CAPH, un langage dédié à la synthèse d'applications flot de données sur circuits FPGA

Jocelyn Sérot, François Berry

Institut Pascal, UMR 6602 CNRS/UBP/IFMA
Campus des Cézeaux, 24 Avenue des Landais, BP 80026, 63171 AUBIERE Cedex, France
{Jocelyn.Serot,Francois.Berry}@univ-bpclermont.fr

Résumé − On décrit CAPH, un langage dédié à la synthèse d'applications flot de données sur circuits FPGA. Sont présentés les principales caractéristiques du langage, les outils de développement associés et un exemple d'application.

Abstract − The paper describes the CAPH programming language for implementing data-flow applications on FPGAs. The key features of the language are introduced, the associated tool chain is presented and results are given for an image processing application.

1 Introduction

Les circuits reconfigurables de type FPGA offrent des potentialités très intéressantes en termes d'implantation pour les applications de traitement du signal et des images caractérisées par un parallélisme potentiel important et à grain fin.

La programmation de ces circuits reste néanmoins un exercice délicat, réservé à un public familiarisé avec les concepts, techniques et outils de la conception de circuits numériques en général, et avec les langages de description de matériel comme VHDL ou Verilog en particulier. Ceci limite *de facto* l'usage de ce type de solution.

A terme, et dans le cas général, on peut espérer que les progrès des outils de synthèse de haut niveau, exploitant des descriptions purement comportementales (via des langages comme SystemC ou System Verilog) devraient permettre de réduire ce fossé sémantique et donc de rapprocher (fusionner ?) les communautés des développeurs de logiciel et de matériel.

En attendant, nous pensons que l'usage de langages orientés domaine (DSL, *Domain Specific Languages*) permet d'apporter une réponse, sinon universelle, au moins pragmatique et efficace à ce problème.

Dans cet article, nous présentons donc un DSL dédié à la description et à l'implantation sur FPGA d'applications devant opérer en mode flot de données à la volée des capteurs. Ces applications, du fait des contraintes temporelles associées à la nécessité du traitement à la volée des capteurs, requièrent le plus souvent des puissances de calcul qui excèdent les capacités des processeurs généralistes. Exhibant en général un fort taux de parallélisme à grain fin et souvent destinées à être embarquées, elles constituent de bons candidats pour une implantation sur FPGA.

Le langage CAPH répond précisément au besoin de ces applications, de la phase de spécification et de simulation jusqu'à la phase de génération du code VHDL synthétisable permettant l'implantation sur une cible FPGA quelconque. Il s'agit d'un langage de haut niveau, d'inspiration fonctionnelle, doté de fortes capacités d'abstraction mais générant néanmoins du code dont les performances restent proches d'un code VHDL écrit "à la main". CAPH a déjà été utilisé pour implanter plusieurs applications de traitement d'images temps-réel, en particulier sur des plateformes de type *smart camera* embarquant un FPGA. CAPH est distribué librement sur le site de l'équipe DREAM de l'Institut Pascal [6].

2 Le langage CAPH

L'article n'a pas vocation à faire une présentation complète du langage. Une telle présentation peut être trouvée par exemple dans [5] et, de manière plus formelle dans le manuel de référence du langage [6]. On se contente dans cette section d'une présentation informelle par l'exemple.

CAPH est fondé sur un modèle de calcul de type *flot de données* : les applications y sont représentées sous la forme d'un réseau d'unités de calcul indépendantes (*acteurs*), échangeant des *jetons* par le biais de canaux unidirectionnels de type FIFO.

