Méthode de sélection de checkpoint matériel avec outil de synthèse de haut niveau

Alban Bourge, Alexandre Ghiti, Olivier Muller, Frédéric Rousseau

Méthode de sélection de checkpoint matériel avec outil de synthèse de haut niveau

Alban Bourge, Alexandre Ghiti, Olivier Muller, Frédéric Rousseau
Laboratoire TIMA
46 avenue Félix Viallet
38031 Grenoble CEDEX 1, France

E-mail: {prénom.nom}@imag.fr

Résumé

Les puces reconfigurables (FPGA) représentent une ressource de calcul flexible et puissante mais encore trop peu utilisées dans cette optique. Les outils permettant une intensification de l'intégration des FPGA dans les flots de calculs actuels ne sont pas matures. C'est pourquoi nous cherchons à donner au domaine les outils nécessaires à sa croissance. En particulier, l'article suivant montre comment nous mettons en place une solution permettant de donner une réponse au problème du changement de contexte dans le domaine matériel. Deux spécificités caractérisent notre solution. D'une part, on a cherché à diminuer le surcoût liée à l'extraction du contexte d'une tâche matérielle. D'autre part, la sélection des points de sauvegarde s'effectue à haut niveau, avant synthèse logique, grâce au logiciel AUGH, et apporte donc certains avantages sur lesquels nous reviendrons.

1 Introduction

Dans le cas d'un processeur classique, il est tout à fait indispensable d'échanger les tâches exécutées avant que celles-ci ne s'arrêtent d'elles-mêmes. C'est à l'ordonnanceur d'un système d'exploitation que revient ce travail pour rendre le fonctionnement de la machine fluide. Arrêter une tâche dans le but de la faire reprendre plus tard s'appelle la commutation de contexte. Celle-ci est importante pour que le fonctionnement d'une machine puisse être considéré comme multitâche. Le changement de contexte consiste donc à enregistrer dans la mémoire de l'ordinateur tous les éléments nécessaires au redémarrage de la tâche dans l'état dans lequel elle était avant son arrêt pour ensuite restaurer le contexte d'une tâche en attente. Dans le cas d'une tâche exécutée sur une puce reconfigurable, telle qu'un FPGA, la commutation de contexte est tout sauf évidente à mettre en œuvre.

2 Description du problème

2.1 Changement de contexte matériel

Pour pouvoir effectuer le changement de contexte d'une tâche matérielle, il est nécessaire d'extraire tout ce qui définit l'état de la tâche au même titre qu'une tâche logicielle sur un processeur *general purpose*. Ce qui définit l'état d'une tâche matérielle, ce sont la valeur des registres à l'état considéré, ainsi que le contenu des mémoires internes au FPGA. Dans la suite de l'article, nous nous concentrerons plus particulièrement sur la méthode d'extraction de la valeur des registres car il s'agit de la première étape dans la résolution de cette problématique.

2.2 Méthodes existantes

Il existe déjà plusieurs méthodes pour extraire le contexte d'exécution d'une tâche matérielle. Parmi elles, les méthodes dites CPA et TSAS que nous allons détailler dans les paragraphes suivants. [1] a récemment comparé les deux méthodes, appliquant d'un côté une méthode CPA optimisée et une TSAS de type *memory mapped* déjà éprouvée dans [2].

CPA pour *Configuration Port Access*, est la méthode la plus rapide à mettre en œuvre pour extraire le contexte d'une tâche. Pour programmer un FPGA, on écrit un *bitstream* correspondant à la configuration de chacun des éléments le composant (LUT, FF, éléments de routage ...). Il existe une méthode, dite de relecture ou *readback*, afin de réaliser l'opération inverse. Il va s'agir de lire le contenu du FPGA à un instant donné pour connaître l'intégralité de son contexte. Le désavantage majeur de cette méthode est que le flux de configuration contient énormément de données inutiles dans la connaissance du contexte d'exécution car redondantes !
[3] et [4] proposent un modèle client-serveur complet pour obtenir un fonctionnement multitâche à l'aide de cette méthode. Cette méthode brute n'est plus applicable aujourd'hui car les puces demandent un fichier de configuration beaucoup plus conséquent qu'alors. On peut améliorer l'efficacité de la relecture en ne sélectionnant que certaines zones du FPGA tel qu'il est expliqué dans [5]. Un autre désavantage de cette méthode est qu'elle dépend de la puce utilisée, car le *bitstream* varie en fonction de la technologie. En revanche, l'avantage principal est que le développeur n'a rien à modifier au niveau RTL.

