ROB307 Drone Livreur

1nd Kai Zhang
ingénieur logiciel
ENSTA Paris
Palaiseau, France
kai.zhang.2021.54@ensta-paris.fr

2st Icare Sakr
Ingénieur automatization
ENSTA Paris
Palaiseau, France
icare.sakr@ensta-paris.fr

3rd David Velasquez Ospina
Ingénieur intégration MPSOC
ENSTA Paris
Palaiseau, France
david.velasquez-ospina@ensta-paris.fr

4rd Ricardo Rico Uribe
Ingénieur intégration MPSOC
ENSTA Paris
Palaiseau, France
ricardo.rico-uribe@ensta-paris.fr

5rd Mengyu Pan
Ingénieur NOC
ENSTA Paris
Palaiseau, France
mengyu.pan.2021@ensta-paris.fr

6rd Yan Chen
Ingénieur NOC
ENSTA Paris
Palaiseau, France
yan.chen.2021@ensta-paris.fr

I. Abstract

This is the MPSOC project report for ROB307. The project aims to build a heterogeneous multiprocessor architecture on a chip and use it in a real world application. The project was divided in 4 development groups, a group in charge of a communication network for resource sharing, in our case memory, another group (MPSOC development) took this network and added 3 different processors, one in chip and two other simulated by hardware (ARM and Microblaze respectively) with a multiprocessor design, the next step was the code implementation in each processor, the coded algorithms were chosen for their use in the case of our application, a delivery drone, which requires image recognition to find its destination and landing site. In adition an automated research for the optimal design was done in ten computers of the University.

II. Introduction

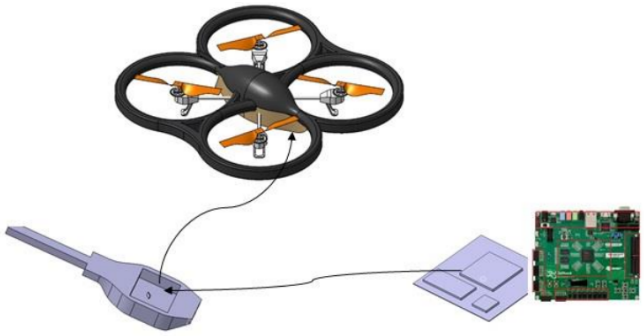

Fig. 1. Le drone livreur opérera sur le campus de l'ENSTA

Le but de ce project est de concevoir un système embarqué multicoeur hétérogène pour drone livreur. Le drone peut livrer les colis sur le campus de l'ENSTA Paris. Les drones doivent être légers, peu gourmands en énergie, flexibles, avoir une longue portée et être multifonctionnels. Pour atteindre cet objectif, la technologie MPSOC est utilisée dans ce projet. Ce MPSOC hétérogène sera implémentée sur un FPGA Zynq sur une carte zedboard placée à l'intérieur du drone. Ce projet est dévissé 4 parties :

1) Automation Optimisation
2) Logiciel embarquée applications
3) Intégration MPSOC
4) NOC conception

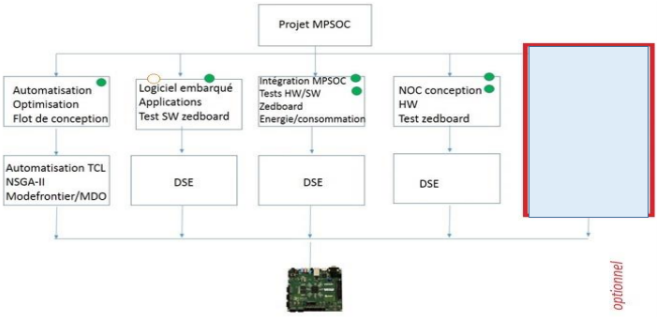

Fig. 2. Les taches dévissées

Premièrement, nous définissons les fonctions nécessaire qui réalisent livreur de drone. Par exemple, la perception, le traitement d'information et algorithmes de contrôle, etc. Puis, les fonctions sont implémentés sur MPSOC de drone. Et la communication entre des multi-processeurs et les autres sources (FPGA, BRam,etc) ou épuisements(Sensors,Camera, Lidar) est réalisée par NOC (Network on Chip). Finalement, automation du flot de conception est fait pour répondre aux mieux au objectif a atteindre.

III. LOGICIEL EMBARQUE

Pour construire un drone pour livrer les colis aux utilisateurs, nous devons implémenter trois modules, la perception, le traitement et le contrôle. Dans un premier temps, il a besoin de percevoir les informations de l'environnement, puis à partir des données obtenues à partir des capteurs, il extrait les informations utiles et les transforme en informations de

contrôle. Enfin, en fonction des informations de contrôle, le drone peut voyager dans l'environnement.

Dans cette section, nous implémentons les trois modules dans le système embarqué et l'optimisons en tant qu'application en temps réel.

A. Structure des fonctions

Nous avons développé cinq algorithmes pour réaliser la perception, le traitement et le contrôle du drone. Comme le montre la figure 3, l'explication détaillée peut être trouvée comme suit.

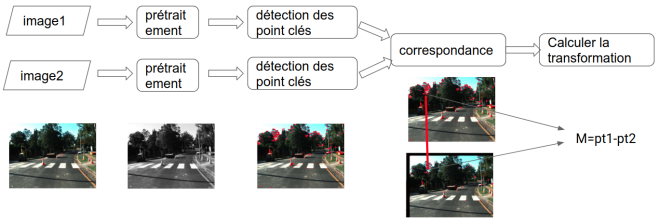

Fig. 3. Structure des fonctions

1) Lecture d'image. La lecture d'image signifie charger les données des capteurs, ce qui comprend la communication entre les processeurs et les capteurs, remodeler les données et les convertir en processeurs. Dans cette expérience, nous générons deux images avec mouvement en raison du manque de capteurs pour la simulation.

2) Prétraitement. Le prétraitement d'image vise à modifier l'éclairage des images, comme la normalisation de l'histogramme, la conversion de l'image RVB en image grise, le redimensionnement de l'image, etc. Dans notre expérience, nous utilisons l'opération de conversion pour obtenir l'image grise afin d'économiser la mémoire pour les données et réduisez le coût de calcul.

3) Détecter les points clés. Le point clé comprend divers types de points, comme les coins, les points d'isolement, etc. C'est toujours le point évident dans le contexte local et existe à l'endroit où les gradients changent considérablement. Dans cette expérience, nous implémentons l'algorithme de détection de point clé Harris[2] à partir de zéro car le système embarqué ne contient pas de bibliothèque correspondante.

4) Correspondance. Faire correspondre signifie trouver les mêmes points projetés dans différentes images. Il compare toujours le contexte local des points dans deux images et trouve les paires avec le moins de différence. Dans notre expérience, nous utilisons la différence d'erreur quadratique moyenne pour comparer les points clés détectés dans différentes images. Pour le rendre plus robuste, nous adoptons la comparaison entre deux fenêtres autour des points clés.

5) Calculer la transformation. La transformation comprend généralement la translation et la rotation. Cependant, pour le drone, le mouvement ne peut s'exprimer que par la traduction. Dans cette expérience, nous obtenons la traduction en calculant le mouvement des points clés appariés.

