Plateforme matérielle–logicielle d'émulation de fautes pour des opérateurs arithmétiques

Pierre Guilloux, Arnaud Tisserand

Compas'2016 : Parallélisme/ Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Plateforme matérielle–logicielle d'émulation de fautes pour des opérateurs arithmétiques

Pierre GUILLOUX[3,1] et Arnaud TISSERAND[2,1]

[1]IRISA, [2]CNRS – [3]Université Rennes 1 – INRIA, 6 rue de Kerampont, 22305 Lannion.

Résumé
Nous présentons les premiers développements d'une plateforme matérielle–logicielle d'émulation de fautes dans des opérateurs arithmétiques matériels. Basée sur un réseau de cartes FPGA intégrant des processeurs multicœurs embarqués et un serveur pour les outils de CAO, elle permet d'évaluer rapidement et précisément de nombreuses techniques de détection de fautes appliquées à différents opérateurs arithmétiques.

Mots-clés : tolérance aux fautes, émulation de faute, FPGA, conception matérielle–logicielle.

---

1. Introduction

Différentes techniques de détection ou de tolérance aux fautes ont été proposées pour les circuits intégrés numériques (voir p. ex. [2, 7]). Dans le cas des opérateurs arithmétiques (voir une introduction dans [9]), sélectionner une bonne technique, et ses paramètres, en fonction des types d'opérations, des algorithmes de calcul, des représentations des nombres, des diverses optimisations de l'architecture et du circuit, et du niveau d'erreur mathématique autorisé (incluant l'effet des fautes et des arrondis) est une tâche souvent complexe (voir p. ex. [14]).
La modélisation à haut niveau de l'impact de fautes dans un circuit arithmétique est souvent superficielle. Se contenter de simuler des fautes en entrée des opérateurs n'est pas suffisant. Il faut le faire dans le cœur même de l'opérateur selon différents scénarios, p. ex. sur les signaux/portes internes pour les données intermédiaires mais aussi sur le contrôle de l'opérateur. On recourt souvent à des simulations de bas niveau, de type « bit précis et cycle précis », de l'opérateur dans lequel on injecte successivement de nombreuses fautes dans le temps selon différents scénarios représentatifs. Cette technique purement logicielle est simple mais limitée par la vitesse des simulateurs. Les circuits FPGA sont souvent utilisés pour accélérer de telles simulations : on parle alors d'*émulation matérielle* (voir Section 2).
Nous présentons les premiers développements d'un projet d'aide à la conception et au prototypage de circuits arithmétiques tolérants aux fautes sur ASIC pour différents projets de recherche. En section 3, nous décrivons une *plateforme* basée sur un petit *réseau de cartes FPGA Zynq Xilinx* et d'un *serveur Linux*. Les parties reconfigurables des FPGA permettent d'*instrumenter* les opérateurs arithmétiques avec des *blocs d'injection de fautes*. Le serveur héberge les *outils de CAO* et le *contrôle* de la plateforme. L'interface entre FPGA et serveur est assuré par un Linux embarqué sur les cœurs ARM des Zynq et un programme de gestion, sur le serveur, des tâches d'émulation parallèles sur les FPGA. En section 4, nous présentons un exemple d'analyse de la précision d'un multiplieur entier 16×16 bits en présence de différents scénarios de fautes.

