Gestion des zones en fautes d'une architecture reconfigurable lors du placement des tâches matérielles

Daniel CHILLET, Dinh MA CHIN Olivier SENTIEYS

Laboratoire IRISA
ENSSAT - Université de Rennes 1

Résumé – La gestion des fautes dans les circuits numériques est un point de plus en plus crucial qui nécessite des techniques de détection et d'isolation de celles-ci permettant d'assurer au mieux les fonctionnalités des applications. Pour les circuits FPGA reconfigurables dynamiquement, les fautes dans la partie opérative peuvent engendrer l'impossibilité d'utiliser des grandes zones du circuit, il est donc important de pouvoir proposer des techniques permettant de réduire la zone inutilisable et de conserver un maximum de ressources reconfigurables disponibles. Ce papier propose une gestion à haut niveau du placement des tâches en tenant compte des fautes détectées et en profitant du placement des tâches pour isoler au mieux ces fautes. La technique utilisée est basée sur un modèle "statistique" de localisation des fautes, modèle sur lequel se base l'ordonnanceur pour positionner spatialement les tâches au sein du FPGA.

Abstract – Management of fault is a crucial challenge for electronics devices and in particular, it is necessary to be able to isolate faults in the context of reconfigurable circuit. Indeed, if a fault appears in an FPGA, a large part of the circuit can be unusable and this situation can be unacceptable for the application. The objective consists then in trying to isolate the fault and try to reduce the unusable area as small as possible. To address this issue, we define a Dynamic Fault Localizaion Model which is dynamically updated during the execution of the application. The computation of area in fault, and the statistic of fault, we explain how the model can help the operating system to decide which place can be used for task configuration.

1 Introduction

L'évolution actuelle de la technologique de conception des circuits intégrés conduit à une miniaturisation telle qu'il est de plus en plus délicat d'obtenir des transistors homogènes sur une puce de silicium. Cette évolution peut, dans certains cas, conduire à l'apparition de fautes dans le circuit, fautes permanentes ou fautes dues au vieillissement prématuré de certaines parties du circuit. Dans ce contexte, il est de plus en plus important d'étudier des techniques permettant de s'affranchir de ces fautes pour continuer à assurer une fonctionnalité correcte du circuit. Pour ce qui concerne les FPGA, on peut distinguer plusieurs types de fautes impactant soit la mémoire de configuration soit la partie opérative du circuit. Si les fautes de la partie mémoire de configuration peuvent être en partie détectées et gérées par une relecture du *bitstream*, les fautes au sein de la partie opérative doivent également faire l'objet de technique permettant de les détecter et de les isoler. En particulier, dans la partie opérative, une faute va se traduire par une impossibilité d'utiliser la zone en faute. La difficulté consiste alors à isoler au mieux la zone en faute afin d'offrir la plus grande surface reconfigurable possible pour les tâches à exécuter. Ce papier adresse cet objectif et propose une gestion à haut niveau du placement des tâches au sein d'un FPGA en isolant progressivement les fautes qui peuvent apparaitre dans le circuit. La technique utilisée est basée sur l'hypothèse que nous disposons d'un moyen de détecter des tâches en fautes, des techniques du type redondance fonctionnelle ou codage de données (parité, Hamming, etc) peuvent être envisagées pour détecter la faute et signaler un dysfonctionnement à l'*operating system*. Sur cette base, un modèle "statistique" de fautes est alors développé pour guider le positionnement spatial des tâches au sein du FPGA.

L'organisation du papier suit le séquencement suivant : la section 2 présente un bref état de l'art de la détection et la gestion des fautes pour les circuits de type FPGA. La section 3 présente la modélisation que nous proposons pour réduire progressivement la zone en faute et parvenir à conserver disponible la plus grande zone reconfigurable possible. La section 5 conclue ce papier et expose les perspectives.

Ces travaux étant en cours, nous développons actuellement un algorithme implémentant notre proposition. La version finale du papier présentera des résultats pour des FPGA homogènes, un nombre de fautes variant de une à quelques unités et des ensembles de tâches générés aléatoirement.

2 Etat de l'art

L'apparition de fautes dans un FPGA peut provenir de plusieurs sources.

