Méthodologies de conception pour multiprocesseurs sur circuits logiques programmables

Riad Benmouhoub

▶ **To cite this version:**

Riad Benmouhoub. Méthodologies de conception pour multiprocesseurs sur circuits logiques programmables. Sciences de l'ingénieur [physics]. ENSTA ParisTech, 2007. Français. NNT : . pastel-00002797

HAL Id: pastel-00002797
https://pastel.hal.science/pastel-00002797

Submitted on 24 Jul 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° D'ORDRE

UNIVERSITE PARIS-SUD XI
Faculté des Sciences d'Orsay

ECOLE NATIONALE SUPERIEURE DE TECHNIQUES AVANCEES

THÈSE DE DOCTORAT

SPECIALITE : PHYSIQUE

Ecole Doctorale « Sciences et Technologies de l'Information des Télécommunications et des Systèmes »

Présentée par : **Mr Riad BENMOUHOUB**

Sujet :

METHODOLOGIES DE CONCEPTION POUR MULTIPROCESSEURS SUR CIRCUITS LOGIQUES PROGRAMMABLES

Soutenue le 07 Mai 2007 devant les membres du jury :

M AUGUIN Michel	Rapporteur
M ETIEMBLE Daniel	Directeur de Thèse
M MEHREZ Habib	Rapporteur
M HAMMAMI Omar	Co-Directeur de Thèse
M NAJJAR Walid	Invité

.

2

.

Avant-Propos

Le travail de recherche présenté dans cette thèse a été réalisé à l'École Nationale Supérieure de Techniques Avancées (ENSTA de Paris) au sein du laboratoire Électronique et Informatique (LEI). Je remercie Mr Alain SIBILLE, responsable du LEI, de m'avoir accueilli et mis à disposition les moyens pour effectuer ce travail.

Je tiens aussi à remercier mon encadrant Mr Omar HAMMAMI, enseignant chercheur à l'Ecole Nationale Supérieure de Techniques Avancées de Paris, pour son suivi, sa disponibilité et ses conseils qui ont joué un rôle capital dans la réalisation de ce travail.

Je remercie vivement Mr Daniel ETIEMBLE, Professeur à l'Université de Paris Sud et chercheur au Laboratoire de Recherche en Informatique (LRI), d'avoir accepté de diriger ma thèse et pour avoir supervisé mes travaux, ainsi que pour tous ses conseils, ses orientations et ses remarques.

Je tiens aussi à remercier le professeur Michel AUGUIN Directeur de Recherche au CNRS (Sophia Antipolis) d'avoir bien voulu avoir accepter d'être rapporteur pour cette thèse. Sa participation nous honore par son expérience reconnue dans les méthodologies de conception.

Je tiens à remercier le professeur Habib MEHREZ professeur à l'Université de Pierre et Marie Curie d'avoir bien voulu avoir accepter d'être rapporteur pour cette thèse. Sa participation nous honore par son expérience reconnue dans la conception de circuits et les nouvelles architectures FPGA.

Je tiens à remercier le professeur Walid NAJJAR professeur à l'Université de Californie Riverside, spécialiste dans l'optimisation à la compilation pour les FPGA, d'avoir bien voulu avoir accepter d'être invité pour cette thèse lors de son passage en Europe.

Merci à tous mes collègues du LEI à l'ENSTA pour tous les moments de soutien et d'amitié.

Je remercie chaleureusement mon père Ahmed, ma mère Atika, ma femme Dalila, ma soeur Mounia, mon frère Mouloud et ma petite soeur Ratiba de m'avoir toujours soutenu et pour leur aide à travers leurs conseils et encouragements.

Résumé

L'augmentation continue de la capacité d'intégration d'une part, la complexité croissante des applications embarquées d'autre part, ont conduit aux systèmes sur puce (SoC) puis aux systèmes multiprocesseurs sur puce (MPSoC). Une spécificité de la conception de ces systèmes est de devoir aborder conjointement la conception de la partie matérielle et la conception de la partie logicielle. Les outils de conception doivent être capables de travailler à tous les niveaux d'abstraction afin d'exploiter efficacement les ressources sur puce mises à disposition tout en conservant des temps de conception raisonnables pour respecter la contrainte de temps de mise sur le marché. Dans cette thèse, nous présentons une méthodologie de conception pour les systèmes multiprocesseurs sur circuits logiques programmables, dont l'originalité porte sur trois aspects :

- L'utilisation de circuits logiques programmables (FPGA) permet d'éviter des temps prohibitifs de simulation ou de co-simulation matériel logiciel en exécutant directement l'application sur différentes instances de l'architecture MPSoC. La reconfigurabilité simple et rapide des FPGAs permet de programmer rapidement l'architecture du système sur puce, puis de mesurer les temps d'exécution.
- Une exploration intelligente de l'espace de conception à l'aide d'algorithmes de recherche évolutionnaires multi objectifs permet de contourner l'impossibilité d'une recherche exhaustive de l'espace d'exploration architecturale tout en obtenant des résultats proches des solutions optimales.
- L'utilisation d'un langage de haut niveau (Occam) permet de faire rapidement la synthèse d'architectures multiprocesseur, notamment du réseau d'interconnexion, à partir de la description de l'application. Combinée avec le monitoring sur puce, cette possibilité permet d'adapter plus facilement l'architecture à l'application.

Nous validons d'abord l'exploration évolutionnaire multi objectifs sur un système monoprocesseur implanté sur un circuit logique programmable (FPGA Xilinx) en utilisant trois benchmarks relativement simples : deux filtres de traitement d'images et le codeur entropique de JPEG2000. Les objectifs à optimiser sont le nombre de " slices " du FPGA, la taille des mémoires, le temps d'exécution. En faisant varier un certain nombre de paramètres (utilisation ou non du cache données, taille des FIFO de communication), nous comparons les résultats obtenus en terme de performances et d'utilisation des ressources matérielle du FPGA pour une recherche exhaustive d'une part, et une recherche utilisant l'algorithme NSGA-II d'autre part. Les résultats d'exploration au sens de Pareto avec NSGA-II montrent la qualité de la démarche évolutionnaire par rapport à une exploration exhaustive suivie d'une classification de l'ensemble des solutions au sens de Pareto. Ensuite, nous étendons la méthode à un système multiprocesseurs constitué d'une grille 2D de quatre processeurs sur lequel nous exécutons l'enchaînement de trois filtres de traitement