Le comportement de chaque acteur est spécifié sous la forme d'un ensemble de *règles de transition*. Une règle de transition comprend deux parties : d'une part un ensemble de *motifs*, faisant référence aux entrées de l'acteur ou à des

```
actor suml
  in (a: signed<8> dc)
 out (c: signed<16>)
 var st: {Waiting,Summing}=Waiting
 var acc : signed<16>
 rules
   st:Waiting, a:SoL -> st:Summing, acc:0
 | st:Summing, a:Data v -> acc:acc+v
 | st:Summing, a:EoL -> st:Waiting, c:acc
```

Figure 1: Un acteur calculant la somme de listes

variables internes; d'autre part un ensemble d'*expressions*, ciblant les sorties de l'acteur ou les variables internes. La sémantique associée est celle du filtrage (*pattern-matching*) des langages fonctionnels : une règle est executée dès que les valeurs des entrées (resp. variables internes) satisfont le motif correspondant; dans ce cas, les expressions associées sont évaluées et les valeurs résultantes sont produites sur les sorties correspondantes (resp. écrites dans les variables).

Une des originalités du langage CAPH consiste à distinguer, si besoin, deux catégories de jetons : jetons de *données*, et jetons de *contrôle*. Les premiers correspondent aux données proprement dites (pixels d'une image par exemple), les seconds traduisent la structuration de ces données (signaux de début/fin de trame ou de ligne pour une image par exemple). De cette manière, il est possible d'exprimer des algorithmes opérant sur des structures de données arbitrairement complexes sans recourir à un mécanisme de synchronisation global inter-acteurs, le contrôle étant ici strictement local à chaque acteur.

A titre d'exemple, la figure 1 donne le code d'un acteur `suml` calculant la somme de listes de longueur variable. Les jetons de données sont notés `Data v` et les jetons de contrôle `SoL` et `EoL` (*Start Of List* et *End Of List*). Si le flux connecté sur l'entrée `a`, par exemple, contient, dans cet ordre, les jetons suivants : `SoL, Data 1, Data 2, Data 3, EoL, SoL, Data 4, Data 5, EoL`, alors le flux de sortie sera formé des jetons `6` et `9`.

L'acteur `suml` utilise deux variables internes : l'une comme accumulateur (`acc`), l'autre comme variable d'état (`st`). La variable `st` indique si l'acteur est en train de calculer une somme ou s'il attend le début d'une nouvelle liste. Les trois règles de transition peuvent se lire ainsi : dans l'état `Waiting`, dès qu'un jeton `SoL` est présent sur l'entrée `a`, le lire, initialiser la variable `acc` à 0 et passer dans l'état `Summing`; dans l'état `Summing`, si le jeton présent sur l'entrée `a` est une donnée, l'ajouter à l'accumulateur; s'il s'agit du jeton `EoL`, écrire la valeur de l'accumulateur `acc` sur la sortie `c` et revenir à l'état `Waiting`.

Une autre originalité de CAPH réside dans l'utilisation d'un formalisme purement fonctionnel pour décrire la structure des réseaux d'acteurs formant les applications. Ce langage, qui reprend l'essentiel de la syntaxe des langages *ML-like* (comme `caml`) et doté d'un système de typage polymorphique, permet d'encoder la structure de ces réseaux sous la forme de définitions et d'applications de *fonctions de cablage* (*wiring functions*). Une fonction de cablage est une fonction acceptant et retournant des *flots* (autrement dit, des arcs du graphe représentant le réseau). Ceci est illustré sur la figure. 2, où le réseau d'acteurs sur la gauche est décrit ("généré") par le programme à droite. Deux fonctions de cablage sont utilisées ici : `neigh13` et `neigh33`. La fonction `neigh13` prend un flot et retourne trois flots, correspondant au voisinage 1×3 du flot d'entrée, en instanciant pour cela deux fois l'acteur `dp` (cet acteur retarde de un pixel son flot d'entrée). La fonction `neigh33` prend un flot et retourne neuf flots, correspondant cette fois au voisinage 3×3 du flot d'entrée, ce en appliquant la fonction `neigh13` et en instanciant deux fois l'acteur `dl` (cet acteur retarde de une ligne son flot d'entrée). Sur la figure, les graphes résultant de l'application des fonctions `neigh13` et `neigh33` sont délimités en pointillés courts et longs respectivement.

