en est qu'à un stade préliminaire, et nous avons beaucoup d'opportunités d'optimisations à explorer :

– le grain moyen n'implique que quelques configurations distinctes par application. Par exemple la configuration d'une ligne comme un additionneur 32 bits nécessitant 8 cellules, ne nécessite de stocker que deux configurations de cellule distinctes : 7 cellules identiques avec

application	profondeur	max. largeur niveau	(# cellules utilisées)	area ratio wrt ASIC
AES	10	615	84	320
SHA1	3	48	200	120
FIR	14	55	525	225

TABLE 3 – comparaison d'aire DIRF/ASIC à fonction constante

propagation de retenue, et une différente, sans retenue entrante, pour le poids faible. En utilisant un cache de configuration rempli au lancement de la reconfiguration et en propageant des identifiants de configurations plutôt que des longs *bitstream*, nous allons pouvoir réduire la taille en mémoire des configurations et aussi peut être le temps de configuration.
– nous sommes en train de mettre au point, en nous inspirant d'un travail d'Hilewitz et Lee [9], un réseau de Benes en taille non puissance de 2 capable de supporter les rotations et les shifts dynamiques. Nous comptons nous en servir pour accélérer la virgule flottante sur notre opérateur.
– nous pourrions aussi considérer l'adjonction d'un contrôle minimaliste directement dans la matrice reconfigurable de manière similaire à PicoGA. Une alternative est de permettre l'accès des cœurs K1 aux registres intermédiares du pipeline, pour leur déléguer les éventuelles réinjections et boucles sans ajouter de contrôle interne.
– nous envisageons de nous intéresser à la compilation dynamique, pour que quelques parties spécifiques d'une configuration (comme par exemple celle contenant la clef de l'AES) puissent être directement injectées dans le bitstream de configuration. Ceci enlève une des entrées du problèmes : la clef, en l'injectant dans la configuration.
– nous explorons aussi les configurations multi-contextes pour améliorer notre ratio global de performance : à cause du grain moyen, les opérateurs durs de la cellule de base sont plus coûteux que les registres de configuration, on a donc la possibilité de dupliquer ces registres pour stocker plusieurs configurations et en changer à la volée pour étendre temporalement les capacités du DIRF.
– nous envisageons aussi d'utiliser des bibliothèques de portes logiques spécialement conçues plutôt que les bibliothèques standard pour la synthèse du DIRF.

Durant la réalisation du travail de cet article, RKC représentait 4697 lignes de C++ et 9084 lignes de code python.

Bibliographie

1. Clark (N.), Blome (J.), Chu (M.), Mahlke (S.), Biles (S.) et Flautner (K.). – An architecture framework for transparent instruction set customization in embedded processors. *In : In Proceedings of the 32nd Annual International Symposium on Computer Architecture. Pages*.
2. Cong (J.) et Ding (Y.). – Flowmap : An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs. *IEEE TRANS. CAD*, 1994.
3. David (J.). – *Architecture synchronisée par les données pour système reconfigurable*. – Thèse de PhD, Université Catholique de Louvain, juin 2002.
4. FIPS. – Secure hash standard (shs).
5. Goldstein (S. C.), Schmit (H.), Moe (M.), Budiu (M.), Cadambi (S.), Taylor (R. R.) et Laufer (R.). – Piperench : a co/processor for streaming multimedia acceleration. *SIGARCH Comput. Archit. News*, 1999.
6. Gueron (S.). – Intel's new aes instructions for enhanced performance and security. *In : Fast Software Encryption*, éd. par Dunkelman (O.), pp. 51–66. – Springer Berlin / Heidelberg, 2009.

7. Hadi (P.-A.), Grace (Z.), Philip (B.) et Paolo (I.). – Reducing the pressure on routing ressources of fpgas with generic logic chains.
8. Hilewitz (Y.), Lauradoux (C.) et Lee (R.). – Bit matrix multiplication in commodity processors.
9. Hilewitz (Y.) et Lee (R. B.). – A new basis for shifters in general-purpose processors for existing and advanced bit manipulations.
10. Jääskeläinen (P.), Guzma (V.), Cilio (A.), Pitkänen (T.) et Takala (J.). – Codesign toolset for application-specific instruction-set processors.
11. Kuon (I.) et Rose (J.). – Measuring the gap between FPGAs and ASICs. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, n2, 2007, pp. 203–215.
12. Luu (J.), Kuon (I.), Jamieson (P.), Campbell (T.), Ye (A.), Fang (W. M.) et Rose (J.). – VPR 5.0 : FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling. *In : FPGA'09*. pp. 133–142. – ACM.
13. Mitchell John (M.). – *Medium-Grain Reconfigurable Architecture For Digital Signal Processing*. – Thèse de PhD, Washington State University, may 2006.
14. Mucci (C.), Rossi (D.), Campi (F.), Pizzotti (M.), Perugini (L.), Vanzolini (L.), De Marco (T.) et Innocenti (M.). – The dream digital signal processor. *In : Dynamic System Reconfiguration in Heterogeneous Platforms*. – 2009.
15. Mucci (C.), Vanzolini (L.), Campi (F.) et Toma (M.). – Implementation of aes/rijndael on a dynamically reconfigurable architecture. *In : Proceedings of the conference on Design, automation and test in Europe*.
16. Noori (H.), Mehdipou (F.), Murakami (K.), Inoue (K.) et SahebZamani (M.). – A reconfigurable functional unit for an adaptive dynamic extensible processor. *In : Field Programmable Logic and Applications FPL '06.*
17. Opferman (D.) et Tsao-Wu (N. T.). – On a class of rearrangeable switching networks, part 1 : Control algorithm. *The Bell System Technical Journal*, vol. 50, May-June 1971.
18. Paar (C.). – *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. – Essen, Germany, Thèse de PhD, Institute for Experimental Mathematics, University of Essen, 1994.
19. Park (Y.), Park (H.) et Mahlke (S.). – Cgra express : accelerating execution using dynamic operation fusion. *In : Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*.
20. Rudra (A.), Dubey (P. K.), Jutla (C. S.), Kumar (V.), Rao (J. R.) et Rohatgi (P.). – Efficient rijndael encryption implementation with composite field arithmetic. *In : Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*. pp. 171–184. – London, UK, 2001.
21. Standards (F. I. P.). – Announcing the advanced encryption standard (aes), 2001.
22. Villarreal (J.), Park (A.), Najjar (W.) et Halstead (R.). – Designing modular hardware accelerators in c with roccc 2.0. *In : Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. pp. 127–134. – IEEE Computer Society.
23. Yehia (S.), Clark (N.), Mahlke (S.) et Flautner (K.). – Exploring the design space of lut-based transparent accelerators. *In : Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*.