B. Détails de l'expérience

En raison du manque de capteurs, nous avons proposé quelques hypothèses:

1) La largeur de l'image acquise est de 128 pixels et la hauteur de 128 pixels avec 3 canaux.

2) La distance de sécurité du drone est de 0,1 m.

3) la vitesse maximale du drone est de 1 m / s (3,6 km / h). Compte tenu de l'hypothèse 2, le temps d'exécution maximum du processeur est de 100 ms pour promettre la sécurité.

4) La fréquence de la caméra est de 30 HZ, donc pour traiter l'image en temps réel, le temps d'exécution de la contrainte est de 33 ms.

Les processeurs que nous avons utilisés dans les expériences comprennent le bras et le MicroBlaze de la carte Xilinx ZYNQ7. Le logiciel de simulation et d'implémentation est Vivado 2019.1. La conception de bloc correspondante est illustrée sur la figure 4.

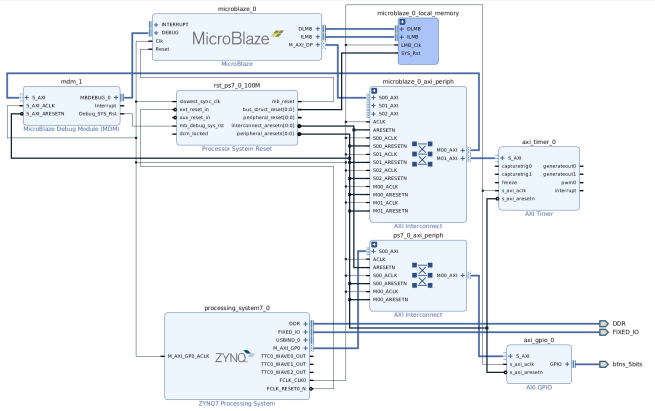

Fig. 4. Le block design dans ce test.

C. Résultats

Nous testons nos algorithmes dans ARM et MicroBlaze respectivement. Pour chaque étape, nous comptons également le temps d'exécution séparément. De plus, pour améliorer les performances, nous utilisons également la méthode d'optimisation pour améliorer l'efficacité.

Comme le montre le tableau [], nous calculons le temps d'exécution de chaque étape dans deux processeurs. En raison de la limitation de la RAM, nous recadrons l'image originale en plusieurs échantillons afin qu'elle puisse la traiter. Pour le processeur ARM, nous divisons l'image en 16 échantillons dont la largeur est de 32 pixels et la hauteur est de 32 pixels. Alors que dans le processeur MicroBlaze, nous le découpons en 63 échantillons et chaque échantillon est de 16 x 16. À partir du temps d'exécution illustré dans le tableau I, nous pouvons constater que le processeur ARM fonctionne plus efficacement que le processeur MicroBlaze. Du fait que la contrainte de temps est de 33ms, la détection des points clés ne satisfait pas la contrainte de temps pour le processeur ARM donc avant l'optimisation, il ne peut pas traiter en temps réel. Le temps de fonctionnement total est de 97 ms, ce

qui est inférieur à 100 ms, donc l'exécution des algorithmes dans le processeur ARM peut toujours permettre la sécurité. Cependant, l'opération dans Microblaze ne peut pas promettre la distance de sécurité.

TABLE I
PERFORMANCE SANS OPTIMISATION

Fonction	Description	temps(ms)	ARM(ms)	Microblaze(ms)
F1	lecture d'image	33	3.5376	146.048
F2	prétraitement	33	4.4096	3.968
F3	détection	33	89.8736	54.976
F4	correspondance	33	0.0928	119.232
F5	Transformer	10	0.0032	1.472
sum			97.9168	325.696

Le tableau II montre les performances après optimisation. Nous utilisons l'optimisation O3 pour la compilation. Après optimisation, les algorithmes nécessitent moins de mémoire, nous avons donc divisé l'image originale en 4 échantillons dans le processeur ARM. Chacun d'eux a 64 pixels de largeur et 64 pixels de hauteur. En outre, la grande amélioration peut être vue pour le temps de fonctionnement. Toutes les étapes du processeur ARM peuvent être exécutées en temps réel et garder la distance de sécurité. Pour Micoblaze, bien qu'avec l'amélioration, il ne peut toujours pas satisfaire la contrainte de sécurité et la contrainte temps réel.

TABLE II
PERFORMANCE AVEC OPTIMISATION

Fonction	Description	temps (ms)	ARM(ms)	Microblaze(ms)
F1	lecture d'image	33	0.2124	12.4992
F2	prétraitement	33	0.6388	3.315072
F3	détection	33	7.7896	54.976
F4	correspondance	33	0.0012	120.256
F5	Transformer	10	0.00024	1.472
sum			8.6422	192.518272

IV. MPSOC

Dans cette section, on cherche à expliquer le processus suivi pour la conception et la mise en œuvre d'un bloc design qui permet l'exécution parallèle de plusieurs processus, la mise en œuvre de ces processus dans SDK et plusieurs optimisations qui permettent d'améliorer les performances de conception. On mentionnera également certains problèmes constatés, et les améliorations possibles qui n'ont pas été réalisées mais qui pourraient être intéressantes à analyser dans le futur.

L'appareil sur lequel tous les tests seront effectués est la ZedBoard: Zynq Evaluation and Development Kit (xc7020clg484-1) qui possède une puce ARM Cortex-A9 dual-core basé sur 1 Xilinx Zynq-7000 SoC. La puce permet d'exécuter deux programmes en parallèle, mais ce modèle n'est pas optimal car toutes les autres ressources disponibles ne sont pas utilisés. Dans le cas où l'on n'a que 2 programmes à exécuter en parallèle, on peut choisir de mettre en place un accélérateur de matériaux. Cela permettrait d'utiliser les ressources disponibles pour améliorer les performances des programmes exécutés sur la puce. Cependant, si l'on a besoin de mettre en œuvre plus de 2 programmes, cette solution serait courte, dans ce cas on pourrait mettre en œuvre les microblazes, qui utilisent les ressources du FPGA pour simuler le comportement d'un processeur.

Comme il est de notre intérêt de mettre en parallèle plusieurs fonctions du drone, nous cherchons à mettre en œuvre l'approche avec les microblazes, cependant, nous n'excluons pas la mise en place d'un accélérateur pour certaines fonctions qui pourraient être plus exigeantes. Nous avons d'abord conçu un bloc design qui permet l'implémentation du processeur et d'une microblaze, puis nous avons reproduit la structure pour implémenter 2 microblazes, et le dernier design en a 4. Les tests d'optimisation ont été réalisés avec la conception de 2 microblazes.

A. Bloc design avec 1 microblaze

La première conception réalisée tente d'exécuter un programme sur le processeur ARM et un autre sur la microblaze, pour cela différentes connexions sont nécessaires qui seront décrites ci-dessous. Le microblaze, comme mentionné ci-dessus, est un processeur factice qui exécute une instruction à chaque cycle d'horloge. Ce processeur peut être optimisé, et certaines de ces optimisations sont abordées dans la section IV-D.