d'images (filtre médian, filtre moyenne et filtre conservatif). Les résultats obtenus confirment l'efficacité de la méthode en terme de qualité des solutions et de temps d'exploration. Ces résultats dépendent cependant de l'architecture multiprocesseur initiale considérée. Pour optimiser l'architecture multiprocesseur en fonction des caractéristiques des applications, nous proposons ensuite une synthèse d'applications parallèles à partir d'un langage parallèle haut niveau. Le réseau d'interconnexion du système multiprocesseur sur puce d'une part, le code C pour les différents processeurs constituant la plate forme d'autre part, sont générés automatiquement à partir de la description Occam de l'application. Occam a été choisi pour sa simplicité, sa prouvabilité mathématique et la facilité pour effectuer les transformations. Nous présentons des résultats d'expériences pratiques sur un cas d'étude. Pour faciliter l'exploration dans l'espace des paramètres des différents modules de l'architecture multiprocesseur, nous avons ajouté une structure de monitoring remontant instantanément les résultats comportementaux des IPs formant la plateforme multiprocesseur et du trafic entre les différents composants. Ce monitoring permet de collecter rapidement des informations utilisables pour fixer avec précision l'espace de recherche pour une exploration : elles peuvent être réinjectées comme entrée pour la spécification parallèle ou comme données pour la reconfiguration dynamique sur FPGA de certains éléments déterminants de l'architecture. Cette approche, utilisant Occam et le monitoring, est testée et validée avec 9 processeurs MicroBlaze sur un réseau de neurones de type SOM (Self Organizing Map).

Mots clés : MPSoC, SoC, NoC, MOEA

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

1.1 Contexte de la thèse

Les progrès continus des technologies CMOS et la multiplication des systèmes électroniques grand public ont conduit au développement de systèmes complets dans une seule puce de silicium. On trouve de plus en plus ces puces dans des applications diverses et variées utilisées dans la vie courante. Elles équipent les nouvelles générations de téléphones mobiles, assurent des fonctions critiques dans les voitures (freinage ABS, gestion des airbags, etc.), constituent le coeur des consoles de jeux (PlayStation 3 de Sony, par exemple) et sont utilisées dans la plupart des appareils multimédia comme les lecteurs/encodeurs vidéo portables, les appareils photo numériques, les assistants personnels (PDA), etc. La figure 1.1 présente les évolutions exponentielles que l'on peut constater depuis le début des années 80. La performance des processeurs a approximativement doublé tous les 18 mois, en s'appuyant notamment sur le doublement tous les 18 mois du nombre de transistors par unité de surface (loi de Moore). En fait, c'est une combinaison de l'augmentation des fréquences de fonctionnement d'une part, et des progrès au niveau de la microarchitecture des processeurs (pipelines, caches, exécution superscalaire, etc.) résultant de l'augmentation du nombre de transistors qui a permis ce gain de performance de 60% par an. La figure 1.1 montre cependant que les besoins des applications, exprimées par la complexité algorithmique ou loi de Shannon, croît encore plus vite que les performances. Les besoins des générations successives de téléphones cellulaires (1G, 2G, 3G) indiqués sur la figure illustrent cette situation. Une autre exponentielle est aussi très significative : le différentiel croissant entre l'évolution des performances et l'évolution de la durée de vie des batteries fait de la performance en fonction de l'énergie consommée un facteur décisif, sur lequel nous reviendrons.

L'évolution exponentielle des performances ne s'est pas accompagnée d'une évolution régulière des méthodes de conception pour concevoir les circuits et les systèmes électroniques. Au contraire, elle s'est traduite par l'apparition régulière de "nouvelles" méthodes de conception, ou plus précisément, de changement dans la

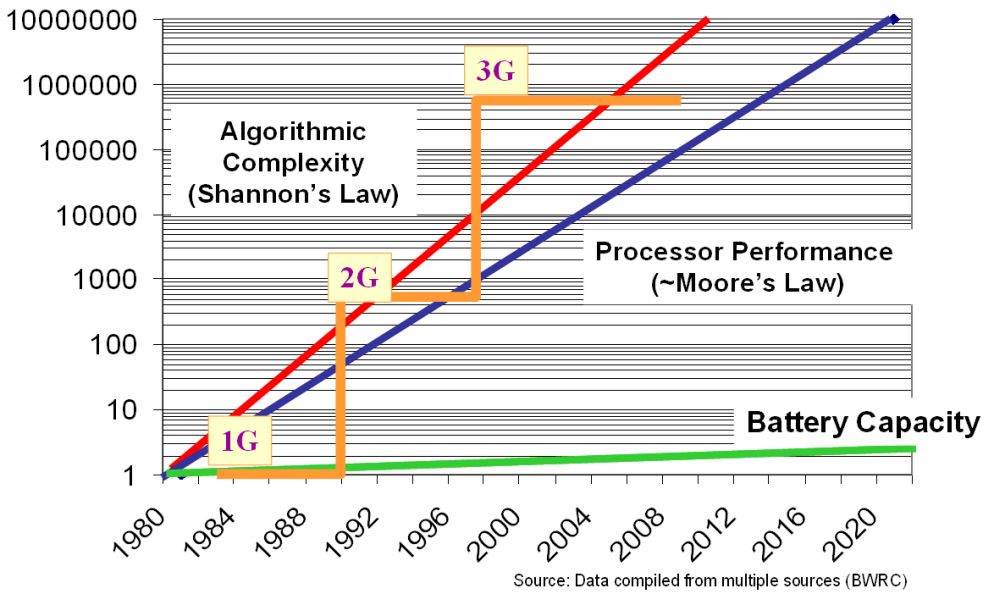


FIG. 1.1 – Des exponentielles significatives

granularité des éléments de base considérés pour la conception. La figure 1.2 résume l'évolution des méthodes de conception depuis le début des années 70. On constate qu'une nouvelle méthode de conception prend comme "élément de base" un ensemble ("mer") d'éléments de base de la méthode précédente.

L'étape actuelle, les systèmes multiprocesseurs sur puce, rassemble sur une même puce plusieurs processeurs homogènes ou hétérogènes, des accélérateurs matériels, des mémoires et des dispositifs d'interconnexion de ces éléments. Dans le cas de processeurs hétérogènes, on trouve généralement un processeur d'usage général de type RISC faisant office de contrôleur, un processeur de traitement du signal, des accélérateurs matériels spécifiques à l'application, etc. OMAP de Texas Instruments, Nomadics de ST sont des exemples de tels systèmes multiprocesseurs sur puce disponibles sous forme de composants d'étagère.