2. Techniques d'accélération de la simulation de fautes

Fabriquer un circuit pour « seulement » le tester est rarement possible. On modélise plutôt l'effet de fautes à différents endroits du circuit par simulation. La sélection des points d'injection et du type de fautes, *collage* à 0/1 ou *inversion*, est liée à la technologie et au type de faute envisagés : p. ex. distribution uniforme d'inversion sur la surface pour des fautes dues à des rayonnements ; des collages plus probables sur les signaux/portes les plus actifs ou chargés (*fanout*) ; des fautes plus probables en fin de chemin critique pour des problèmes temporels.
De nombreux travaux portent sur l'accélération logicielle de telles simulations. Par exemple, au début des années 90, l'algorithme *Proofs* parallélisait la simulation de plusieurs collages sur un même mot machine 32 bits [18]. Différentes techniques de parallélisation ont été proposées, p. ex. sur des machines parallèles généralistes [17] ou plus récemment sur des GPU [12].
L'émulation de fautes en matériel a été proposée très tôt dans l'histoire des FPGA [5, 13]. On implante en logique reconfigurable une version *instrumentée* d'une description bas niveau du circuit (*netlist*), provenant d'un flot de conception de circuits, afin que des fautes soient simulées à différents endroits de la description originale comme illustré en figure 1. Souvent un bloc matériel d'*analyse en-ligne* sur le FPGA permet, p. ex., de déterminer le taux de détection de fautes (pour éviter de devoir sortir de gros volumes de données en dehors du FPGA).

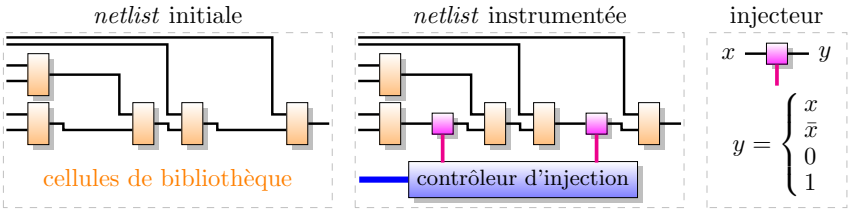

FIGURE 1 – Principe des points et du contrôleur d'injection de fautes.

Le circuit instrumenté est plus grand et plus lent du fait des *injecteurs* de fautes et de l'*analyse en-ligne*, p. ex. comparaison avec une version sans fautes. Mais il fonctionne quand même à plus de 100 MHz sur des FPGA récents et surpasse ainsi des simulations logicielles, y compris parallèles. Utiliser une nouvelle configuration (*bitstream*) du FPGA pour chaque type et chaque position d'injection est très coûteux (temps de synthèse, placement/routage et chargement de la configuration). On place plutôt de nombreux points d'injection qui émulent successivement dans le temps différents *scénarios*. Pour une faute unique p. ex., tous les injecteurs effectuent $y = x$ sauf celui choisi qui effectue $y \in \{0, 1, \bar{x}\}$ selon le modèle de faute retenu. De nombreux travaux portent sur l'émulation de fautes en FPGA : p. ex. [11, 16, 8, 6, 4]. La reconfiguration dynamique partielle est aussi utilisée pour accélérer les émulations [1, 15]. Des cartes Zynq Zedboard ont été utilisées pour estimer le nombre de bits critiques du *bitstream* d'un FPGA [21].
L'émulation FPGA est aussi utilisée, de façon similaire, pour accélérer des simulations « électriques » d'opérateurs arithmétiques, p. ex. consommation d'énergie de multiplieurs [20] ; évaluer la robustesse d'opérateurs cryptographiques aux attaques par analyse de la consommation d'énergie [19].

3. Description de la plateforme matérielle–logicielle

Pareillement à de nombreux travaux de la littérature, nous émulons des fautes de collage et d'inversion en insérant des injecteurs de fautes à des endroits choisis de la *netlist* comme illustré en figure 1. Nous ne pouvons pas modéliser les fautes se produisant sur un transistor donné,

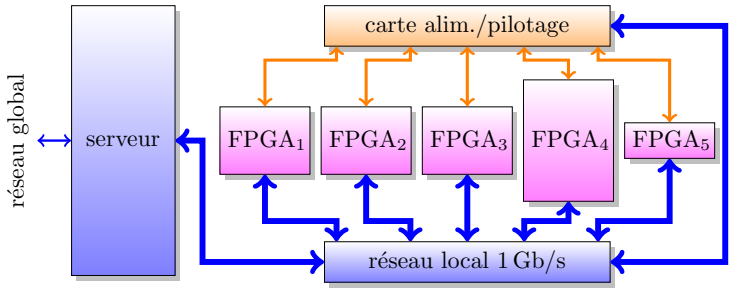

FIGURE 2 – Architecture globale de la plateforme d'émulation de fautes.