Pour le microblaze, on a activé le débogage par UART qui offre des capacités de débogage avancées comme les points d'arrêt matériel, les points de surveillance de la mémoire de lecture/écriture et une meilleure visibilité des processeurs MicroBlaze. Ensuite, on a mis en place deux AXI_Interconnect , l'un pour connecter les caches microblaze au processeur Zynq et l'autre pour connecter l'interface de données périphérique au timer (nécessaire à l'implémentation du code), au débogueur et à la mémoire partagée dont on parlera après. Il est aussi connecté aux blocs de mémoire locale pour optimiser le traitement de variables locaux et il est connecté au même reset et à la même horloge que le Zynq pour assurer la synchronisation.

Pour la communication entre les différents processus, des zones de mémoire partagée ont été établies, dans lesquelles chaque processus peut écrire et lire des variables écrites par les autres. Ces espaces sont définis par plusieurs blocs BRAM, et sont accessibles par une AXI_crossbar. Les connexions du microblaze sont montrés dans le tableu 6 et le bloc diagrammes se trouve dans la figure 5. Les ressources FPGA utilisées dans la conception sont indiquées dans le tableau III. Il convient de noter que la ressource la plus utilisée est la mémoire BRAM, et cependant, celles-ci n'occupent même pas 13% de la capacité totale de la carte, ce qui indique qu'il est possible de mettre en œuvre un plus grand nombre de microblazes.

Ressource	Utilisée	Disponible	% Utilisée
LUT	5100	53200	9.59
LUTRAM	309	17400	1.78
FF	5194	106400	4.88
BRAM	18	140	12.86
BUFG	2	32	6.25

TABLE III
TABLE DE RESSOURCES UTILISÉS POUR LE BLOC DESIGN AVEC 1 MICROBLAZE

Ressource	Utilisée	Disponible	% Utilisée
LUT	7771	53200	14.61
LUTRAM	568	17400	3.26
FF	7683	106400	7.22
BRAM	26	140	18.57
BUFG	2	32	6.25

TABLE IV
TABLE DE RESSOURCES UTILISÉS POUR LE BLOC DESIGN AVEC 2 MICROBLAZES

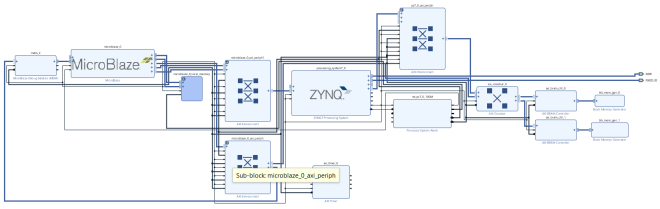

Fig. 5. Bloc design avec 1 microblaze

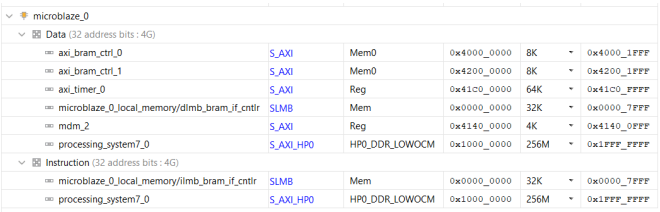

Fig. 6. Connexion du microblaze

B. Bloc design avec 2 microblazes

Lorsque la première conception fonctionnait bien, on est passé à l'intégration d'une seconde microblamme. Lors de la première conception, on a réalisé que les IP nécessaires à la mise en œuvre d'une microbase sont le débogueur, la mémoire locale, une timer et deux AXI_interconnect pour effectuer les connexions. Une fois que ces IP sont connectées, les adresses de mémoire doivent être vérifiées pour s'assurer que le microbloc a accès à la mémoire partagée. Une fois que cela est fait, la conception peut être validée pour la mise en œuvre dans le SDK. Le bloc design est montré dans la figure 7

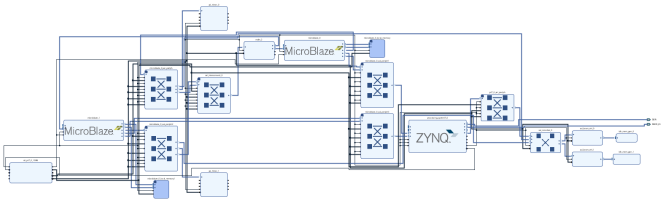

Fig. 7. Bloc design avec 2 microblazes

Les ressources du FPGA utilisées dans la conception sont indiquées dans le tableau IV. On peut voir que le montant des ressources utilisées a légèrement augmenté, sauf pour le BUFG. Nous pouvons également constater que la mémoire BRAM est toujours la ressource la plus utilisée, mais il y a encore de la place pour d'autres microbags ou d'autres IP

C. Bloc design avec 4 microblazes

Un design final a été conçu avec 4 microblazes pour vérifier la reproductibilité du design, ainsi que la croissance linéaire des ressources utilisées. Le même processus a été suivi que pour le cas précédent, et la conception est illustrée dans la figure 8. Un facteur à souligner est qu'un seul débogueur est utilisé pour les 4 microblazes, et qu'une AXI_Interconnect est utilisée pour faire varier l'entrée du débogueur selon les besoins.

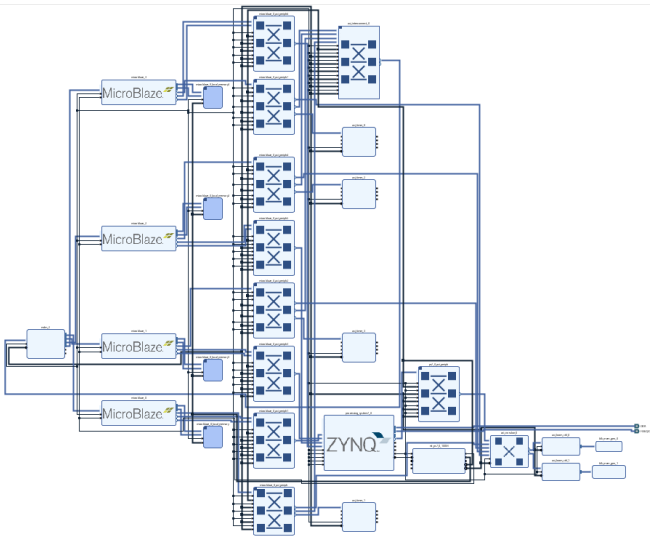

Fig. 8. Bloc design avec 4 microblazes

Dans le tableau V, on peut voir le comportement linéaire des ressources du FPGA lorsque le nombre de microblazes dans la conception est augmenté. On peut également constater que ces derniers ne consomment pas beaucoup d'espace, et si un accélérateur matériel est mis en œuvre pour exécuter le traitement des images par exemple, les performances du système pourraient être considérablement améliorées

D. Optimisations réalisées

Les optimisations que nous recherchions nous permettent d'ajouter options de calcul ou la manipulation mémoire avec hardware et que ceux-ci ne doivent pas être simulés dans le logiciel, pour ces optimisations nous avons utilisé la conception avec 2 Microblaze et nous avons analysé la consommation électrique et la quantité de ressources utilisées.