1.2 Méthodologie de conception SoC

Pour concevoir des systèmes monopuces performants, il faut disposer de méthodologies de conception efficaces car s'ajoutent aux contraintes économiques et aux contraintes classiques de conception des circuits intégrés des contraintes techniques liées à l'évolution des technologies CMOS fortement submicroniques et à l'augmentation significative de la complexité matérielle et logicielle des systèmes. Sans prétendre être exhaustif, on peut citer les contraintes suivantes :

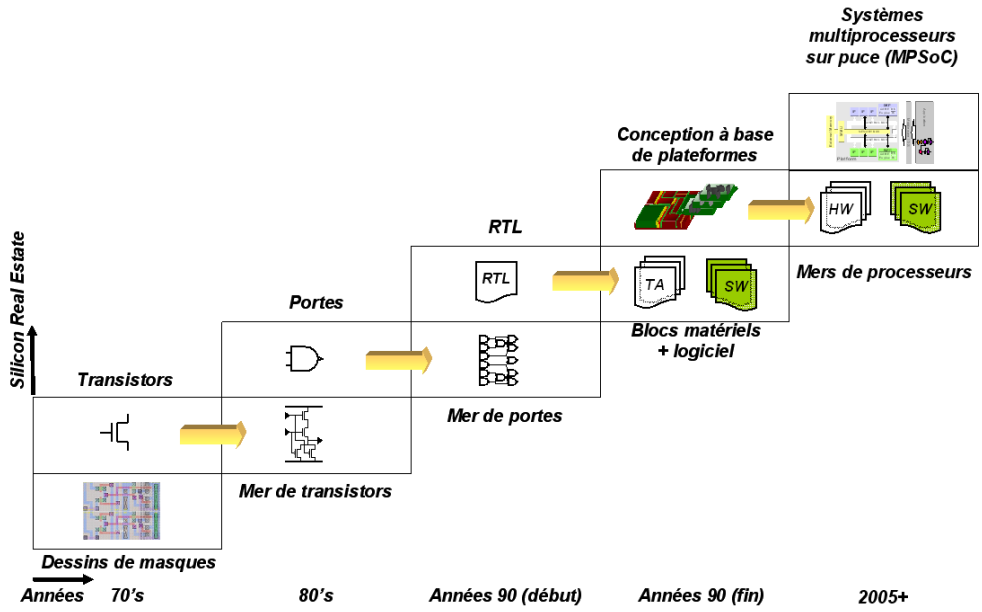


FIG. 1.2 – Evolution des méthodes de conception

- Nécessité d'un développement rapide imposé par la contrainte de mise sur le marché (Time to market)
- Les résultats obtenus du point de vue de la performance, de la surface et de l'énergie consommée constituent les points clés d'une introduction réussie sur le marché
- Les problèmes liés aux contraintes de temps et de propagation se complexifient dans le domaine submicronique
- Les systèmes sur puces englobent un à plusieurs coeurs de processeur, ce qui implique une introduction significative de composants logiciels d'une part et l'irruption des problèmes de méthodologies de programmation parallèle d'autre part
- La gestion concomitante de deux flots matériel et logiciel pose de nouveaux problèmes
- Le nombre croissant de composants hétérogènes, la description de ces composants à différents niveaux de modélisation, l'existence de plusieurs domaines d'horloge, etc. rendent plus complexes le processus de validation, voire de certification dans le cas des applications critiques

La figure 1.3, extraite de [1] , présente un flot de conception classique pour les systèmes monopuces. Elle met en évidence un certain nombre de problèmes. Le partitionnement entre matériel et logiciel est décidé dès la validation fonctionnelle et ensuite les deux flots " matériel " et " logiciel " sont conçus en parallèle pour ne se rejoindre qu'au niveau très bas : description RTL du matériel, instructions

pour le logiciel. L'autre problème est lié à la nécessité d'une validation au niveau ISA/RTL(Instruction-Set Architecture/Register Transfer Level), c'est-à-dire à un niveau très bas, qui peut être très coûteuse en temps si la validation est faite par simulation. Plus généralement, si l'on considère les capacités de conception (productivité des concepteurs de circuits et systèmes) et l'évolution de la complexité des circuits et systèmes, on constate un écart croissant, comme l'illustre la figure 1.4. Alors que la capacité d'intégration croît de près de 60% par an (loi de Moore), la croissance de productivité des concepteurs n'est que légèrement supérieure à 20%. Réduire cet écart est l'une des questions fondamentales, qui peut être attaquée de plusieurs manières :

- concevoir au niveau système
- disposer de méthodologies de conception conjointes matériel-logiciel
- réutiliser des composants prédéfinis et caractérisés (IP pour "Intellectual Properties")

La conception orientée plate-forme vise à répondre à ces exigences. La figure 1.5 illustre cette approche : cette figure est principalement divisée en deux parties. La première partie illustre le flot de conception au niveau système en partant d'une spécification. Une plateforme est générée à base de bibliothèques préalablement définies de composants matériels et logiciels. La plateforme résultante est continuellement testée dans une boucle fermée où une analyse de performances et un partitionnement matériel/logiciel sont effectués à chaque itération. La deuxième partie représente celle sur laquelle nous avons proposé une méthode et travaillé dans cette présente thèse. Celle-ci est basée sur une exploration architecturale des plateformes systèmes sur puce et plus précisément sur puces programmables. Sans vouloir à ce stade discuter en détail les méthodologies de conception des systèmes monopuces, les flots de conception classique ou orientée plate-forme introduits mettent en évidence deux aspects :

- l'évaluation et la validation de la conception conjointe matérielle logicielle est généralement faite par simulation, et plus précisément par co-simulation
- L'optimisation des performances conduit à examiner un grand nombre de solutions correspondant à l'utilisation de composants matériels et logiciels différents et en fonction de la valeur des paramètres associés à ces composants : ceci correspond à l'exploration architecture, et une exploration architecturale exhaustive est impossible

L'idée principale introduite dans cette thèse est d'utiliser des algorithmes d'optimisation pour l'exploration et d'éviter la simulation en la remplaçant par une exécution directe sur FPGA pour récolter les différentes métriques.