mais nous supposons, comme dans la plupart des travaux, que l'effet est similaire à celui d'une faute de collage ou d'inversion sur la sortie de la porte contenant le transistor.
Notre plateforme matérielle est décrite en figure 2. Elle comporte : 1 serveur Debian (8 cœurs 4.0 GHz, 32 Go RAM, 128 Go SSD, 1 To HDD, 2 GbE, coût 1100 €) ; des cartes FPGA Zynq Xilinx ; 1 *switch* ethernet 1 Gb/s (180 €) ; 1 carte d'alimentation électrique et de pilotage maison (200 €). Actuellement, les cartes FPGA sont : 3 Zedboard (Zynq 7020 à 250 € pièce) ; 1 PicoZed (Zynq 7035 à 810 €) ; 1 Zybo (Zynq 7010 à 150 €). Le coût total de la plateforme, avec câblage et accessoires, est d'environ 3500 €HT.
Les circuits Zynq embarquent un processeur double cœur ARM Cortex A9 et un FPGA série 7. La partie reconfigurable (PL, *programmable logic*) va porter les *blocs d'émulation* contenant l'opérateur matériel évalué et l'instrumentation nécessaire à l'émulation de fautes, cf. figure 3. La partie processeur (PS, *processing system*) va recevoir l'interface entre la PL et le serveur ainsi que la gestion à haut niveau de l'émulation (p. ex. gestion des fichiers de données/résultats).
Un bloc d'émulation, fig. 3, se compose des ressources matérielles suivantes :

- l'opérateur original, sans instrumentation, qui servira de référence pour l'analyse en-ligne : pour l'entrée x, il calcule la sortie $y = f(x)$ où f est la fonction de l'opérateur ;
- l'opérateur instrumenté avec des injecteurs de fautes qui calcule la sortie « fautée » $\widetilde{y}$;
- le contrôleur d'injection pilote chaque injecteur selon les scénarios définis ;
- la mémoire et le générateur de vecteurs fournissent des entrées aux opérateurs ;
- l'unité d'analyse en-ligne (cf. plus bas) avec sa mémoire temporaire de résultats ;
- le contrôle global pilotant l'ensemble des activités du bloc d'émulation ;
- enfin, l'interface vers la PS.

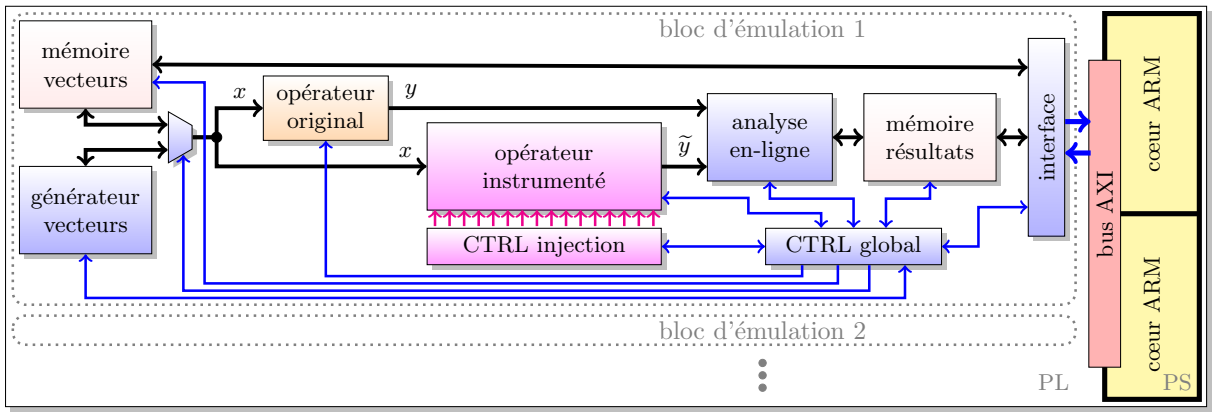

FIGURE 3 – Architecture d'un bloc d'émulation matérielle et son interface logicielle.