Ressource	Utilisée	Disponible	% Utilisée
LUT	15221	53200	28.61
LUTRAM	1096	17400	6.30
FF	14565	106400	13.69
BRAM	42	140	30.00
BUFG	2	32	6.25

TABLE V
RESSOURCES UTILISÉS POUR LE BLOC DESIGN AVEC 4 MICROBLAZES

Les optimisations analysées ont été :

- BTC (Branch Target Cache, Prediction)
- MMU (Memory Management Unit for Cache)
- Barrel Shifter (décalage binaire)
- FPU (unité à virgule flottante)
- Diviseur entier
- Multiplicateur entier

Nous avons également essayé de supprimer les blocs ICache et DCache, mais cela pose des problèmes car cela éliminerait le moyen de communication avec le monde extérieur du Microblaze, nous avons trouvé intéressant que DCache est celui qui utilise le plus de ressources, même si les deux mémoires ont la même quantité de blocs BRAM. Vous pouvez voir dans les images supplémentaires à ce rapport les différentes configurations que nous obtenons lors de la mise en œuvre de chaque optimisation, vous pouvez apprécier comment la conception implémenté change lorsque nous ajoutons ou supprimons différents blocs, ceci est observé avec l'apparition ou la disparition des blocs bleus clairs. Comme prévu, les optimisations de mémoire ajoutent un BRAM supplémentaire pour accomplir leur tâche, et nous avons constaté que toutes les optimisations utilisent la même quantité d'énergie (1,69W), sauf par MMU, ce qui augmente la consommation jusqu'à 1,71W. Une autre particularité à souligner est l'ajout de blocs DSP par FPU et Multiplier.

Toutes les optimisations proposées ont été validées, synthétisées et mises en œuvre, mais la BTC a été la seule à produire une erreur à la fin parce qu'elle ne respectait pas de contraints de timing.

Comme mentionné précédemment, cette recherche a été faite pour voir la comparaison des ressources et de l'énergie utilisées par chacun, une approche plus adéquate serait de voir les exigences de chaque code et d'appliquer les optimisations nécessaires au Microblaze chargé de l'exécuter. Cela n'a pas été fait en raison de la nature de la distribution de notre projet, du parallélisme dans le développement de chaque étape et des contraintes de temps.

E. Implementation dans SDK

Nous avons effectué un test au sein de SDK pour vérifier que notre diagramme était réalisable dans le monde réel. Pour cela, nous avons utilisé la conception avec deux Microblaze et un ARM, il est bon de rappeler que le processeur ARM a deux cœurs, ce qui nous donne un total de quatre processeurs indépendants. Pour chaque processeur, nous avons effectué une tâche infinie et avons réussi à prouver que les quatre processeurs fonctionnaient en même temps. Nous n'avons pas réussi à tester le parallélisme en raison des limites de temps, mais nous considérons qu'il est réalisable puisque les 4 processeurs partagent 2 espaces mémoire, de sorte que nous pouvons réaliser la communication de données entre eux en écrivant et en lisant dans des espaces mémoire prédéterminés.

F. Améliorations Futurs

Après avoir terminé la conception du MPSOC, nous avons trouvé des améliorations possibles ou des aspects que nous fassions différemment au futur.

- En utilisant Microblaze, nous laissons de nombreuses ressources disponibles sur la carte, ce qui nous permettrait d'ajouter d'autres processeurs Microblaze au cas où nous aurions besoin de plus de parallélisme ou d'ajouter un accélérateur de matériaux pour une tâche quelconque qui le nécessite.
- La communication entre les processeurs n'a pas été vérifiée pour confirmer qu'ils partagent correctement l'espace mémoire alloué, en plus, un système doit être mis en place pour assurer la simultanéité des données (atomicité) afin de prévenir la corruption des données lorsqu'un processus tente de lire un espace mémoire qui est en cours d'écriture, ou qu'une donnée utilisée par un processus est toujours mise à jour.
- Pour parvenir à une mise en œuvre et un développement corrects, il faut travailler en étroite collaboration avec les autres membres, car nous ne connaissons pas l'état d'avancement et le développement de la partie du Logiciel Embarqué, nous ne savions pas que des optimisations ou des modifications de conception étaient nécessaires. En outre, nous manquions de communication avec la partie NOC, puisque leur développement était séparé du nôtre, nous sommes partis d'une conception différente et celle qu'ils avaient développée n'a pas été mise en pratique, en fin de compte nous avons eu un peu de communication avec nos collègues mais nous n'avons jamais déterminé si le parallélisme ou la communication étaient nécessaires, également la partie automatisation est partie de notre conception mais les solutions d'optimisation trouvées n'ont pas été appliquées.

V. NOC

NoC(Network On Chip) est une nouvelle méthode de communication pour SoC (System On Chip). Les performances de la NoC sont nettement meilleures que celles des systèmes traditionnels basés sur les bus. Les systèmes basés sur la NoC sont mieux adaptés aux mécanismes d'horloge globalement asynchrones et localement synchrones utilisés dans les futures conceptions complexes de SoC multicœurs.

Le but de ce projet est de permettre la livraison de drones. Les drones doivent être légers, compacts, réactifs et avoir des fonctions multiples. Ces fonctions doivent fonctionner en harmonie afin de réaliser parfaitement la mission. Le poids excessif et la conception redondante du drone réduiront ses performances. Par conséquent, la conception de la NOC est un élément nécessaire pour mener à bien ce projet.

Dans cette section, on utilise deux type de Noc. Ils sont Noc 3×3 et Noc 5×5. On crée et simule les deux genres de Noc et analyse leur performances et fonctions avec Vivado Xilinx sur Zedboard. Ensuit, on combine Noc 3×3 et Noc 5×5 avec PS(ZYNQ). La fonction est de transférer les data du Ps ou IP aux BRam. On simule et teste les fonction avec Vivado et avec SDK sur ZedBoard.

A. Protocoles et Communication du NOC

Dans cette partie, le protocole principale est AXI4. AXI4 est l'interface extensible avancée dans l'architecture de bus de micro-contrôleur avancée de ARM. Généralement, il y a AXI4-Lite pour accéder l'adresse et AXI4-Stream pour transfert les plusieurs données au même temps.

Pour faire un réseau il faut implémenter un crossbar qui est un commutateur avec plusieurs entrées et sorties. Sa fonction est d'inter-connecter ces entrées et sorties avec son configuration des commutateurs comme une matrice.