Les défauts de construction sont généralement permanents, et conduisent à des problèmes de *timing* ou de collage à un état 0 ou 1 [5]. Le vieillissement du circuit est l'une des raisons qui peut conduire le circuit en faute [9]. Des défauts plus ponctuels (*Single Event Transients*) provenant d'une particule énergétique frappant le circuit peuvent changer l'état d'un bit et perturber les calculs ou le stockage [1]. Les fautes logicielles peuvent également provoquer des fautes, notamment les fautes liées à la reconfiguration [4, 7].

Dans [2], les auteurs présentent les différentes méthodes qui sont utilisées pour détecter et éventuellement corriger des fautes. Parmi les techniques utilisées pour gérer les fautes dans un FPGA, la technique développée dans [3] consiste à utiliser les zones en fautes pour les tâches matérielles qui ne sont pas perturbées par la faute en question. Par contre, les tâches qui sont sensibles à la faute seront placées dans une autre zone. De même, les travaux de [8] présentent une technique permettant de tester le FPGA (*built-in self-test*), et proposent un mécanisme permettant de poursuivre l'utilisation du FPGA même en présence de faute au sein du circuit.

L'approche que nous proposons consiste à avoir une gestion à haut niveau du placement des tâches et à profiter d'une analyse fine de celles-ci pour décider, en ligne, à quel emplacement une tâche peut être placée.

3 Modèle statistique de localisation des fautes

3.1 Objectif et modèles

L'objectif de la modélisation "statistique" de la localisation des fautes consiste à isoler au mieux les zones reconfigurables susceptibles de conduire à une mauvaise exécution d'une tâche. Pour parvenir à cet objectif, nous supposons que le FPGA est constitué de blocs de logique reconfigurable élémentaire (CLB Configurable Logic Blocs). Pour présenter notre proposition, nous prenons l'exemple d'un FPGA disposant de ressources homogènes, mais une extension pour un FPGA disposant de ressources hétérogènes est possible en considérant que l'on dispose de plusieurs *bitstreams* différents pour une tâches ou que des techniques de génération de *bitstreams* à partir d'un *bitstream* partiel sont disponibles [6]. Le cas que nous considérons ici peut être assimilé à la disponibilité d'un *bitstream* générique qu'il est possible de venir modifier pour un placement dans la zone souhaitée du FPGA. Dans un FPGA hétérogène, le nombre d'instances maximale imaginable correspondrait à la régularité du FPGA au regard de la taille de la tâche à placer. Il est évident que le stockage de tous les *bitstreams* est un problème en soi, et peut limiter l'intérêt de la méthode si leur nombre est réduit à 2 ou 3 positions possibles.

Au niveau du FPGA, les CLB sont considérés indépendants, et constitue le grain minimum de reconfiguration.

Dans l'état initial, le circuit FPGA est considéré comme étant complètement fonctionnel, c'est-à-dire sans faute. On dira alors que la statistique de fautes est, à cet instant, égale à 0.

Nous définissons $DFLM$ (Dynamic Fault Localization Model) comme étant un modèle statistique dynamique de faute. $DFLM$ est défini par un ensemble de zones reconfigurables, sur lesquelles ont été détectées des fautes dans le circuit. Initialement, c'est à dire à l'instant t_0, $DFLM$ est vide ($DFLM = \emptyset$). Ce qui signifie que le FPGA est supposé sans faute, ou en d'autres termes qu'aucune faute n'a encore été détectée. D'une manière générale, $DFLM$ est l'ensemble des zones reconfigurables z_k telle que $z_k = \{x_k, y_k, w_k, h_k\}$, avec $\{x_k, y_k\}$ la position du la zone reconfigurable en faute, et $\{w_k, h_k\}$ la largeur et la hauteur de cette zone reconfigurable.

L'objectif de la méthode présentée va consister à faire évoluer cet ensemble $DFLM$ en modifiant les zones dans lesquelles des fautes ont été détectées, et à faire en sorte que cet ensemble contienne des zones les plus petites possibles.

L'application $\mathcal{A}$ a exécuter sur le FPGA est modélisée par un ensemble de tâches T_i

$$\mathcal{A} = \{T_i\} \qquad \forall\, i = 0, \ldots, N_T - 1 \tag{1}$$

avec N_T le nombre de tâches de l'application A.