L'analyse en-ligne peut être la simple détermination du taux de couverture de fautes (nb. fautes détectées vs. nb. fautes injectées), celle de la moyenne ou de la distribution de l'erreur mathématique de la sortie « fautée ». Dans ces derniers cas, on évalue l'erreur $\varepsilon = |y - \widetilde{y}|$ pour chaque entrée x donnée et chaque élément du scénario de fautes émulé. Pour la moyenne c'est très simple, un simple accumulateur suffit, la division finale se fait en PS pour éviter un diviseur très peu actif. Pour la distribution, il faut stocker de nombreuses valeurs, idéalement autant que de valeurs d'erreur possibles. En pratique, nous découpons le domaine d'erreurs en intervalles. Chaque mot mémoire de résultats stocke le nombre de valeurs de ε dans l'intervalle correspondant dont l'adresse est, p. ex., les k bits de poids forts de ε pour une mémoire de 2^k mots. D'autres schémas de découpage sont possibles au prix de plus de calculs en-ligne.

Le flot logiciel de la plateforme, présenté en figure 4, s'exécute essentiellement sur le serveur. La description VHDL originale de l'opérateur est transformée par les outils ASIC commerciaux, pour nous Synposys en synthèse physique et Cadence pour le *backend*, en une description de bas niveau (*netlist*). Cette *netlist* contient l'ensemble des cellules de bibliothèque utilisées à plat, c.-à-d. sans hiérarchie interne, et leurs connexions. Elle prend en compte la sortance (*fanout*) purement logique comme celle des cellules de recodage de Booth reliées à de nombreuses cellules de produits partiels d'un multiplieur [10, Chap. 3], p. ex. un bit a_i recodé du multiplieur « rencontre » tous les bits b_j du multiplicande pour former les différents $a_i b_j$. Nous prendrons en compte prochainement la probabilité de faute liée aux capacités parasites du routage.

Un programme ajoute des injecteurs de fautes à différents endroits de la *netlist* originale pour produire la *netlist* instrumentée. Actuellement, nous gérons une répartition aléatoire sur les signaux internes ou une répartition aléatoire uniforme sur la surface (en utilisant la surface des différentes cellules comme métrique de choix). Notre programme ajoute à ce résultat de *netlist* instrumentée les autres éléments du bloc d'émulation de la figure 3. Le contenu du bloc d'émulation, en particulier le contrôleur d'injection de fautes, est généré par un autre programme dont l'entrée utilisateur est le(s) choix sur le(s) scénario(s) de fautes à considérer et les types de vecteurs d'entrées (génération ou bien fichier de données). La génération de vecteurs d'entrée peut être un simple réseaux de compteurs ou des LFSR (*linear feedback shift register*), ou certains de leurs dérivés, pour engendrer des distributions de données particulières. La mémoire de vecteurs d'entrée est utilisée dans le cas de données d'applications représentatives.

Selon les scénarios de fautes considérés et le type d'opérateur évalué, plusieurs blocs d'émulation peuvent fonctionner en parallèle dans un même circuit. Différentes solutions sont possibles comme utiliser un même fichier de données d'entrée diffusé à tous les blocs d'émulation et des