TSAS signifie *Task Specific Area Structure*. [2] a proposé plusieurs mécanismes apparentés, tels que l'ajout d'une interface de lecture à chaque flip-flop (on pense à une chaîne de sérialisation) ou encore l'ajout de composants rendant la mémoire interne du FPGA adressable depuis l'extérieur. Dans tous les cas, cette méthode implique une intervention de la part des développeurs d'application pour ajouter la fonctionnalité voulue, ce qui est son inconvénient majeur. Cette limite est extrêmement bien illustrée dans [6]. Dans cette article, les auteurs présentent leur méthode pour simuler un comportement parallèle en mettant en œuvre des mécanismes complexes (bitoduc CLSA pour *Context Switching Logic Array*, composé de CSFF, CSLUT etc.). Il est dans ce cas impensable de dissocier le développement de l'application et le support du changement de contexte. [2] a présenté une méthode pour abstraire l'implantation des procédés de commutation de contexte mais avec pour objectif la tolérance aux fautes.

La table 1 présente les avantages et les inconvénients des méthodes décrites précédemment. On observe l'opposition des deux méthodes sur tous les points soulevés.

	CPA	TSAS
Taille des données	Importante	Réduite
Efforts de développement RTL	Non	Oui
Dépendant du FPGA	Oui	Non
Dépendant de l'application	Non	Oui
Passage à l'échelle	Possible	Imprévisible

TABLE **1. Résumé qualitatif des méthodes CPA et TSAS**

2.3 Aller plus loin avec le checkpointing

Nous avons vu qu'il est possible d'extraire le contexte d'une tâche matérielle moyennant certaines manipulations. Dans le cas d'une méthode de type CPA, il est clair que les données utiles à extraire sont diluées, et ce malgré les optimisations que l'on peut ajouter. Dans le cas d'une implantation de TSAS, il peut être très coûteux d'être capable de faire une préemption à chaque état d'une tâche. En effet, le coût en surface de ces structures peut devenir prohibitif par rapport au bénéfice retiré.
L'idée est donc de sélectionner des états particuliers de la tâche dont les caractéristiques intrinsèques les favorisent par rapport aux autres. C'est-à-dire qu'il sera plus intéressant de n'autoriser le changement de contexte que dans ces états. C'est la définition d'un *checkpoint*, ou point de préemption. Le système ne pourra extraire la tâche matérielle que si cette dernière est dans un état de ce type.
En définissant un certain nombre de points de préemption bien choisis, on obtient des caractéristiques algorithmiques plus intéressantes que dans un cas général.

3 Notre méthode

3.1 Spécificité

Nous avons remarqué qu'il serait idéal de composer une solution qui pourrait regrouper les avantages des méthodes précédentes. Grâce à un outil de synthèse de haut-niveau (AUGH [7]), il a été possible de cumuler plusieurs atouts :

- Avantages du TSAS :
  - indépendant de la plateforme ;
  - efficace en terme de quantité de données à extraire.
- Avantages du CPA :
  - efforts de développements RTL réduits.

3.2 Description de notre solution

Notre solution s'approche d'une solution du type TSAS tout en s'abstrayant des problèmes d'implémentation.

L'extraction du contexte de la tâche se fait par insertion d'une chaîne de sérialisation (*scan-chain*). C'est un traitement classique pour une application visant un FPGA. Le principe d'une *scan-chain* est décrit en figure 1. On remarque l'ajout de sélecteur (ou *switch*) à l'entrée de chaque élément mémoire permettant la propagation des valeurs. À chaque coup d'horloge, la valeur d'un élément de la chaîne est lue à son extrémité. Dans ce cas simple il faudra autant de coup d'horloge que d'éléments mémoire pour extraire toutes les valeurs.

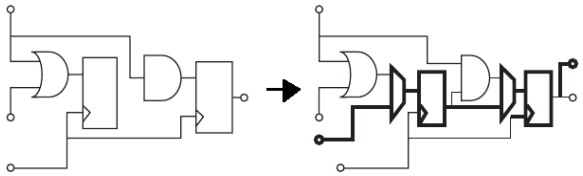
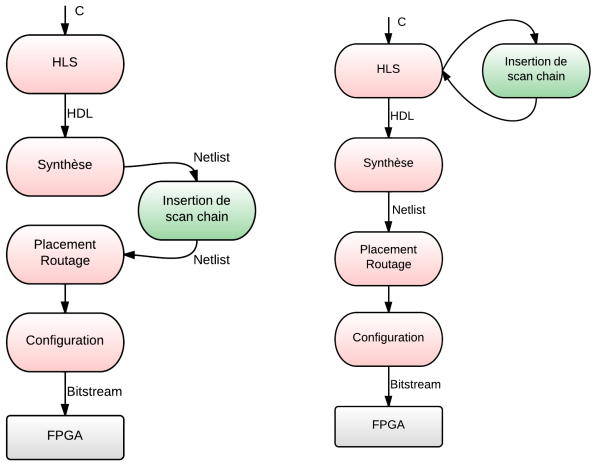


FIGURE **1. Exemple d'insertion d'une chaîne de sérialisation**

L'efficacité en terme de données est assurée. En effet, on utilise un algorithme de sélection de points de sauvegarde pour limiter l'impact de l'insertion de la chaîne de sérialisation, i.e. la quantité de données à extraire pour définir complètement l'état de la tâche. Nous détaillerons cet algorithme en 3.4.