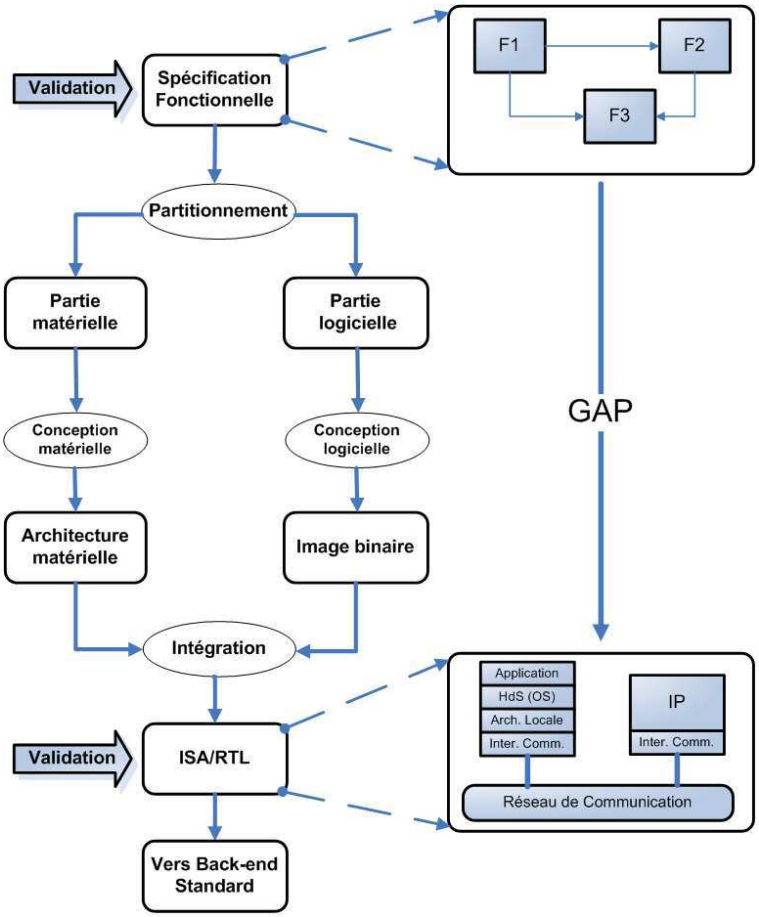


FIG. 1.3 – Flot de conception classique d'un système sur puce

1.3 Simulation ou exécution ?

Comme déjà mentionné, la conception conjointe matériel-logiciel implique la co-simulation des parties matérielles et logicielles pour évaluer le partitionnement matériel-logiciel, estimer le coût et analyser les performances obtenues. Plus généralement, la simulation est la méthode traditionnelle utilisée lorsque les temps et coûts de conception et de réalisation sont tels qu'il est impensable de pouvoir directement mesurer les performances du prototype réalisé. Nous reviendrons plus en détail dans le chapitre suivant sur les méthodologies de conception des systèmes sur puces. La simulation pose traditionnellement deux problèmes : le premier est celui de la modélisation des systèmes à simuler et le second est celui des temps de simulation. Une simulation à un haut niveau d'abstraction (par exemple, la simulation fonctionnelle) sera rapide, mais n'apporte aucune information réelle (Résultats basés sur des estimations) sur les performances en terme de temps d'exécution, énergie consommée, surface utilisée. Une simulation à bas niveau, par exemple au cycle d'horloge près,

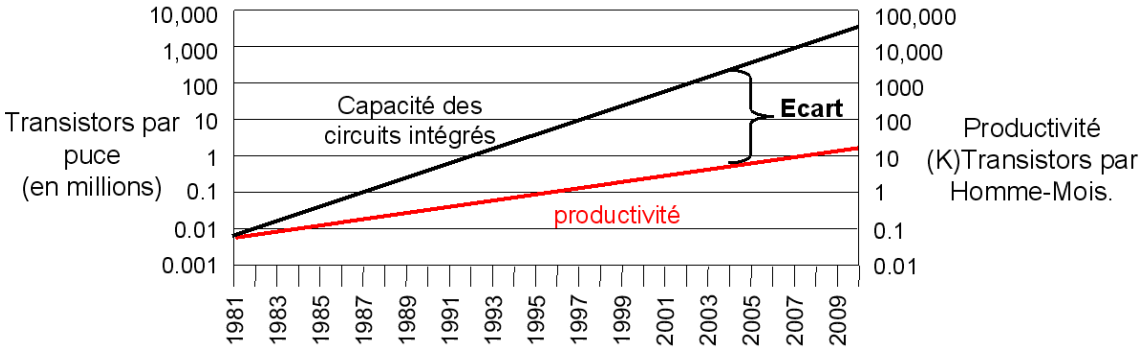


FIG. 1.4 – Ecart entre la progression de la capacité d'intégration et la capacité de production (source : MEDEA+)

peut nécessiter des temps de simulation tels qu'ils réduisent la possibilité d'une large exploration de l'espace architectural. Même les simulateurs utilisant les techniques les plus sophistiquées de simulation ont des temps d'exécution plusieurs ordres de grandeur que le temps d'exécution sur le système réel. Une manière de résoudre les deux problèmes (modélisation et temps de simulation) est de remplacer le simulateur par le système réel en remplaçant la simulation par l'exécution directe. Ceci n'est évidemment possible qu'à deux conditions :

- le temps de conception et le coût de conception du système sur puce doit être faible, ce qui implique qu'il puisse être conçu à partir d'une plate forme matérielle à faible coût, qui soit facilement configurable par une technique de " programmation " du matériel
- il doit être facile de changer les composants du système sur puce, et les paramètres de ces composants, ce qui signifie que le système doit être facilement reconfigurable.