Chaque tâche T_i de l'application A dispose d'au moins une instance sur le FPGA. Nous définissons les positions possibles des instances d'une tâche T_i par la variable $P_{i,m}$ et l'ensemble des positions possibles par PT_i. Cette variable est définit comme suit

$$PT_i = \{P_{i,m}\} \qquad \forall\, m = 0 \ldots, NP_i - 1 \tag{2}$$

avec NP_i le nombre de position possible pour la tâche T_i sur le FPGA, et $P_{i,m} = \{x_{i,m}, y_{i,m}, w_{i,m}, h_{i,m}\}$.

3.2 Principe de la méthode proposée

Sur la surface reconfigurable du FPGA, nous allons supposer qu'il existe une faute quelque part dans le circuit (sur la figure 1.a, la faute est supposée est présente en position $(4; 6)$, sachant que les coordonnées $(0; 0)$ correspondent au coin en bas à gauche). La faute peut affecter un bit de registre, ou encore un bit d'une LUT, etc, mais nous considérons que cette faute affecte la totalité du CLB, et que l'idéal consisterait à pouvoir isoler cette faute à ce niveau de granularité.

À partir de cet état initial, nous considérons alors qu'une application A, composée de différentes tâches T_i, démarre son exécution.

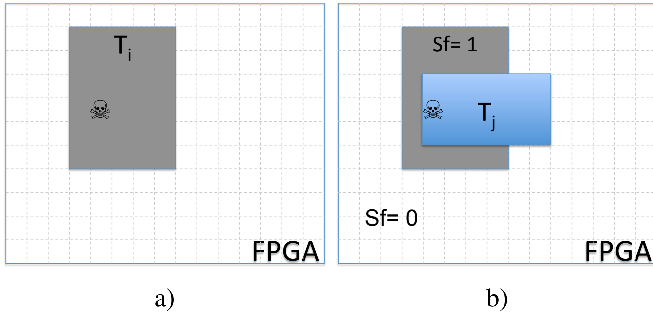
Au temps t_0, nous supposons que le système d'exploitation, disposant d'un ordonnannceur spatio-temporel, décide de placer une première tâche de l'application, dans notre cas, nous considérons qu'il s'agit de la tâche T_i. Si plusieurs tâches sont à exécuter au même instant, nous considérons, pour ces travaux, que l'ordonnanceur est invoqué autant de fois qu'il existe de tâches à exécuter à cet instant. La tâche T_i à exécutée est supposée placée sur l'une des positions possibles $P_{i,m}$, c'est à dire sur l'un des PT_i.

Sur la figure 1.a, nous représentons le placement $P_{i,m}$ de la tâche T_i et nous supposons que le placement couvre la faute présente dans le FPGA. Sur la base de l'hypothèse qu'il est possible de détecter une tâche en faute, nous supposons que la tâche est détectée en faute lors de son exécution à l'instant t_1.

A cet instant, la variable modélisant les zones en fautes est mise à jour, cette mise à jour est faite de la façon suivante.

$$DFLM = \{z_0\} \tag{3}$$

avec $z_0 = P_{i,m}$.



a) b)

FIGURE 1 – a) Placement de la tâche T_i sur une zone ayant un CLB en faute ; Placement d'une seconde tâche T_j sur une partie de la zone en faute.

Une faute ayant été détectée dans cette zone, nous considérons que la région z_0 a une statistique de faute égale à 1. Nous définissons la statistique de faute du FPGA par SF et nous la relions aux zones en fautes de la façon suivante :

$$SF = \{S_{i,m}\} \tag{4}$$

avec $S_{i,m}$ la statistique de faute de la région $z0 = P_{i,m}$.

Ayant détecté cette faute, le système d'exploitation pourra prendre la décision d'exécuter une autre instance de la tâche T_i, instance localisée sur une autre zone supposée sans faute.

Supposons maintenant que le système ait à placer la tâche T_j sur le FPGA et supposons qu'il existe une instance de T_j qui couvre une partie de la zone en faute z_0. Dans ce cas là, deux solutions s'offrent à l'ordonnanceur.