Les indépendances en terme de cible et d'implémentation sont assurées par la nature du flot de conception.
L'indépendance de la plateforme est obtenue directement grâce l'intégration de notre outil sous forme de *plugin* de AUGH. En effet, le fait de travailler sur un modèle haut-niveau d'une application rend l'ajout des structures de sérialisation à la fois transparent pour l'utilisateur



(a) Flot classique

(b) Flot proposé

FIGURE **2. Comparaison des deux flots de conception**

et indépendant de la plateforme visée car le code produit sera compilé par la suite. La figure 2b illustre parfaitement ces deux avantages :

- Étant intégrée à l'outil de HLS, l'intégration de la chaîne de sérialisation n'est pas ajoutée par l'utilisateur ;
- Le HDL étant généré par AUGH (notre outil de HLS), les étapes qui rendent les descriptions suivantes de l'application dépendantes de la cible sont ultérieures.

3.3 Définition mathématique du problème

Pour rappel, afin de limiter l'impact de la chaîne de sérialisation, nous sélectionnons un panel d'états de la tâche dans lesquels il sera possible de faire une commutation de contexte. Ces états présenteront une empreinte mémoire intéressante et permettront de couvrir les états les précédants.

Le problème auquel nous devons faire face est de type ≪problème de couverture par ensembles≫ particulièrement connu. Il est défini dans [8] de la manière suivante :

$$\min \sum_{j=1,n} c_j x_j \text{ avec } Ax \geq \mathbf{1} \text{ et } x \in \{0,1\}^n$$

Où $A = (a_{ij})$ est une matrice $n \times n$ dont les coefficients $\in [0,1]$ et $\mathbf{1}$ est le vecteur de dimension m dont les composantes sont égales à 1. De plus, on ne considère qu'aucune ligne de A n'est identiquement nulle.
En clair, on cherche à sélectionner un certains nombres d'éléments pour couvrir un ensemble tout en minimisant un coût. Appliqué à notre problème, cet énoncé donne les paramètres suivants :

- $x_i = \begin{cases} 1 & \text{si l'état i est un } \textit{checkpoint} \\ 0 & \text{sinon} \end{cases}$
- c_i : le coût de i en tant que *checkpoint*. Le coût de c_i de x_i dépend entre autre de la taille de la chaîne de sérialisation qu'il est nécessaire d'ajouter pour cet état ;
- $a_{ij} \in [0,1]$: représente la proportion dans laquelle l'état i couvre l'état j.

Par la suite, nous utiliserons aussi S_i, qui est l'ensemble des états que couvre i et dans quelles proportions. On aura donc $S_j = \{i \mid a_{ij} > 0\}$, le sous-ensemble de $\{1,2,..,m\}$ qui représente la j^{ieme} colonne de A.

3.4 Algorithme proposé

Le problème de couverture par ensembles est de type NP-complet, tout comme le problème du voyageur de commerce par exemple. C'est donc une heuristique gloutonne que nous avons décidé d'utiliser. Nous n'avons pas sélectionné un algorithme d'approximation pour des raisons de rapidité d'exécution de notre programme. Dans les deux paragraphes qui suivent, nous allons décrire comment nous avons décidé de calculer les hypothèses de notre problème, à savoir les deux paramètres S_i et c_i sur lesquels nous appliquons notre heuristique gloutonne.

S_i représente la couverture de l'état i. Dans la version actuelle de notre programme, cette décision est binaire. Soit un état peut en couvrir un autre, soit il ne le peut pas. C'est une hypothèse simplificatrice ($a_{ij} \in \{0,1\}$). On décide de la couverture d'un état par un autre à l'aide d'un paramètre d'entrée au programme : la latence autorisée. C'est le temps maximal que laisse le système à la tâche pour atteindre un *checkpoint*. C'est donc à partir d'un état de départ dont on veut calculer la couverture qu'on va récursivement trouver les états précédents qui y amènent et décider en fonction du temps restant si oui ou non ces états précédents sont couverts. Plusieurs cas particuliers existent. En effet, il faut pouvoir remonter les boucles et les branchements conditionnels par exemple, ce qui complexifie la recherche récursive. Voici en figure 3 un exemple illustrant la remontée d'un branchement conditionnel. On remarque que la couverture ne se propage pas à l'état $i-5$ tant que toutes ses branches ne sont pas entièrement couvertes. Enfin, quand on cherche à couvrir l'état $i-6$, la latence disponible n'est plus suffisante.

c_i, le coût de i en tant que *checkpoint* est calculé de la manière suivante :

$$c_i = \frac{c'_i}{|S_i|}$$

où c'_i est le coût d'extraction de l'état i, c'est-à-dire la taille en bit de la chaîne de sérialisation à ajouter pour cet état, et $|S_i|$ le nombre d'états couvert par i.