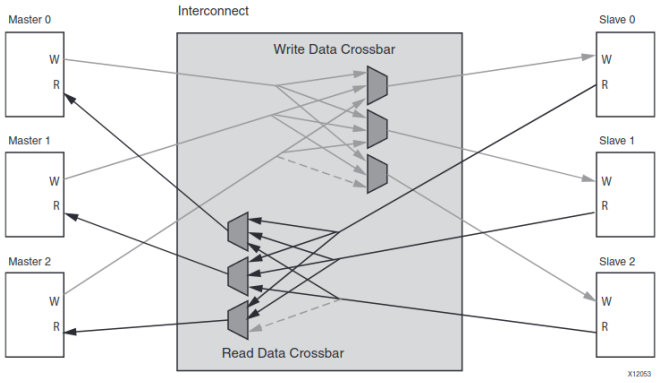

Fig. 9. Sparse Crossbar Write and Read Data Pathways

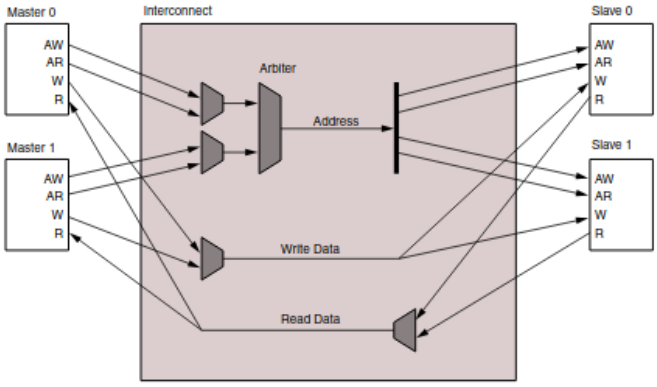

Fig. 10. Shared Access Mode

AXI Crossbar a deux type d'architecture de l'interconnexion éligible - Crossbar mode et Shared Access Mode[4]. Comme indiqué à la Figure 9 et à la Figure 10. Crossbar mode peut optimiser la performance par adresse partagée, architecture de crossbar de multi-donnée, des crossbar voies parallèles pour les canaux d'écriture de données et de lecture de données et des crossbar chemins de données clairsemés selon la carte de connectivité configurée, etc. Shared Access Mode peut optimiser la zone par des données d'écriture partagées, des données de lecture partagées, des chemins d'adresse uniques partagés et minimisation de l'utilisation des ressources, etc.

B. Design de NOC

Au début, on fait les design de bloques des NOC 3×3 et 5×5 dans la figure 11 et 12. Puis, on valide ces design de bloques et fait les simulation (Latence et Bande Passante) par contraints.

Noc 3×3 ont 3 IPs(myipAxi4) qu'on crée, une AXI crossbar, 3 AXI BRam Contoller et 3 Block Memory Generator. Axi crossbar est comme une bridge capable à communiquer avec mes IPs et AXI BRam Controller. Puis, la fonction d'AXI BRam Controller est de contrôler la lecture et le stockage des Block Memory Generator. Noc 5×5 a même fonction avec Noc 3×3. La différence entre les deux type de Noc est le nombre de Master et de Slave d'AXI crossbar.

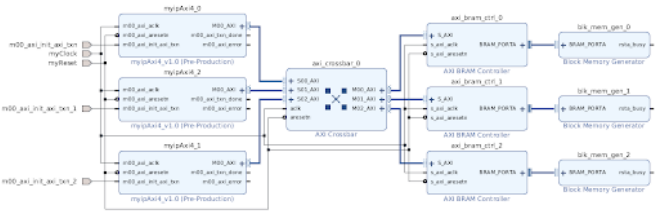

Fig. 11. Diagram design de NOC 3×3

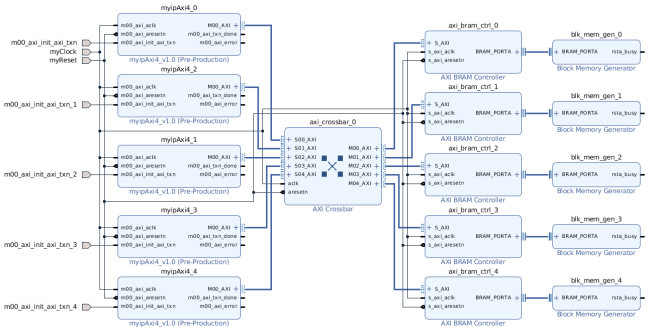

Fig. 12. Diagram design de NOC 5×5

Ensuite, on ajoute le ZYNQ board dans les diagrammes et les nouveaux diagrammes sont représenté dans la figure 13 et 14. Pour ces nouveaux diagrammes, à parti de validation du diagramme et simulation, on fait aussi le test mémoire dans le SDK et les résultats sont représentés dans la figure 15 et 16. On fait les tests mémoires des 8 bits, 16 bits et 32 bits et tous les tests sont réussis.

On ne seulement ajoute ZYNQ, mais aussi ajoute une AXI Inteconnect dans les diagrammes. AXI Inteconnect connecte plusieurs dispositifs AXI maîtres (AXI BRam Controller) à un dispositifs esclaves (ZYNQ). Dans ces diagrammes, chaque Block Memory Generator sont contrôlées par deux AXI BRam Controller. Une connecte AXI Internnect, l'autre connecte AXI crossbar.

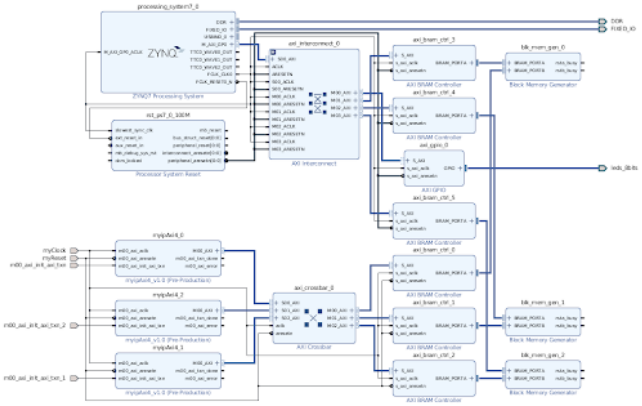

Fig. 13. Diagram design de NOC ZYNQ 3×3

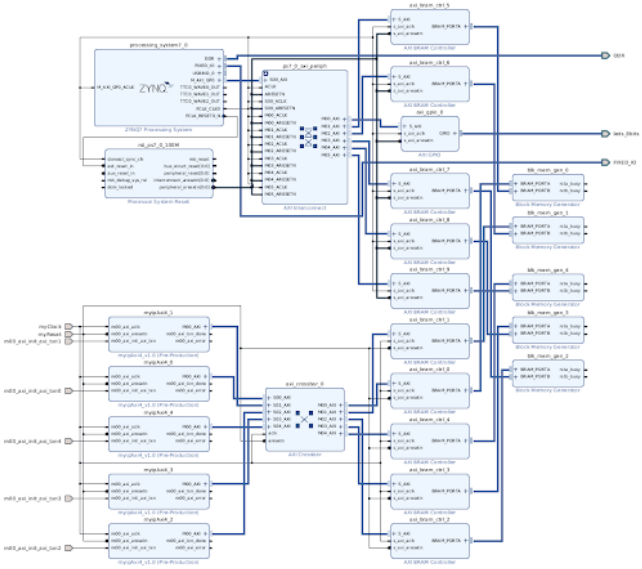

Fig. 14. Diagram design de NOC ZYNQ 5×5

C. Simulation et Test

Il y a pas mal de ressource sur ZedBoard, qui est relative à l'efficace de computation. On peut voir que les nombres de ressource total de LUT, LUTRAM, FF, BRAM, IO et BUFG sont respectivement 53200, 17400, 106400, 140, 200 et 32 sur le talaux VI et le tablaux VII. Puis, le taux d'utilisation de ressource sur crossbar Zynq 5×5 est extérieur à cela sur crossbar 5×5. En autre termes, Zynq peut augmenter le taux d'utilisation de ressource de ZedBoard.