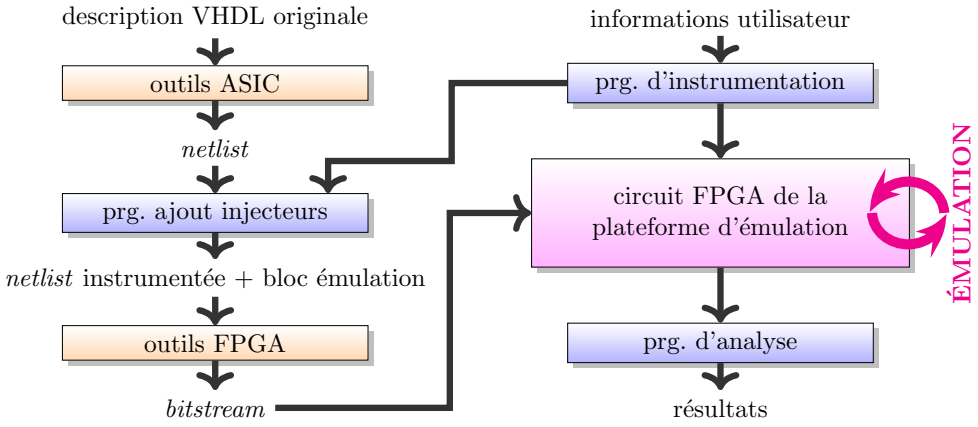

FIGURE 4 – Flot de l'outillage de la plateforme d'émulation (en bleu nos programmes).

scénarios différents ; ou bien utiliser des générateurs de vecteurs d'entrée différents et un même scénario de fautes. Nous travaillerons sur d'autres techniques prochainement.
Les blocs d'émulation sont synthétisés, placés/routés pour former la configuration (*bitstream*) du FPGA servant à l'émulation. Le *bitstream* est alors chargé sur le FPGA à travers le réseau et l'émulation est lancée. Enfin, les résultats de l'émulation sont envoyés au serveur pour l'analyse finale. La partie logicielle sur la PS du circuit Zynq gère la récupération du *bitstream*, le contrôle de haut niveau de l'émulation dans chaque FPGA et l'envoi des résultats vers le serveur.
Notre fréquence cible pour le bloc d'émulation est de 100 MHz pour simplifier les échanges PL↔PS via le bus AXI. Un autre avantage de fonctionner « seulement » à 100 MHz est que tout le contrôle est simple sans grand besoin de pipeline. On peut ainsi émuler 100 millions de fautes et/ou de données d'entrée en une seconde, ce qui est plusieurs ordres de grandeur plus rapide que des simulations logicielles, comme le montrent les autres travaux du domaine, p. ex. [11] affiche une accélération de $\times 318$ et même $\times 517$ pour [8].
Le fait d'avoir plusieurs cartes FPGA permet de recouvrir les temps d'émulations et ceux des transferts de données sur le réseau. Nous déployons sur la plateforme un système libre de gestion et d'ordonnancement de tâches parallèles pour grappe de machines en utilisant SLURM (*Simple Linux Utility for Resource Management* `http://slurm.schedmd.com/`), mais il nous reste encore beaucoup de travail sur cet aspect.

4. Exemple d'application : multiplieur protégé par code résidu

Nous présentons ci-dessous un petit exemple d'utilisation de la plateforme qui illustre les possibilités de test intensif par émulation matérielle. L'opérateur évalué est un multiplieur entier 16 bits non signé, $p = a \times b$, auquel on ajoute un code résidu pour détecter des fautes, le tout totalement combinatoire. La description du multiplieur provient de la bibliothèque d'opérateurs arithmétiques de Reto Zimmernann disponible sur le web [22]. La détection par code résidu est décrite p. ex. dans [14, 3]. Avec m le *modulo*, un entier naturel p. ex. 3, 5, 7, 15, 31, 127, elle se base sur la relation suivante :

$$(a \times b) \bmod m \quad = \quad \big((a \bmod m) \times (b \bmod m) \big) \bmod m$$