- Si la tâche T_j a une *deadline* très proche et qu'il n'est pas envisageable de l'exécuter deux fois, l'ordonnanceur doit, dans la mesure du possible, s'assurer que le choix de placement à une faible probabilité de faute. Dans ce cas, l'ordonnanceur choisira un placement $P_{j,n}$ tel qu'il n'existe pas de recouvrement avec z_0. D'une façon générale, il faut vérifier que le placement de la tâche T_j ne couvre pas une zone en faute. La vérification peut donc s'exprimer de la façon suivante

$$\nexists k \mid z_k \cap P_{j,m} \neq \emptyset \tag{5}$$

  En d'autres termes, si $\exists k \mid z_k \cap P_{j,m} \neq \emptyset$ alors l'emplacement $P_{j,m}$ ne peut pas être retenue comme position possible sans risquer d'avoir une faute lors de l'exécution de la tâche T_j.
- Par contre, si la tâche T_j a une *deadline* éloignée, permettant d'envisager deux exécutions complètes de la tâche T_j à partir de l'instant courant, alors l'ordonnanceur pourra profiter de cette souplesses pour tenter un placement de la tâche T_j sur un emplacement ayant un recouvrement avec la zone z_0.

Pour l'ordonnanceur, il s'agit alors de vérifier dans un premier temps la *deadline* permet d'envisager au moins deux exécutions complètes de la tâche qu'il doit ordonnancer. Si tel est le cas, alors il doit vérifier s'il existe une instance de la tâche qui permet de recouvrir la zone en faute. Si tel est le cas, alors l'ordonnanceur tente l'exécution de la tâche sur cette zone et observe le comportement de la tâche.

Supposons que nous soyons dans ce dernier cas. La position retenue est alors la position $P_{j,n}$,

avec $P_{j,n} = \{x_{j,n}, y_{j,n}, w_{j,n}, h_{j,n}\}$.

En considérant le placement représenté à la figure 1.b, il y a une probabilité non nulle que la tâche T_j soit également détectée en faute. La probabilité de faute de la tâche T_j est forte si la zone commune entre les tâches T_i et T_j est configurée avec la même fonctionnalité pour les deux tâches. Par contre, si les fonctionnalités configurées dans cette zone sont très éloignées, alors la probabilité de faute pour la tâche T_j est faible.

Pour évaluer cette similarité de fonctionnalités, nous proposons de calculer une distance de Hamming entre les deux bitstreams de la zone commune des deux tâches. On note B_i (respectivement B_j) le bitstream de la tâche T_i (respectivement de la tâche T_j). On définit alors la zone commune entre les deux tâches T_i et T_j par $P_{i,m \cap j,n}$, avec $P_{i,m \cap j,n} = P_{i,m} \cap P_{j,n}$.

À partir de cette définition, on peut définir $B_{i,P_{i,m \cap j,n}}$ et $B_{j,P_{i,m \cap j,n}}$ les bitstreams partiels des deux tâches T_i et T_j. Ces bitstreams partiels correspondent à la zone commune $P_{i,m \cap j,n}$.

La distance de Hamming est alors calculée comme suit :

$$H(B_{i,P_{i,m \cap j,n}}, B_{j,P_{i,m \cap j,n}}) = \sum_{P_{i,m \cap j,n}} B_{i,P_{i,m \cap j,n}} \oplus B_{j,P_{i,m \cap j,n}} \tag{6}$$

Finalement, la similarité entre les deux bitstreams est calculée par

$$S(B_{i,P_{i,m \cap j,n}}, B_{j,P_{i,m \cap j,n}}) = 1 - \frac{H(B_{i,P_{i,m \cap j,n}}, B_{j,P_{i,m \cap j,n}})}{Size(B_{i,P_{i,m \cap j,n}})} \tag{7}$$

Si les deux bitstreams partiels sont identiques (indiquant que la même fonctionnalité est configurée dans la zone commune pour les deux tâches), alors la similarité $S(B_{i,P_{i,m \cap j,n}}, B_{j,P_{i,m \cap j,n}})$ est égale à 1. Si les deux bitstreams sont "exactement" différents, la similarité est égale à 0.