Resource	utilization	available	utilization %
LUT	1893	53200	3.56
LUTRAM	26	17400	0.15
FF	1787	106400	1.68
BRAM	6	140	4.29
IO	5	200	2.50
BUFG	1	32	3.13

TABLE VI
RESSOURCES UTILISÉS POUR LE BLOC DESIGN AVEC CROSSBAR 5×5

Resource	utilization	available	utilization %
LUT	5256	53200	9.88
LUTRAM	161	17400	0.93
FF	4870	106400	4.58
BRAM	5	140	3.57
IO	15	200	7.50
BUFG	2	32	6.25

TABLE VII
RESSOURCES UTILISÉS POUR LE BLOC DESIGN AVEC CROSSBAR ZYNQ 5×5

La latence est la quantité de temps entre démarrage et travail. Dans le NOC, elle est le délais entre le signal de l'initialisation et la démarrage du travail. On étudie l'influence du type de fonctionnement utilisé pour l'AXI crossbar, du taille de NOC et de Zynq Board. Les latences des deux modes (Crossbar Mode et Shared Access Mode) sont notées. On peut voir qu'il y a des delais entre les signals dans la mode de Shared Access dans les tables VIII et IX. Mais le zynqboard n'influence pas la latence de NOC dans les tables X et XI.

signal	crossbar/ns	shared access/ns
s1	140	120
s2	140	240
s3	140	360

TABLE VIII
LATENCE DE CROSSBAR 3×3

signal	crossbar/ns	shared access/ns
s1	140	120
s2	140	140
s3	140	160
s4	140	180
s5	140	200

TABLE IX
LATENCE DE CROSSBAR 5×5

signal	crossbar/ns
s1	140
s2	140
s3	140

TABLE X
LATENCE DE CROSSBAR 3×3 AVEC ZYNQ BOARD

signal	crossbar/ns
s1	160
s2	160
s3	160
s4	160
s5	160

TABLE XI
LATENCE DE CROSSBAR 5×5 AVEC ZYNQ BOARD

La bande passante est la volume des données qu'on peut envoyer par unité de temps. Dans NOC, on peut la calculer par calculer la durée de data brust. Dans la table XII et XIII, on peut voir que la bande passante diminue avec l'augmentation

du data brust et le size de crossbar n'influence pas cette bande passante.

Largeur Data Burst	Durée(ns)	Bande Passante(MHz)
4	80	12.5000
8	140	7.14286
16	320	3.12500
32	620	1.61290
64	1260	0.79365

TABLE XII
BANDE PASSANTE DE CROSSBAR 3×3

Largeur Data Burst	Durée(ns)	Bande Passante(MHz)
4	80	12.5000
8	160	6.25000
16	300	4.94624
32	640	1.56250
64	1280	0.78125

TABLE XIII
BANDE PASSANTE DE CROSSBAR 5×5

Fig. 15. Test mémoire de NOC ZYNQ 3×3

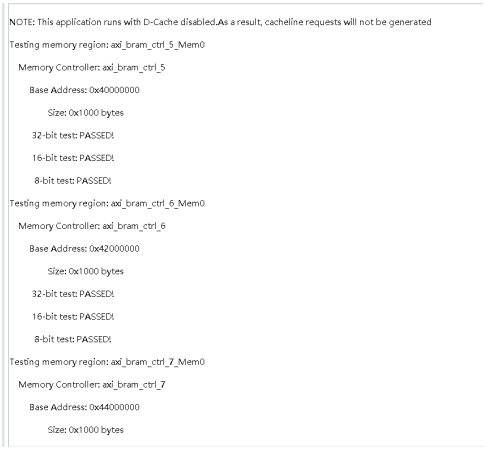

Fig. 16. Test mémoire de NOC ZYNQ 5×5

VI. AUTOMATISATION DU FLOT DE CONCEPTION

A. Problème d'optimisation multi-objectifs

Dans le cadre de l'optimisation de notre design pour répondre aux mieux aux objectifs à atteindre, le problème est NPC (NP-complet) en la taille et la complexité du design. Pour notre application de drone livreur le critère d'optimalité est un terme non déterministe. Il traduit la manière et la capacité à répondre aux fonctionnalités dans les contraintes (limitations) du système. Premièrement, comme l'opération de calcul du flot optique nécessite une exécution en temps réel (3ms) (pour éviter des catastrophes face à des évènements nécessitant une action rapide), un premier objectif qu'il faut optimiser est **la performance en terme de temps d'exécution** de notre design par rapport à la tache demandée. Dans ce cadre bien précis, cela peut être représenté par le fait de respecter une fréquence d'horloge aussi grande que possible sans perte d'informations. D'autre part, dans notre MPSOC, il est intéressant de maximiser le nombre de processeurs parallèles pour effectuer le plus d'opérations possible en parallèle, dans la surface limitée disponible en respect des dimensions de notre drone (en partie). Il s'agit donc d'un objectif de **minimisation de la surface occupée par le design (en termes d'utilisation des ressources (slices))**. Ensuite, comme nous voudrions que le drone vole le plus longtemps possible avec une batterie de charge électrique donnée, nous nous intéressons aussi à **minimiser l'énergie consommée** par notre circuit en fonctionnement. Il s'agit donc d'un problème d'optimisation multi-objectifs où il est impossible de trouver une solution unique (si le problème n'est pas trivial) optimisant chaque objectif d'une manière indépendante (sans influencer sur les autres). Les solutions optimaux constituent un front de Pareto [22] des solutions non dominées.

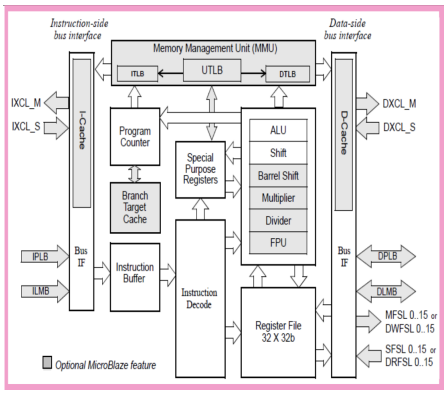

Fig. 17. Architecture du coeur du processeur Xilinx MicroBlaze™ [1]

B. Exploration de l'ensemble des paramètres

Pour résoudre le problème d'optimisation multi-objectifs évoqué dans la partie précédente, il est nécessaire d'explorer l'ensemble des paramètres du projet (design MPSOC, NOC, ...) pour évaluer l'influence de ses paramètres sur les objectifs que l'on a fixé. Si nous explorons notre MPSOC constitué de deux MicroBlazes, et nous nous limitons uniquement aux

paramètres relatifs aux MBs, nous pouvons identifié en partie pour chaque processeur les paramètres suivants:

1) **IC:** Taille de la cache d'instructions (kB), $IC \in \{2^N, N \in [\![6\ ;\ 16]\!]\}$
2) **DC:** Taille de la cache de données (kB), $DC \in \{2^N, N \in [\![6\ ;\ 16]\!]\}$
3) **fpu_state:** Etat de l'unité de calcul flottants qui améliore beaucoup la performance des opérations flottants, mais augmente beaucoup la surface du MB, prenant 3 valeurs $fpu\_state \in \{none = (sansFPU), Basic(+, -, x, :, compare), Advanced(Basic \cup (convert, \surd)\}$
4) **enable_bc:** activation du Barrel shifter, $enable\_bc \in \{0, 1\}$
5) **enable_int_divider:** activation du diviser d'entier, $enable\_int\_divider \in \{0, 1\}$
6) **optimize_area:** option d'optimisation de surface, $optimize\_area \in \{0, 1\}$