La détection est très simple. En marge du produit complet $p = a \times b$, on calcule les restes, ou résidus, $r_a = a \bmod m$ et $r_b = b \bmod m$, puis leur produit $p_r = r_a \times r_b$. Enfin on compare $p \bmod m$ et $p_r \bmod m$. Ces deux valeurs diffèrent lorsqu'il y a une erreur (entière dans ce cas) non congrue à m (c.-à-d. $\mathrm{pgcd}(|y - \widetilde{y}|, m) = 1$).
Dans un premier temps, nous évaluons le multiplieur sans protection pour différents scénarios d'inversion d'un signal interne de la *netlist* choisi aléatoirement parmi les 1132 signaux internes de l'opérateur. Pour chaque scénario, nous testons exhaustivement les $2^{16} \times 2^{16}$ valeurs d'entrée. La génération des vecteurs d'entrée est triviale puisqu'il suffit de 2 compteurs 16 bits pour fournir les deux opérandes a et b.
La table 1 présente les principaux résultats obtenus aux différentes étapes du flot. On remarque qu'avec 30 points d'injection, la surface et la vitesse du multiplieur instrumenté ne sont que peu modifiées. Un bloc d'émulation complet occupe moins de 4 % de la surface d'un FPGA 7020 (la marge au niveau des BRAM est encore plus importante). On peut donc envisager 25 blocs d'émulation par FPGA pour des petits opérateurs et des scénarios de faute très simples.
Nous avons comparé ces résultats d'émulation à une simulation RTL (*register transfert level*) du multiplieur instrumenté avec l'outil de simulation de Vivado 2014.4. Dans le cas d'un seul bloc d'émulation sur un seul FPGA, nous obtenons un facteur d'accélération de plus de $\times 6880$ par

étape	durée outil	surface	délai op.
Outils ASIC multiplieur original	25 s	1164 portes	–
Outils FPGA multiplieur sans instr.	43 s	444 LUT	9.4 ns
Outils FPGA multiplieur avec instr.	48 s	453 LUT	10.1 ns
Outils FPGA bloc émulation	389 s	1714 LUT + 1 BRAM	11.0 ns
Boot petalinux Zedboard	15 s	–	–
Temps émulation 1 faute sur 1 bloc	94.5 s	–	–

TABLE 1 – Résultats d'implantation du multiplieur pour les différentes étapes du flot avec 30 points d'injection de fautes pour la version instrumentée (un FPGA Zynq 7020 contient 53 000 LUT6 et 140 BRAM 36Kb).

rapport à la simulation logicielle sur la même machine. Des accélérations encore plus grandes sont trivialement possibles avec des blocs d'émulations et des FPGA en parallèle dans le cas de génération exhaustive des vecteurs d'entrée (il suffit de « découper » et répartir les compteurs). Dans le pire cas, il y a 2^{32} valeurs d'erreur possibles. Nous utilisons un découpage uniforme en 256 paquets d'erreur. Pour le paquet 0, le cas sans erreur ($y - \widetilde{y} = 0$) n'est pas compté comme élément de ce paquet. La figure 5 présente la distribution d'erreur obtenue pour différentes positions aléatoires du signal interne fauté (une par couleur sur la figure).

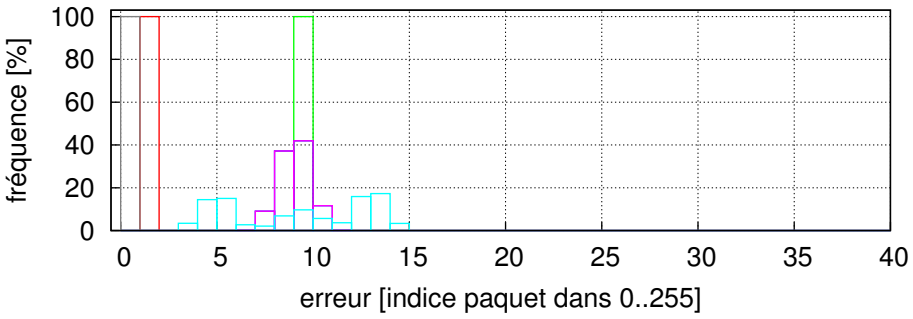

FIGURE 5 – Distribution de l'erreur mathématique pour différents scénarios de fautes (attention seuls les 40 premiers paquets d'erreur sont représentés en abscisse sur les 256 mesurés, au-dessus ils sont tous à 0).