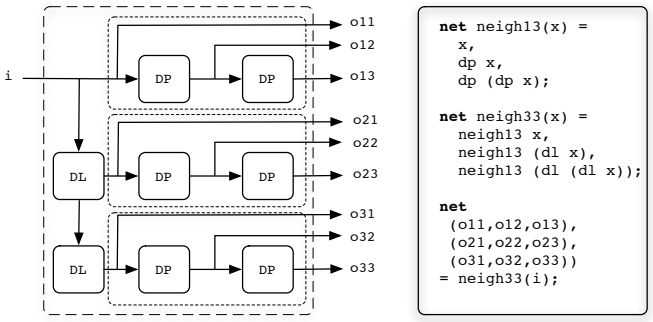

Figure 2: Un réseau d'acteurs et sa description en CAPH

Par rapport à un outil purement graphique cette approche offre un niveau d'abstraction bien supérieur. Elle évite au programmeur d'avoir à décrire *explicitement* les connexions entre acteurs (ce qui est à la fois long et source d'erreurs). Le typage polymorphique permets par ailleurs de s'assurer que les réseaux sont bien formés. Enfin, elle permet la définition de véritables bibliothèques de fonctions de cablage réutilisables. La version de CAPH distribuée contient ainsi toute une collection de telles fonctions correspondant aux motifs de graphes les plus couramment utilisés en traitement du signal et des images de bas et moyen niveaux (extraction de voisinage $n \times n$, filtrage linéaire et non linéaire, *etc.*).

3 Les outils

La chaîne de développement supportant le langage est décrite sur la figure 3. Les principaux outils sont un interpréteur de référence et un compilateur produisant du code SystemC et VHDL.

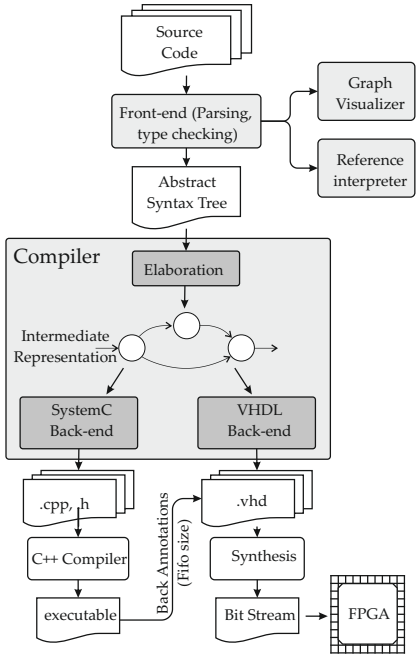

Figure 3: Chaîne de développement

L'interpréteur de référence est déduit strictement de la sémantique formelle du langage donnée dans le manuel de référence du langage [6]. Il permet de simuler et de mettre au point les applications en utilisant toutes les facilités de traçage qui sont plus difficiles à mettre en œuvre au niveau du circuit cible.

Le compilateur proprement dit procède en deux temps : élaboration d'une représentation intermédiaire à partir de l'arbre de syntaxe abstrait typé dans un premier temps, puis génération par des dorsaux spécialisés du code final à partir de cette représentation intermédiaire dans un second temps. La représentation intermédiaire est essentiellement un graphe de processus communiquants où les arcs correspondent à des canaux de type FIFO et où le comportement de chaque processus est encodé sous la forme d'une machine à états finis. Deux dorsaux sont actuellement fournis. Le premier génère du code SystemC précis au niveau bit et cycle (*bit and cycle accurate*), le second du code VHDL synthétisable plateforme-indépendant (portable). Le code SystemC est utilisé pour la simulation et le *profiling*. C'est lui qui permet notamment d'estimer la profondeur des FIFOs instanciées dans le code VHDL (d'où la présence du lien de rétro-annotation sur la figure 3). La synthèse proprement dite du code VHDL sur les circuits cibles est assurée par des outils tiers (nous utilisons l'environnement *Quartus II* d'*Altera*).