Ceci correspond à $11^2 \times 11^2 \times 3^2 \times 2^2 \times 2^2 \times 2^2 = 8,433,216$ configurations possibles du MPSOC et cela pour les deux MBs uniquement. Ainsi si nous devons synthétisé et évaluer la performance de chaque configuration avec le logiciel Vivado, et supposant que chaque synthèse nécessite une heure, l'exploration de toutes les configurations nécessite 962 années! Par suite, il est impossible de faire une exploration exhaustive des configuration, et nous devons appliquer des méthodes de recherche opérationnelles pour l'optimisation multi-objectifs, permettant de n'évaluer que les configurations à grande probabilité de domination comme le NSGA-II qu'on va évoquer dans la suite.

C. Automatisation de l'exploration optimale

1) Fichiers TCL: TCL, ou $ToolCommandLanguage$, est un langage de programmation dynamique. Dans Vivado, il est utilisé afin de fournir un outil de développement entièrement basé sur la console et les fichiers TCL. Un fichier TCL définit complètement les paramètres et configurations d'un projet sur l'interface graphique, ainsi que les toutes les opérations qu'on est capable de faire (simulation, synthèse, génération des rapports,...).

Cette spécificité de Vivado, permet d'automatiser le flot de conception par la génération de projets entièrement définis par leur fichiers TCL et de les exécuter dans la console en mode batch. Nous avons expérimé avec ce language et avons créer des représentation de nos designs MPSOC ainsi qu'une génération automatique des rapports de performances (dans des fichiers textes), à travers les fichiers TCL.

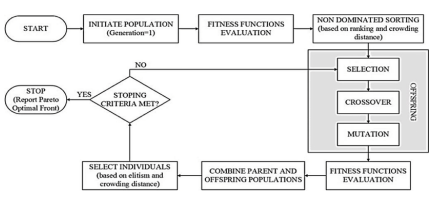

Fig. 18. Architecture de l'algorithme NSGA-II [**nsga2**]

2) Algorithme NSGA-II: L'algorithme NSGA-II permet une exploration stochastique des paramètres, permettant à chaque itération d'explorer uniquement les configurations à grande probabilité d'amélioration / domination. Il est basé sur l'évolution génétique dans la nature. Dans le cadre de notre projet Il consiste à générer une population initiale d'individus caractérisés par leur chromosomes qui est définis par ses paramètres. Chaque individu définit donc une configuration possible du design. Ensuite, pour chaque itération i, l'algorithme applique les opérations de base de la génetique:

1) Selection: selection des K meilleurs individus (non dominés) pour les fonctions obtectifs fixés.
2) Crossover: Effectuer des croisements entre individus dans l'optique où 2 bons individus produirons probablement un individu meilleur par mélange de leur gènes.
3) Mutation: variation aléatoire d'un gène (don paramètre), avec une faible probabilité α, et cela pour éviter de tomber à des minimums locaux lors de l'exploration et tenter de nouvelles configurations dans l'espace des configurations.

Avec ces opérations de base, nous pouvons intuitionner qu'a chaque itération, la probabilité d'amélioration des individus (en terme de domination) est élevée (car descendants de bons individus), ce qui explique l'intérêt de cet algorithme à éliminer une partie de l'ensemble de configurations qui très probablement est inutile à explorer. La condition d'arrêt de cet algorithme est difficile à définir et quantifier, et dépend fortement des fonctions objectifs, contraintes, et paramètres de l'algorithme, mais nous pouvons dire qualitativement qu'on peut s'arrêter lorsqu'on est satisfaits des résultats obtenus. Dans le cas du projet, si nous optons à une population de 40 individus, et 30 itérations, nous devons alors évaluer uniquement 1200 configurations, et si on parallélise l'évaluation des 40 individus (définis par leur fichiers TCL) sur 40 ordinateurs, alors nous passons à 30 heures d'exécution pour obtenir un front de Pareto.

Le NSGA-II est robuste et a démontré sa capacité à traiter des problèmes très complexes d'une manière entièrement automatisée.

D. Application et architecture du processus d'automatisation

Pour concrétiser le problème d'automatisation de l'exploration optimale des configurations, nous avons définis l'architecture de code dans la figure 19

Elle consiste en un programme coeur où tourne l'algorithme NSGA-II, qui est configuré par un fichier de configuration $algoConf$ définissant l'ensemble de ses paramètres et la structure du chromosome choisi (paramètres du design à explorer). Ensuite, l'algorithme génère sa population, et écrit les paramètres de chaque individu dans un fichier $CONFIG-DSE.txt$, qui est utilisé par le programme $DSE-TCL.c$ pour génerer les différents fichiers TCL correspondants à chaque individu, en se basant sur la structure du design à explorer ($design\_ref.tcl$). $DSE-TCL.c$ envoie ensuite chaque fichier TCL à une machine pour exécution en parallèle

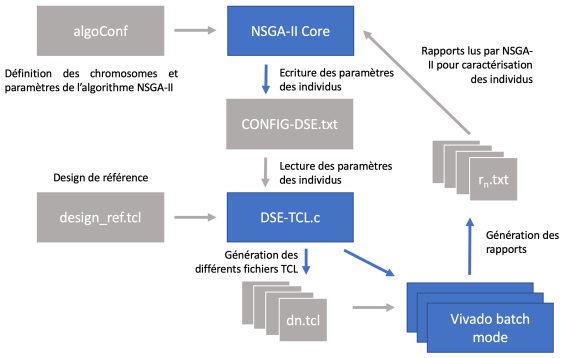

Fig. 19. Notre architecture du processus d'automatisation

avec Vivado en mode batch, et chaque instance générera les rapports de performances de son individu.

E. Essais et résultats

1) Définition des paramètres de l'algorithme: Par contraintes de temps et pour un but de test, nous avons limité les paramètres à explorer aux valeurs des caches d'instructions et de données de nos MBs pour le design à 2 MBs. Le chromosome ainsi définit est donné en figure 20.

IC1	DC1	IC2	DC2	...
xc1 kB	xd1 kB	xc2 kB	xd2 kB	...

Fig. 20. Définition du chromosome

Également, nous avons limité l'ensemble des valeurs possibles des caches à l'ensemble suivant, $\{2^N, N \in [\![9\ ;\ 16]\!]\}$, de cardinalité 2^3 et ceci puisque nos variables (paramètres) sont définis comme binaires dans NSGA-II, et pour pouvoir les codés sur un nombre de bits sans permettre des valeurs de variables hors de la zone des possibilités. (on veut évité aussi d'ajouter des contraintes à cause du problème de random de c...). Finalement, par contrainte de temps, nous nous sommes limités aux fonctions objectifs de minimisation surface, d'énergie, et maximisation des tailles des caches. L'intégration du temps d'exécution ($\sim$ ou fréquence) dans les objectifs n'a pas été faite puisque en partie nous n'avons pas peu aboutir à l'intégration des fonctionnalités du drone en parallèle sur nos designs.