Les progrès technologiques réalisés ces dernières années sur les composants programmables, plus précisément les FPGA (Field Programmable Gate Array) ont permis l'amélioration de leur capacité d'intégration ainsi que de la connexion entre leurs différentes cellules logiques, rendant ainsi possible d'y implémenter tout un système mono ou même multiprocesseur appelé système sur puce programmable (SoPC : System On Programmable Chip). Les circuits programmables sont devenus très performants et sont dotés de composants adaptés aux exigences des applications actuelles tels que les mémoires à accès rapides (Blocs de RAM), d'unités Entrées/Sorties ultra rapides (GigaBit Tranceivers) et de modules DSP (multiplieurs rapides). Les processeurs implémentés sur composants configurables sont le plus souvent des IPs soft paramétrables. On peut aussi trouver sur le marché, des circuits programmables avec des coeurs de processeur matériels (hard core). Comme les prix des composants programmables sur le marché sont de plus en plus bas, implémen-

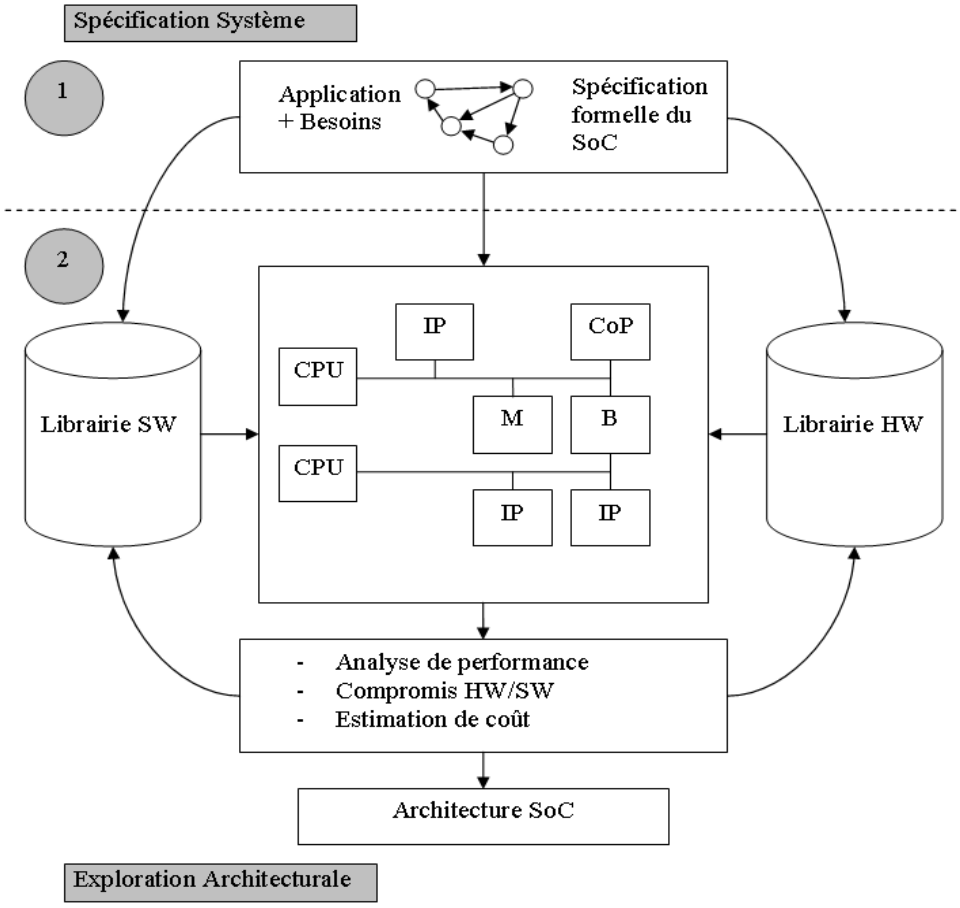


FIG. 1.5 – Flot de conception orienté plateforme

ter un système complet sur un circuit tel qu'un FPGA devient une méthode très attractive. Les systèmes sur puces programmables (SoPC) permettent de construire des systèmes complexes, évolutifs, modulaires, réalisés sur mesure avec une relative facilitée. Pour réduire le temps de spécification d'un système, de réalisation et de validation, les circuits programmables comme les FPGA sont de très bons candidats. Ils permettent d'avoir un prototypage rapide réalisé dans des conditions réelles et répondant aux contraintes de temps de mise sur le marché [2].

Comme déjà mentionné, cette approche permet d'effectuer des mesures de performance sur le système réel (temps d'exécution, utilisation des ressources matérielles, énergie consommée). Il permet une utilisation directe des logiciels fournis par les fabricants de FPGA pour la conception à partir des descriptions du matériel et/ou de composants IP d'une part, des codes binaires générés par les compilateurs pour le code exécuté par les processeurs. C'est cette approche que nous utilisons dans cette thèse. Les limites de cette approche sont liées aux limites de ce qui est implantable dans les circuits FPGA les plus performants. Les limites actuelles sont plutôt du côté

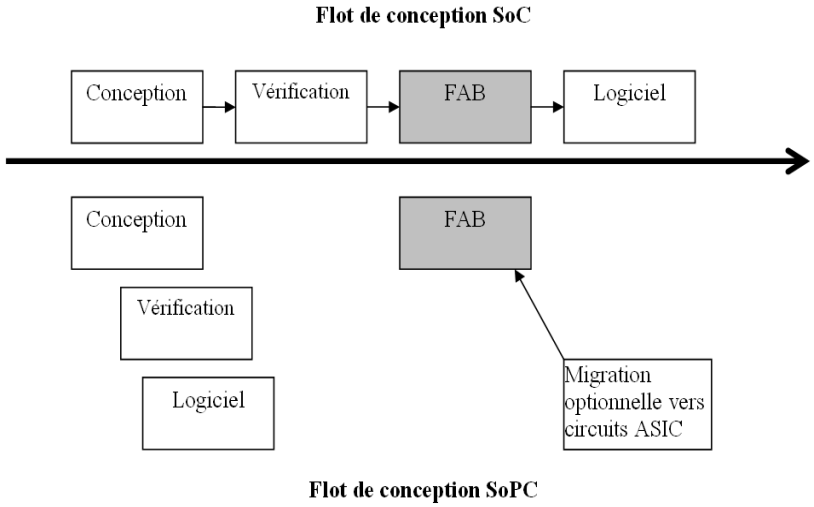


FIG. 1.6 – SoPC vs SoC

de la taille des mémoires RAM disponibles et de la panoplie de processeurs " soft " disponibles. Si de nombreux accélérateurs matériels existent d'ores et déjà sous forme d'IP, et si les processeurs RISC sont déjà disponibles, de véritables processeurs DSP sont encore rares.