4 Résultats

Nous avons utilisé CAPH pour décrire et implanter plusieurs applications de traitement d'images temps-réel sur

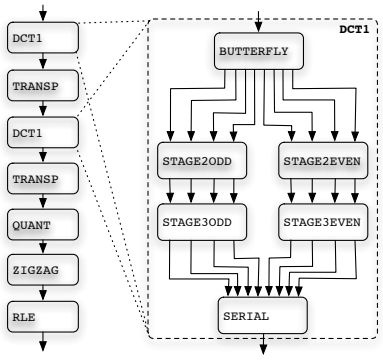

Figure 4: Graphe flot de données pour l'encodeur JPEG4

```
actor transpose
  in (a:signed<32> dc)
  out (c:signed<32> dc)
var s : { S0, S1, S2, S3 } = S0
var z : signed<32> array[64]
var i : unsigned<8>
var j : unsigned<8>
var k : unsigned<8>
rules
| (s:S0, a:SoL)           -> (s:S1, i:0, j:0, k:0)
| (s:S1, a:EoL) when k=8   -> (s:S2, i:0, c:SoL)
| (s:S1)        when j=8   -> (s:S1, i:k+1, j:0, k:k+1)
| (s:S1, a:Data v)         -> (s:S1, i:i+8, j:j+1, z:z[i<-v])
| (s:S2)        when i=64 -> (s:S0, c:EoL)
| (s:S2)                   -> (s:S2, i:i+1, c:Data z[i])
```

Figure 5: Code de l'acteur effectuant la transposition des blocs 8×8

une plateforme de type *smart camera* développée au laboratoire, intégrant un imageur CMOS rapide et un circuit FPGA. Dans cette configuration, deux processus VHDL dédiés sont chargés de l'interface entre le réseau d'acteurs généré par le compilateur et les entrées-sorties physiques (insertion et retrait des jetons de contrôle notamment). Les applications concernées sont, pour ne citer que celles-ci, une extraction de points d'intérêts par filtre de Harris, une détection et localisation d'objets en mouvement, un cœur d'encodeur MPEG-4 et un étiquetage en composante connexes. On décrit ici rapidement l'encodeur JPEG-4 qui constitue la partie centrale de l'encodeur MPEG, 1 (une description beaucoup plus détaillée de l'implantation est donnée dans [4]).

Le graphe complet de l'application est donné sur la figure 4. On y reconnait les étapes classiques de calcul de la transformée en cosinus discrète (DCT), de quantification (QUANT), d'encodage en zig-zag (ZIGZAG) et de compression par longueur de plages (RLE). La DCT est calculée par séparation des lignes et des colonnes, après transposition, en utilisant l'algorithme de Loeffler [1]. Les blocs font 8×8 pixels. La quantification est effectuée par tabulation des quantificateurs et multiplication. Les étapes d'encodage zig-zag et RLE sont classiques. Le code de l'acteur effectuant la transposition des blocs 8×8 pour le calcul de DCT en colonne est reproduit sur la figure 5.

Les performances de l'application, après synthèse sur un circuit de type STRATIX EP1EP1S (via l'outil *Quartus II* d'Altera) sont données dans le tableau 1. La première colonne du tableau donne le nombre total de lignes de code en CAPH et les performances du code VHDL généré par la compilateur. Ces performances sont mesurées en termes de ressources consommées (blocs logiques et mémoire) et de fréquence maximale de fonctionnement (après placement-routage sur la cible). Pour comparaison, la seconde colonne donne les chiffres obtenus avec un codage "à la main", directement en VHDL, de la même application.

La réduction du nombre de lignes de code (facteur > 4) donne une idée du gain en productivité offert par CAPH. Il faut surtout noter que, si l'on excepte les processus dédiés aux entrées-sorties physiques (qui peuvent être codés et fournis une fois pour toute pour une plateforme dédiée), CAPH permet d'implanter cette application sans écrire une seule ligne de VHDL. En termes de performances, le surcoût observé est de l'ordre de 40% en termes de surface et de 20% en termes de fréquence, ce qui reste tout à fait acceptable au regard du gain en expressivité offert.