2) Résultats: Nous avons lancé notre programme automatisé sur 10 machines de l'ENSTA en parallèle, et avons réalisé deux essais, les deux avec une taille de population de 10, un premier en fixant 5 itérations et deux fonctions objectifs (Surface (% Silces) et Énergie consommée (W)), et un deuxième avec 10 itérations et 3 fonctions objectifs (ajout d'un critère de maximisation de la taille de l'ensemble des caches des 2 MBs (somme des gènes)) pour donner un sens non trivial aux résultats. Les résultats de la dernière génération sont reportés dans la figure VI-E2. On peut voir par exemple que pour le cas non trivial (à 3 fonctions objectifs) (b), l'individu 9: {8192, 32768, 16384, 32768} utilise une surface très proche de l'individu {2048, 4096, 4096, 8192} même si contient des valeurs beaucoup plus élevés de caches, mais ce traduit par une plus grande consommation d'énergie. Également, l'individu 4: {512,512,1024,512} utilise nécessite une surface plus élevée que 9, (sachant que les valeurs des caches sont beaucoup plus faibles) mais consomme moins d'énergie. On peut voir içi qu'il est impossible d'optimiser les objectifs d'une manière indépendante, et il fat choisir la solution qui nous va le plus dans front de Pareto obtenu.

Cependant ces résultats ont des limitations en terme de fiabilité puisque la simulation n'a pas été faite les fonctions du drone qui s'exécutent dans notre design, et il aurait été intéressant de voir, pour notre application spécifique qui est répartie sur les MBs et le processeur ARM, et relativement aux opérations et fonctions exécutés sur chaque processeur, comment les paramètres influence la performance multi-objective spécifique à notre probème. Mais les résultats de surface sont fiables, et l'étape suivant c'est de couplé ce résultat pour la surface avec l'énergie et le temps d'exécution lorsque nos fonctions s'exécutent dans notre design.

```
Generation No.      ->5
 variables (real 0 binary 4)  fitness

IC1   ID1   IC2   ID2     UTIL(%)  POW(W)
512 2048  512  4096   17.9200  1.8210
512  512  512  4096   18.1900  1.8170
8192 2048 512  4096   17.6900  1.8290
2048 4096 4096 4096   17.4400  1.8340
1024 512  512  4096   18.4300  1.8190
2048 4096 4096 4096   17.4400  1.8340
2048 4096 8192 4096   17.4400  1.8360
512  2048 2048 4096   17.6800  1.8280
2048 2048 4096 2048   17.4300  1.8360
1024 512  512  4096   18.4300  1.8190

NO. OF CROSSOVER = 13
NO. OF MUTATION = 78
```

(a) Fronts de Paréto à l'itération 5, essai 1 (taille de population = 10, nombre d'itérations = 5), 2 fonctions objectifs

```
Generation No.      ->10
 variables (real 0 binary 4)  fitness (3)

IC1   ID1   IC2   ID2     UTIL(%)  POW(W)
8192  4096  1024  8192    17.9200  1.8360
4096  1024  1024  512     18.7300  1.8200
4096  1024  1024  4096    18.4600  1.8270
512   512   1024  512     18.6900  1.8140
8192 32768 16384  8192    17.4500  1.8840
2048 4096  4096   8192    17.4400  1.8370
512  4096   4096  8192    17.6800  1.8290
512  2048  1024   2048    18.1600  1.8230
8192 32768 16384 32768    17.4500  1.9170
512  512   1024  2048     18.4200  1.8190

NO. OF CROSSOVER = 29
NO. OF MUTATION = 131
```

(b) Fronts de Paréto à l'itération 10, essai 2 (taille de population = 10, nombre d'itérations = 10), 3 fonctions objectifs

Fig. 21. Fronts de Paréto obtenus pour différents essais de notre algorithme automatique

F. Jonction entre l'optimisation multi-objectif et l'exécution directe sur carte

Nous avons évoqués précédemment le fait qu'il est impossible d'effectuer une exploration exhaustive pour évaluer l'ensemble des paramètres associés au système. En outre, il peut s'avérer que pour des systèmes relativement complexes, même avec une exploration stochastique parallélisée, le time-to-market nécessaire avec une simulation du behavior d'un fpga est inacceptable dans les contraintes de temps alloués au projet (par exemple lorsque nous avons besoin d'augmenter le nombre d'itérations de l'algorithme génétique pour une contrainte de non satisfaction avec le front de solutions obtenus). On sait que la simulation à l'aide du logiciel Vivado, est 1000 fois plus lente par rapport à une exécution directe sur la carte fpga. Ainsi, il est très intéressant d'effectuer un

couplage des techniques d'optimisation multi-objectifs avec des évaluations par exécution directe sur carte. Ce travail a été réalisé pour la première fois en [[3], 2006], avec l'exploration des configurations des caches d'un microblaze, et l'algorithme NSGA-II dans la boucle. Ce travail a été étendu récemment à des architectures multi-coeurs avec 4096 processeurs.

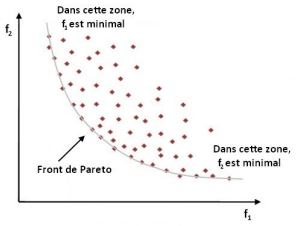

Fig. 22. Front de Pareto

VII. Conclusion

Dans ce projet, nous avons développé une série de méthodes dans différentes plates-formes pour construire un drone de livraison. Cinq fonctions ont été implémentées pour la perception, le traitement et le contrôle du drone. Nous avons évalué ces méthodes dans des processeurs embarqués, ARM et MicroBlaze. Pour améliorer la capacité de traitement, nous avons conçu MPSOC pour exécuter les méthodes en parallèle. De plus, un système NoC est conçu pour communiquer efficacement entre différentes parties. Malheureusement, en raison de la limitation du temps, nous n'avons pas évalué les performances de ces méthodes sur la structure MPSOC et le système NoC. De plus, un script tcl pour exécuter le programme de manière autonome a été testé sur plusieurs ordinateurs. Bien que toutes les parties du projet ne soient pas terminées, nous avons tout de même appris à développer un drone de livraison complet et applicable. Nous croyons qu'avec plus de temps, nous pouvons mieux le terminer.

References

[1] Vincent Andrew Akpan. "Hard and Soft Embedded FPGA Processor Systems Design: Design Considerations and Performance Comparisons". In: *International Journal of Engineering and Technology* Volume 3.11 (2013), p. 054074.

[2] Christopher G Harris, Mike Stephens, et al. "A combined corner and edge detector." In: *Alvey vision conference*. Vol. 15. 50. Citeseer. 1988, pp. 10–5244.

[3] Riad Ben Mouhoub and Omar Hammami. "MOCDEX: multiprocessor on chip multiobjective design space exploration with direct execution". In: *EURASIP Journal on Embedded Systems* 2006.1 (2006), p. 054074.

[4] AXI Xilinx. "Reference Guide, UG761 (v13. 1)". In: *URL http://www. xilinx. com/support/documentation/ip documentation/ug761 axi reference guide. pdf* (2011).