1.4 Exploration architecturale

L'utilisation de multiprocesseurs sur puce sur FPGA permet une évaluation rapide des performances de chaque " architecture évaluée " et un temps de reconfiguration raisonnable lié au temps de compilation de la nouvelle configuration. Elle ne permet pas pour autant de prétendre à une évaluation exhaustive de l'espace des solutions. La taille, le nombre de paramètres ainsi que la complexité des applications des nouveaux systèmes sur puce provoquent une explosion combinatoire des dimensions de l'espace d'exploration. Des algorithmes d'optimisation, où les ressources consommées et la performance sont les critères d'optimisation, permettent de réduire la durée de l'exploration architecturale à une valeur raisonnable, tout en obtenant des résultats similaires à une exploration exhaustive. Nous avons dans notre cas utilisé comme algorithmes d'optimisation les algorithmes génétiques multi-objectif. Ces algorithmes utilisent des méthodes d'optimisation stochastiques faisant partie de la classe des algorithmes évolutionnaires. Ces derniers se basent sur des techniques importées de la génétique et des mécanismes d'évolution (Evolution artificielle) dans la nature où des termes comme population, individus, croisements, mutations, sélections, etc. sont employés. Les algorithmes génétiques sont souvent utilisés lorsqu' il s'agit d'optimiser des problèmes NP-complets très complexes et trop longs voir impossibles à explorer par des méthodes déterministes. Un des points

forts des algorithmes évolutionnaires par rapport aux autres méthodes d'optimisation stochastiques, est qu'ils font évoluer en parallèle les nombreux points que sont les individus de la population, tandis que les précédentes méthodes tentent elles, de déplacer un seul point dans l'espace des solutions et se font, par conséquent, facilement piéger dans des optima locaux.

1.5 Synthèse de NoC

Les systèmes sur puce considérés étant multiprocesseurs, le problème du réseau d'interconnexion des différents processeurs se pose. Notre travail ne vise nullement à définir une méthode générale permettant de définir et implanter le meilleur réseau d'interconnexion pour chaque application (topologie, taille des tampons, politique de transferts, etc.). Cependant, le support matériel utilisé (FPGA) permet d'introduire un dispositif de monitoring permettant d'évaluer les performances de communication : débit des transferts, taille des tampons... Il est donc possible d'avoir également une exploration de topologies différentes, et pour une topologie donnée, des dimensionnements des tampons de communication. Pour faciliter cette exploration, nous avons défini une technique de génération automatique de la topologie de communication à partir de la spécification haut niveau de l'application parallèle considérée. OCCAM, langage de spécification pour applications parallèles, a été utilisé pour cela. Il a été introduit afin de décrire une application parallèle et être capable de vérifier formellement sa communication. La compilation du code OCCAM génère une architecture multiprocesseur sur FPGA ainsi qu'un code C relatif à chaque processeur de la plateforme. Les processeurs communiquent entre eux grâce à une couche de communication logicielle basée sur un algorithme de routage Wormhole et à travers des canaux FSL (Fast Simplex Link) de Xilinx. La couche de communication représente une version allégée (version embarquée consommant peu de ressources) de la norme MPI (Message Passing Interface).

1.6 Organisation de la thèse

Comme l'ont montré les paragraphes précédents, l'originalité de notre travail repose sur trois aspects (figure 1.7) :

- l'utilisation de composants programmables pour remplacer la simulation par l'exécution directe
- l'utilisation d'algorithmes d'optimisation pour réduire l'exploration architecturale
- l'utilisation d'un langage de haut niveau (OCCAM) pour générer automatiquement le réseau de communication.

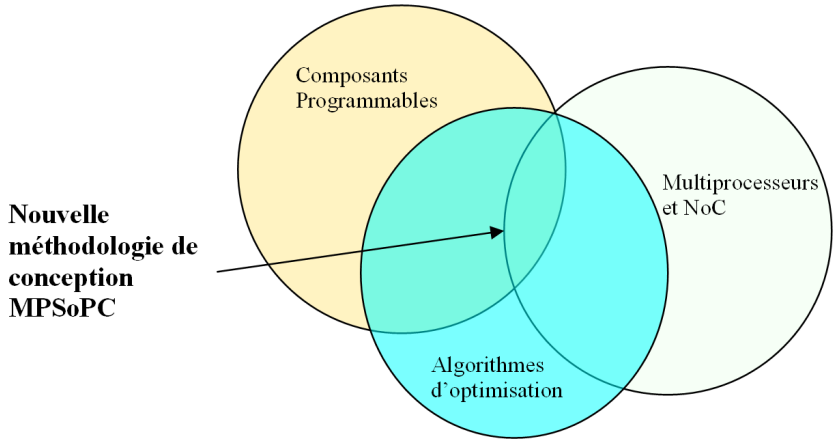


FIG. 1.7 – Spécificités de la méthodologie de conception présentée dans cette thèse

L'organisation du manuscrit est résumée par la figure 1.8. Après cette introduction, le chapitre 2 décrit les méthodologies de conception des systèmes sur puce.

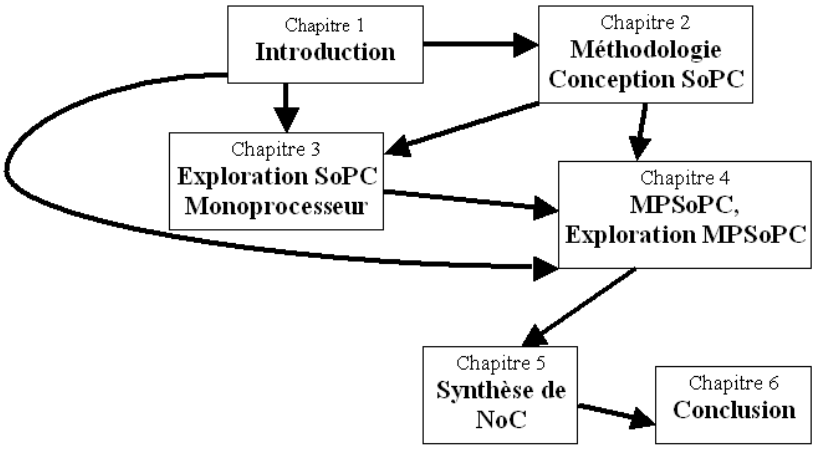


FIG. 1.8 – Organisation de la thèse

Le chapitre 3 examine l'exploration architecturale sur FPGA dans le cadre monoprocesseur. Le chapitre 4 examine les spécificités des multiprocesseurs monopuces sur FPGA et l'exploration architecturale dans le cas multiprocesseur. Le chapitre 5 présente l'utilisation d'OCCAM pour la synthèse du réseau de communication et les résultats associés.