La figure 2.a présente le cas où les deux bitstreams sont exactement similaires, dans ce cas, il peut être conclu que la zone en faute est réduite à la zone commune entre les deux tâches. Nous avons donc une mise à jour de $DFLM$ comme suit

$$DFLM = \{P_{i,m \cap j,n}\} \tag{8}$$

et une mise à jour de la statistique des fautes comme suit

$$SF = \{Sf_{i,m \cap j,n}\} \tag{9}$$

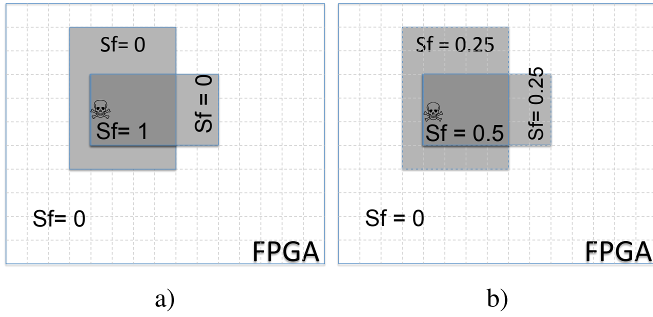
Attention, la mise à jour de $DFLM$ n'est effectuée que dans le cas où la nouvelle tâche placée est détectée en faute. Si aucune faute n'est détectée, la statistique des fautes reste inchangée et basée sur les exécutions précédentes ayant conduit à étiqueter des zones en faute.

La figure 2.b présente le cas où les deux bitstreams ne sont pas similaires, par exemple avec une similarité de $0,5$

$S(B_{i,P_{i,m \cap j,n}}, B_{j,P_{i,m \cap j,n}}) = 0,5$.

Dans ce cas, les statistiques de fautes doivent être adaptées en considérant qu'il n'y a pas de certitude que la faute détectée lors de l'exécution de la tâche T_j soit la même que celle détectée lors de l'exécution de la tâche T_i. Une répartition de la statistique de faute est alors proposée.

$$DFLM = \{P_{i,m \cap j,n} \,;\, P_{i,m} \,;\, P_{j,n}\} \qquad \text{et} \tag{10}$$



a) b)

FIGURE 2 – Calcul des statistiques de fautes des différentes zones avec T_i et T_j a) similaires ; b) de similarité $0,5$.

$$SF = \{Sf_{i,m \cap j,n} \, ; \, Sf_{i,m} \, ; \, Sf_{j,n}\} \qquad (11)$$

Ici on applique le facteur de similarité S pour la zone commune des tâches, et on répartit la statistique restante sur les autres zones des tâches T_i et T_j. La figure 2.b présente cette répartition.

Remarque : Pour éviter d'avoir à gérer des zones autres que rectangulaires, les valeurs stockées dans l'ensemble SF sont triées par ordre décroissants de statistique de fautes, permettant ainsi de trouver la statistique maximale en premier lors de la recherche d'un positionnement pour une tâche.

Le même raisonnement est alors poursuivis tout au long de l'exécution de l'application. Au final, l'ordonnanceur va progressivement pouvoir réduire la zone dite en faute. Les zones dont les statistiques de fautes sont faibles vont également pouvoir se réduire et il peut alors être fixé un seuil en dessous duquel la statistique de faute permet de considérer qu'il n'existe pas de faute dans ladite zone.

4 Premiers résultats

Pour illustrer le fonctionnement de cette stratégie, nous avons développer un outil qui permet de simuler le placement des tâches en respectant l'ordre d'exécution qui est donné par un graphe de tâches. Lorsque 2 tâches sont à exécuter en parallèle, l'outil définit un ordre de placement aléatoire. Les expérimentations réalisées considèrent un FPGA de taille $10000 * 10000$ CLB ayant une faute dont le positionnement n'est pas connu, et des tâches occupant entre 10 et 20% du FPGA avec des positions définies aléatoirement au sein du FPGA. Pour simplifier la présentation de ces premiers résultats, nous considérons ici que la similarité est égale à 1 pour les parties commune des tâches, cette simplification permet de restreindre plus facilement la zone en faute, mais ne permet pas de considérer plusieurs fautes dans le circuit.