Globalement, on remarque que l'erreur du produit fauté $\widetilde{p}$ se limite à $\frac{15}{255}$ du maximum possible (ici 2^{32}) pour les scénarios simples considérés. En lui-même, ce test n'a pas grand intérêt, il montre juste la vitesse de l'émulation car cette courbe est obtenue en quelques minutes. En effet, à environ 100 MHz, le test de 2^{32} valeurs prend seulement $2 \times 4294967296/(100 \cdot 10^6) \approx 90\,\text{s}$ par bloc d'émulation (le facteur 2 vient de la gestion non pipelinée de la RAM de résultats que nous devons encore améliorer). En pratique, tester des centaines de points d'injection est possible et ne nécessite que quelques minutes du fait du parallélisme important disponible (plusieurs blocs par FPGA et plusieurs cartes FPGA).
Enfin, en table 2, nous présentons les résultats d'implantation FPGA et d'émulation de fautes du multiplieur avec différents codes résidus ($m \in \{3, 7, 15\}$). Le taux de détection mesuré lors de l'émulation de valeurs aléatoires est conforme à la valeur théorique de détection de $1 - \frac{1}{m}$.

modulo m	surface [LUT]	délai [ns]	taux détection moyen mesuré [%]
3	525	10.4	66.7
7	526	10.1	85.6
15	553	13.0	93.3

TABLE 2 – Résultats d'implantation FPGA et d'émulation de fautes sur un multiplieur 16×16 bits avec détection par code résidu modulo m.

5. Conclusion

Nous avons présenté les premiers développements d'une plateforme matérielle-logicielle d'étude expérimentale pour des techniques de détection et de tolérance aux fautes dans des opérateurs arithmétiques avec différents paramètres, algorithmes, représentations internes et optimisations. Tout comme l'état de l'art en émulation de fautes sur FPGA, nous obtenons des facteurs d'accélération importants, ×6880 dans notre cas, par rapport à des simulations logicielles. Ceci nous permet d'étudier des propriétés mathématiques requérant de très nombreuses évaluations impossibles à faire en logiciel.
Ce travail est encore en développement, il nous reste beaucoup à faire sur la plateforme et son exploitation, p. ex. explorer l'espace des paramètres pour optimiser les performances pour arriver à celles attendues en table 3 ; ajouter des scénarios de fautes plus complexes comme la prise en compte du chemin critique ; et enfin distribuer les logiciels de la plateforme sous *licence libre* pour susciter des collaborations.

par seconde	par heure	par jour
$100 \cdot 10^6 \times 100 \approx 2^{33}$	$\approx 2^{45}$	$\approx 2^{49}$

TABLE 3 – Capacités d'émulation de fautes/d'entrées attendues de la plateforme équipée de 100 blocs d'émulation à 100 MHZ répartis sur les différents FPGA.

Remerciements

Ce travail a été financé en partie par les projets ARDyT (ANR-11-INSE-15, `http://ardyt.irisa.fr/`) et Reliasic (Labex CominLabs, `http://www.reliasic.cominlabs.ueb.eu/`).

Bibliographie

1. Antoni (L.), Leveugle (R.) et Feher (B.). – Using run-time reconfiguration for fault injection in hardware prototypes. – In *Proc. International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 405–413, Yamanashi, Japan, octobre 2000. IEEE.
2. Benso (A.) et Prinetto (P.) (édité par). – *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. – Springer, 2003, *Frontiers in Electronic Testing*, volume 23.
3. Bigou (K.), Chabrier (T.) et Tisserand (A.). – Opérateur matériel de tests de divisibilité par des petites constantes sur de très grands entiers. – In *Actes Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS)*, Grenoble, France, janvier 2013.
4. Boncalo (O.), Amaricai (A.), Spagnol (C.) et Popovici (E.). – Cost effective FPGA probabilistic fault emulation. – In *Proc 32nd NORCHIP Conference*, pp. 1–4. IEEE, octobre 2014.
5. Cheng (K.-T.), Huang (S.-Y.) et Dai (W.-J.). – Fault emulation : a new approach to fault

grading. – In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pp. 681–686, San Jose, CA, USA, novembre 1995. IEEE/ACM.