Chapitre 2

Méthodologies de Conception SoPC

2.1 SoPC :Circuits et flots de conception

Les circuits intégrés (IC) n'ont pas cessé d'accroître leur capacité en transistors et ce depuis leur apparition dans les années soixante. Cette capacité est à peu près doublée tous les 18 mois comme l'a prédit la loi de Moore [3]. Ce progrès nous a mené à des circuits "System On Chip" (SoC) comprenant plusieurs composants tels des processeurs, des mémoires ainsi que d'autres fonctionnalités. Ce même progrès a fait naître sur le marché un nouveau créneau qui est celui des "Intellectual Proprety" (IP) ou bien Cores. Les sociétés fabriquant des circuits proposent une multitude d'IP permettant de simplifier la tâche au concepteur. On a vu naître aussi des outils de conception améliorant ainsi l'aspect "Time to Market" en facilitant la connexion de ces IPs et la réalisation de plateformes SoC.

Parmi ces outils, le nouvel outil de conception EDK(Embedded Development Kit) de chez Xilinx permet d'établir un lien direct entre la partie matérielle et logicielle d'un système. Cela nous donne la possibilité à partir d'un même outil gérant les deux flots logiciels et matériels en même temps, d'avoir un système embarqué programmable. XILINX fournit aussi un ensemble d'outils nous permettant de mettre au point des projets sur différentes plateformes reconfigurables. Parmi ces outils, on trouve ISE (Integrated Software Environment) qui va nous permettre de réaliser des projets d'IPs matérielles à partir d'un code de description matériel du type VHDL ou bien Verilog. Le deuxième outil EDK étant (Embedded Development Kit) englobant SYSTEM GENERATOR FOR PROCESSOR et XILINX PLATFORM STUDIO, nous permettra la réalisation de systèmes embarqués sur puce, où un code décrit en un langage à haut niveau d'abstraction s'implémentera sur un processeur matériel (exécution IPs Soft) et une seconde partie concernera les IPs matérielles, sera implémentée au sein d'une même puce et assurera la communication entre les deux différents types d'IPs.

Cependant, ce nouveau domaine de SoC a fait apparaître d'autres problèmes car même si on fournit au concepteur toute la panoplie des circuits SoC, de coeurs

IPs et d'outils lui permettant la mise en oeuvre de systèmes embarqués, on ne lui fournit pas un bon moyen de faire le meilleur choix de coeurs, le meilleur choix des connexions et on ne lui assure pas la meilleure implémentation de son architecture. Le seul moyen dont il dispose pour vérifier son architecture reste la simulation sauf que celle-ci est très lente, ce qui est inadmissible (une journée de simulation pour seulement quelque millisecondes d'exécution réelle) [3] et on ressort rarement, si ce n'est jamais, avec une première implémentation correcte. Tous ces problèmes détériorent l'aspect "Time to Market" et par conséquent les IPs seront plus coûteuses, inaccessibles ralentissant ainsi la production. Les concepteurs sont par conséquent démunis en ce qui concerne le choix des meilleurs paramètres pour leur architecture, et ce que nous proposons dans cette thèse consiste en une méthode avec un outil permettant l'exploration automatique des paramètres architecturaux et les compromis qu'on peut avoir à faire entre performance et consommation d'énergie afin de se rapprocher de l'implémentation la plus optimale possible. Ce qui a bien sûr permis ce travail, ce sont les nouvelles plateformes FPGA capables d'englober un système complet avec un processeur, une architecture de communication, et l'ensemble des IP accélératrices matérielles. Ces circuits FPGA ont donné naissance à un nouveau concept : les systèmes sur puces programmables SOPC (System On Programmable Chip). Avant de rentrer plus en détail dans les systèmes sur puces programmables, nous introduirons dans les sections qui vont suivre les circuits FPGA et les différents outils relatifs aux différents circuits reconfigurables.

2.1.1 Plateformes reconfigurables

Les premiers circuits électroniques programmables sont apparus il y a de cela plus d'une trentaine ans. Ces circuits sont les PROM (Programmable Read Only Memory) représentant des mémoires accessibles aussi bien en lecture qu'en écriture, contrairement à leur prédécesseurs les ROM (Read Only Memory) accessibles qu'en lecture. Les PROMs font partie de la classe des SPLD (Simple Programmable Logic Device). Au départ, les PROM étaient utilisées en tant que mémoire d'instructions pour les ordinateurs, puis leur fonctionnalité a été extrapolée à l'implémentation de fonctions logiques simples. Dans ce cadre, les PROM étaient utilisées comme étant des Lookup Table. Le principal désavantage des mémoires programmables PROM est leur lenteur, ce qu'ils les rendent inadéquates pour des circuits où la vitesse d'exécution constitue une contrainte majeure. Cependant, le succès qu'ont connu les PROM a provoqué l'apparition de nouveaux circuits programmables. Nous pouvons citer parmi eux les PLA (Programmable Logic Array) qui ont été proposés pour palier à la lenteur les mémoires PROM. Il a été proposé par la suite des circuits programmables plus complexes comme les CPLD (Complex Programmable Logic Device) et les FPGA (Field programmable Gate Array). Ces circuits permettent l'implémentation de fonctionnalités très complexes (Processeurs, tous types d'algorithmes). En plus de l'importante variété de fonctions pouvant être implémentées sur

un FPGA, ce dernier offre des fréquences de fonctionnement dépassant les 500MHz. Le dernier circuit né dans la famille des composants reconfigurables est le FPOA (Field Programmable Object Array). C'est un circuit qui reprend les mêmes fonctionnalités qu'un circuit FPGA à un niveau macro. Ces circuits sont 2 à 4 fois plus rapides que les circuits reprogrammables existants sur le marché [4]. Les FPOA exécutent les fonctions logiques à la fréquence du Giga Hertz.

2.1.1.1 Les circuits FPGA

Chaque fournisseur de FPGA a sa propre architecture, mais d'une façon générale, elles sont toutes des variantes de ce qui est présenté dans la figure 2.1. Une architecture FPGA se compose de blocs logiques configurables, des blocs d'entrées-sorties configurables, et d'un réseau d'interconnexion programmable. En outre, il y a des circuits d'horloge pour conduire convenablement les différents signaux d'horloge à chaque bloc logique associé.