Pour un graphe de tâches comptant 24 tâches, offrant une flexibilité de placement dans le FPGA, l'algorithme proposé permet de restreindre la zone en faute à une surface réduite de taille inférieure à 1% du FPGA. Par placements successifs sur le FPGA (et donc parfois sur a zone en faute), de tâches se recouvrant partiellement, l'algorithme parvient progressivement à isoler la zone en faute. En imaginant que le FPGA dispose de plusieurs fautes, l'algorithme pourrait parvenir à les isoler sous la condition que les fautes soient éloignées et qu'aucune tâche ne recouvre les deux fautes en même temps, ce qui est assez peu probable. Une extension de l'algorithme serait alors nécessaire pour prendre en compte ces cas de figures.

5 Conclusion

L'apparition de fautes dans les circuits, qu'ils soient de type processeurs ou architectures reconfigurables est une problématique qu'il est nécessaire d'étudier afin de proposer des solutions permettant d'isoler ces fautes et de poursuivre l'exécution des applications. Ce papier adresse la gestion du placement de tâches sur architectures reconfigurables avec pour objectif de limiter la zone inutilisable pour le placement des tâches. L'approche retenue consiste à définir un modèle statistique de faute qui s'adapte au fur et à mesure de la détection de fautes dans le circuit. Le modèle statistique est adapté au cours du temps et sert de support pour l'ordonnancement spatio-temporel des tâches. La dynamité du modèle est importante puisqu'elle permet une réduction progressive des zones en fautes et permet ainsi de pouvoir conserver un maximum de ressources reconfigurables disponibles. L'outil de simulation qui a été développé dans le cadre de ces travaux n'est pas complet puisqu'il ne tient pas compte du cas des architectures hétérogènes de FPGA. Cette particularité sera prise en compte dans une prochaine version et nous permettra de vérifier que notre proposition permet également de gérer des FPGA commerciaux classiques.

Références

[1] P. Bernardi, M.S. Reorda, L. Sterpone, and M. Violante. On the evaluation of seu sensitiveness in sram-based fpgas. In *On-Line Testing Symposium, 2004. IOLTS 2004. Proceedings. 10th IEEE Int*, pages 115–120, July 2004.

[2] Jason A. Cheatham, John M. Emmert, and Stan Baumgart. A survey of fault tolerant methodologies for fpgas. *ACM Trans. Des. Autom. Electron. Syst.*, 11(2) :501–533, April 2006.

[3] J.M. Emmert, C.E. Stroud, and M. Abramovici. Online fault tolerance for fpga logic blocks. *Very Large Scale Integration (VLSI) Systems, IEEE Trans on*, 15(2) :216–226, Feb 2007.

[4] N. Goel and K. Paul. Hardware controlled and software independent fault tolerant fpga architecture. In *Advanced Computing and Communications, 2007. ADCOM 2007. Int Conference on*, pages 497–502, Dec 2007.

[5] I.G. Harris and R. Tessier. Testing and diagnosis of interconnect faults in cluster-based fpga architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans on*, 21(11) :1337–1343, Nov 2002.

[6] Yoshihiro Ichinomiya, Motoki Amagasaki, Masahiro Iida, Morihiro Kuga, and Toshinori Sueyoshi. A bitstream relocation technique to improve flexibility of partial reconfiguration. In Yang Xiang, Ivan Stojmenovic, BernadyO. Apduhan, Guojun Wang, Koji Nakano, and Albert Zomaya, editors, *Algorithms and Architectures for Parallel Processing*, volume 7439 of *LNCS*, pages 139–152. Springer Berlin Heidelberg, 2012.

[7] K.A. Kwiat, Jr. Debany, W.H., and S. Hariri. Software fault tolerance using dynamically reconfigurable fpgas. In *VLSI, 1996. Proceedings., Sixth Great Lakes Symposium on*, pages 39–42, Mar 1996.

[8] M. Naseer, P. Sharma, and R. Kshirsagar. Fault tolerance in fpga architecture using hardware controller - a design approach. In *Advances in Recent Technologies in Communication and Computing, 2009. ARTCom '09. Int Conference on*, pages 906–908, Oct 2009.

[9] Suresh Srinivasan, Prasanth Mangalagiri, Yuan Xie, N. Vijaykrishnan, and Karthik Sarpatwari. Flaw : Fpga lifetime awareness. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 630–635, New York, NY, USA, 2006. ACM.