	CAPH	VHDL
Lignes de code	187	815
Blocs logiques	9686 (17%)	6785 (12%)
Bits mémoire	16728 (¡1%)	12288 (¡1%)
Fréqu. max	40 MHz	47 MHz

Table 1: Résultats expérimentaux pour l'application d'encodage MPEG-4

5 Travaux connexes

CAPH présente de nombreuses similarités avec d'autres langages fondés sur le modèle flot de données et ciblant des implantations sur circuits reconfigurables, CAL [2] et Canals [3] en particulier. Il s'en distingue essentiellement sur deux points : au niveau de la sémantique des règles de transition au sein des acteurs d'une part et au niveau de la description du réseau d'acteurs d'autre part.

CAL et Canals permettent de spécifier des schémas d'ordonnancement complexes au niveau des règles d'activation. En CAL, par exemple, on peut définir des gardes[1] voire des machines à états finis pour spécifier quand telle ou telle règle devient activable. Canals, de son coté, possède un langage permettant de spécifier explicitement l'ordonnancement des acteurs au sein du réseau. Le mécanisme d'ordonnancement est plus simple en CAPH car il se déduit complètement et uniquement de la sémantique du filtrage exprimé par les règles de transition. Ce choix, délibéré, est rendu possible par la présence de jetons de contrôle qui permettent, en quelque sorte, de faire jouer aux données même le rôle d'ordonnanceur. Cette approche simplifie d'autant la génération du code final puisque les acteurs se traduisent alors aisément sous la forme de machine à états, pouvant être synthétisées de manière très efficace en VHDL. Elle facilite par ailleurs grandement l'écriture d'une sémantique formelle du langage, sémantique sans laquelle le processus de compilation est très difficile à documenter et à certifier[2]. On pourrait objecter que cette simplification se fait au détriment de l'expressivité du langage, mais ce n'est pas le cas. En effet, il est toujours possible, si besoin, d'encoder un ordonnancement complexe en utilisant des variable internes.

La description du réseau d'acteurs, en CAL et Canals, se fait via un formalisme de bas niveau, monomorphique et d'ordre 0 : l'utilisateur doit lister l'ensemble des acteurs concernés et les "cabler à la main"en reliant explicitement entrées et sorties[3], ce qui devient rapidement fastidieux et surtout source d'erreur. Le formalisme utilisé dans CAPH pour décrire les réseaux, polymorphique et d'ordre supérieur, offre un niveau d'abstraction bien supérieur. Il autorise par ailleurs la définition de véritables bibliothèques de motifs réutilisables.

[1] Les gardes existent aussi en CAPH, comme illustré sur la figure 5.

[2] Sauf erreur de notre part, aucune sémantique formelle n'a été publiée pour CAL et Canals.

[3] Les dernières versions de CAL ont introduit un outil graphique pour faciliter la tâche, mais cela ne change rien fondamentalement au problème.

References

[1] C. Loeffler, A. Ligtenberg, and G. Moschytz, "Practical fast 1-d dct algorithms with 11 multiplications," in *International Conference on Acoustics, Speech, and Signal Processing,ICASSP-89.*, vol. 2, May 1989, pp. 988 – 991.

[2] C. Lucarz and M. Mattavelli, "Dataflow/actor-oriented language for the design of complex signal processing systems," in *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Brussels, Belgium., 2008.

[3] Dahlin, A., Ersfolk, J., Yang, G., Habli, H., Lilius, J.: The Canals language and its compiler. In: Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems, SCOPES '09, pp. 43–52. ACM, New York, NY, USA (2009).

[4] Ahmed, S. : "Application of a Dataflow Programming Language to the High Level Synthesis of Real-Time Vision Systems on Reconfigurable Hardware", Thèse de doctorat, U. Clermont 2, 2013.

[5] J. Sérot, F. Berry, S. Ahmed. Implementing stream-processing applications on FPGAs : a DSL-based approach. 21st International Conference on Field Programmable Logic and Applications, Chania, Crete, 2-5 sep 2011

[6] J. Sérot Le langage CAPH. `http://dream.univ-bpclermont.fr/index.php/en/caph.html`