Après calcul de ces hypothèses sur le problème donné en entrée du programme, on va lancer la découverte des *checkpoints*. L'heuristique gloutonne va donc se dérouler en deux étapes se répétant en boucle :

1. Choisir l'état possédant le coût le plus bas ;
2. Réduire le problème initial en enlevant les contraintes portées par l'état i pour reprendre le premier point sur le sous-problème dévoilé (les états déjà couverts sont retirés du problème par exemple).

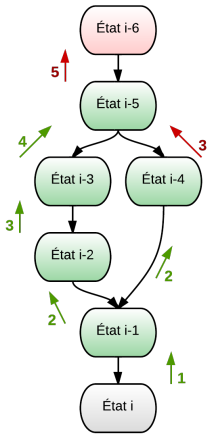


FIGURE **3. Remontée d'un branchement conditionnel (latence disponible = 5)**

4 Implémentation et résultats

Le *plugin* que nous avons développé pour AUGH se nomme CP3, pour *CheckPoint PinPoint*. Les sources de AUGH sont disponibles en [9].

Le tableau 2 donne les résultats obtenus avec un certain nombre d'applications classiques. L'efficacité de notre algorithme est représentée par la quantité de mémoire qu'il ne sera *pas* nécessaire d'enregistrer par rapport à une application utilisant une *scan-chain* complète, sans traitements supplémentaires. C'est donc le gain en pourcentage de la taille de la *scan-chain* optimisée par rapport à la taille de la *scan-chain* complète qui importe. Cette donnée correspond à la dernière colonne de la table 2.

	taille sc totale (bit)	taille sc CP3 (bit)	gain maximal
adpcm	6017	4257	30%
blowfish	760	568	25%
gsm	880	448	50%
mjpeg	5514	1834	67%
sha	3136	416	87%

TABLE **2. Résultats obtenus sur un lot d'applications courantes (sc = scan-chain, cp = checkpoint)**

Notre algorithme donne donc des résultats variables selon les applications, allant jusqu'à une élimination de presque 90% des variables qu'il aurait été nécessaire d'enregistrer dans le cas d'une commutation de contexte impliquant une chaîne de sérialisation complète.

5 Conclusion

On a vu qu'il existait déjà des solutions au problème de l'extraction de contexte sur processeur reconfigurable. Pourtant, aucune de ces solutions n'apporte de réelle flexibilité d'utilisation pour les développeurs d'applications. Notre solution qui s'appuie sur l'outil de HLS open-source AUGH permet d'obtenir des résultats probants et sans surcoût exorbitant ni en terme de développement, ni en terme d'utilisation de ressources matérielles, tout en offrant une optimisation de la commutation de contexte.

La deuxième phase de développement de CP3 consistera à ajouter effectivement la *scan-chain* au niveau de la sortie du logiciel de HLS pour pouvoir commencer les tests sur cible matérielle et l'évaluation de l'impact du traitement sur les ressources.

Références

[1] Krzysztof Jozwik, Hiroyuki Tomiyama, Masato Edahiro, Shinya Honda, and Hiroaki Takada. Comparison of preemption schemes for partially reconfigurable fpgas. *Embedded Systems Letters, IEEE*, 4(2) :45–48, 2012.

[2] Dirk Koch, Christian Haubelt, and 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*, pages 188–196. ACM, 2007.

[3] Harald Simmler, L Levinson, and Reinhard Männer. Multitasking on fpga coprocessors. In *Field-Programmable Logic and Applications : The Roadmap to Reconfigurable Computing*, pages 121–130. Springer, 2000.

[4] L Levinson, Reinhard Männer, M Sessler, and Harald Simmler. Preemptive multitasking on fpgas. In *fccm*, pages 301–302. Citeseer, 2000.

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

[6] Stephen M Scalera and Jóse R Vazquez. The design and implementation of a context switching fpga. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 78–85. IEEE, 1998.

[7] Adrien Prost-Boucle, Olivier Muller, and Frédéric Rousseau. Fast and standalone design space exploration for high-level synthesis under resource constraints. *Journal of Systems Architecture*, 60(1) :79–93, 2014.

[8] Michel Minoux. *Programmation mathématique : théorie et algorithmes*, volume 1. Dunod Paris, 1983.

[9] Adrien Prost-Boucle. Site internet du projet augh, 2013. http ://tima.imag.fr/sls/research-projects/augh/ Consulté le 1er mars 2014.