6. de Andres (D.), Ruiz (J.), Gil (D.) et Gil (P.). – Fault emulation for dependability evaluation of VLSI systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, n. 4, avril 2008, pp. 422–431.
7. Dubrova (E.). – *Fault-Tolerant Design*. – Springer, 2013.
8. Ellervee (P.), Raik (J.), Tammemae (K.) et Ubar (R.-J.). – FPGA-based fault emulation of synchronous sequential circuits. *IET Computers & Digital Techniques*, vol. 1, n. 2, mars 2007, pp. 70–76.
9. Ercegovac (M. D.) et Lang (T.). – *Digital Arithmetic*. – Morgan Kaufmann, 2003.
10. Flynn (M. J.) et Oberman (S. F.). – *Advanced Computer Arithmetic Design*. – Wiley-Interscience, 2001.
11. Garcia-Valderas (M.), Lopez-Ongil (C.), Portela-Garcia (M.) et Entrena-Arrontes (L.). – Transient fault emulation of hardened circuits in FPGA platforms. – In *Proc. International On-Line Testing Symposium (IOLTS)*, pp. 109–114, Como, Italy, juillet 2004. IEEE.
12. Gulati (K.) et Khatri (S.). – Towards acceleration of fault simulation using graphics processing units. – In *Proc. 45th Annual Design Automation Conference (DAC)*, pp. 822 – 827, Anaheim, CA, USA, juin 2008. ACM/IEEE.
13. Hong (J.-H.), Hwang (S.-A.) et Wu (C.-W.). – An FPGA-based hardware emulator for fast fault emulation. – In *Proc. 39th Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 345–348, Ames, IA, USA, août 1996. IEEE.
14. Koren (I.) et Krishna (C. M.). – *Fault-Tolerant Systems*. – San Francisco, CA, USA, Morgan-Kaufman, 2007.
15. Legat (U.), Biasizzo (A.) et Novak (F.). – Automated SEU fault emulation using partial FPGA reconfiguration. – In *Proc. 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pp. 24–27, Vienna, Austria, avril 2010. IEEE.
16. Lopez-Ongil (C.), Garcia-Valderas (M.), Portela-Garcia (M.) et Entrena-Arrontes (L.). – An autonomous FPGA-based emulation system for fast fault tolerant evaluation. – In *Proc. 15th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 397–402. IEEE, août 2005.
17. Mueller-Thuns (R.), Saab (D.), Damiano (R.) et Abraham (J.). – VLSI logic and fault simulation on general-purpose parallel computers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, n. 3, mars 1993, pp. 446–460.
18. Niermann (T.), Cheng (W.) et Patel (J.). – PROOFS : a fast, memory-efficient sequential circuit fault simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, n. 2, février 1992, pp. 198–207.
19. Pamula (D.) et Tisserand (A.). – GF(2^m) finite-field multipliers with reduced activity variations. – In *4th International Workshop on the Arithmetic of Finite Fields*, *LNCS*, volume 7369, pp. 152–167, Bochum, Germany, juillet 2012. Springer.
20. Tisserand (A.). – Fast and accurate activity evaluation in multipliers. – In *Proc. 42nd Asilomar Conference on Signals, Systems and Computers*, pp. 757–761, Pacific Grove, California, U.S.A., octobre 2008. IEEE.
21. Villalta (I.), Bidarte (U.), Santos (G.), Matallana (A.) et Jimenez (J.). – Fault injection system for SEU simulation in Zynq SoCs. – In *Proc. Conference on Design of Circuits and Integrated Circuits (DCIS)*, pp. 1–6, Madrid, Spain, novembre 2014.
22. Zimmermann (R.). – VHDL library of arithmetic units. – In *Proc. 1st Forum on Design Languages (FDL)*, pp. 267–272, Lausanne, Switzerland, septembre 1998. – `http://www.iis.ee.ethz.ch/~zimmi/arith_lib`.