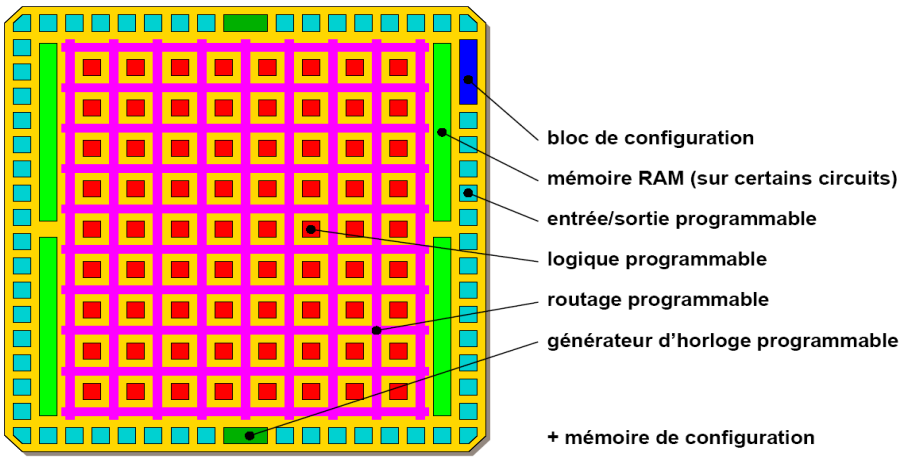


FIG. 2.1 – Architecture interne FPGA

De plus, une architecture FPGA est dotée de ressources logiques additionnelles telles que les Unités Arithmétiques et Logiques (ALUs), les mémoires, et les décodeurs. Les circuits FPGA ont l'entière capacité d'implémenter n'importe quelle fonctionnalité jusqu'ici portée sur des circuits ASIC (Application Specific Integrated Circuits). Le gros avantage ajouté à cela est leur reprogrammabilité, ce qui les rend d'ailleurs de très bons candidats pour le prototypage de circuits. Un circuit FPGA comporte une cellule logique CLB (Configurable Logic Block) qui est basée sur des Look-up Table (LUTs). Une LUT est une petite mémoire dotée d'une rangée d'un bit, où les lignes d'adresse sont les entrées du bloc logique. Dans une cellule configurable CLB, on trouve une RAM pour créer des fonctions de logique combinatoire, également des bascules pour la synchronisation des opérations logiques et des multiplexeurs afin de conduire la logique dans le bloc ainsi que sur les ressources

externes. Afin de connecter les différents blocs logiques et les différents éléments d'entrées/Sorties sur un FPGA, on dispose d'une matrice d'interconnexion et de différents types de fils (liens). Ces fils permettent les connexions suivantes :

- Connexions directes vers les voisins proches
- Connexions générales à travers des matrices de routages et des canaux disposés suivant une topologie simple (grille, tore)
- Connexions longues distances
- Distribution d'horloge spécifique

FPGA, points faibles-points forts Afin de mieux comprendre l'utilité des circuits FPGA nous avons effectué une comparaison entre FPGA et ASIC. Un circuit FPGA présente, comme tout autre circuit, un ensemble d'avantages et d'inconvénients. Ces points sont cités dans ce qui suit :

- Inconvénients :
  1. Nécessité d'une taille importante de silicium (Logique fonctionnalité utilisateur + Logique circuit FPGA)
  2. Une fréquence d'horloge moins élevée par rapport à l'ASIC
- Avantages
  1. Rapidité de réalisation d'une plateforme Hardware
  2. Coût de réalisation très faible
  3. Pas d'importante pénalité sur la fabrication s'il y a erreur dans l'implémentation
  4. Possibilité de réaliser des outils d'aide à la conception avec exploration architecturale sur une implémentation matérielle réelle, avec la possibilité d'avoir une reconfiguration dynamique partielle à la volée (Problème soulevé dans cette thèse)

2.1.2 Exemples de circuits FPGA

Fabriquant	Famille	Technologie	Portes	Nbr bascules	Blocs Mémoires(KBits)	Multiplieurs	Processeurs
XILINX	Virtex-II	SRAM	8M	93.184	3.024	168	-
XILINX	Virtex-II Pro	SRAM	8M	88.192	7.992	444	2 PowerPC
XILINX	Virtex-4LX	SRAM	15M	178.176	6.048	96	-
XILINX	Virtex-4FX	SRAM	11M	126.336	9.936	192	2 PowerPC
XILINX	Spartan 3	SRAM	5M	66.560	1.971	104	-
Altera	Cyclone	SRAM	4M	79.040	7.427	176	-
Altera	Stratix	SRAM	1M	20.060	2095	-	-
Altera	Stratix II	SRAM	9M	143.520	9.383	384	-
Altera	Excalibur	SRAM	1.7M	38.400	256	-	1 ARM933T
QuickLogic	Eclipse II	anti-fusible	320K	4.002	55	-	-
QuickLogic	pASIC	anti-fusible	75K	2.692	-	-	-
QuickLogic	QuickRAM	anti-fusible	176K	2.692	-	-	-
QuickLogic	QuickMIPS	anti-fusible	457K	4.032	83	18	MIPS32 4Kc
Atmel	AT40	SRAM	50K	3.048	18,4	-	-
Atmel	AT6000	SRAM	30K	6.400	18,4	-	-
Atmel	FPSLIC	SRAM	50K	2.962	18,4	-	8-Bit AVR
Actel	Axcelerator	anti-fusible	2M	21.504	294	-	-
Actel	ProASIC PLUS	FLASH	1M	56.320	198	-	-
Lattice	ORCA4	SRAM	899K	18.216	148	-	-
Lattice	ispXPGA	E^2CMOS	1,25M	30.700	414	-	-
Lattice	ECP	SRAM	1M	46.080	645	40	-

TAB. 2.1 – Etat de l'art des Circuits FPGA

2.1.2.1 Virtex-II

Les composants Virtex-II offrent une densité variant de 40K à 8M portes logiques et peuvent atteindre une fréquence de fonctionnement de 420MHz. Ils sont fabriqués avec la technologie 0.15 micron 8 couches de métal avec des transistors haute performance. L'architecture du Virtex-II est donnée par la figure 2.2. Chaque CLB est constitué de 4 slices où chaque slice [1] contient un générateur de fonction, une logique de retenue (carry logic), des portes logiques arithmétiques, un gros multiplexeur et deux éléments RAM pour le stockage. Le nombre de blocs de RAM varie de 4 à 168 blocs pour le plus gros des Virtex-II. Chaque bloc de RAM a une taille de 18Kbits avec une largeur de bus de données maximale de 36 bits (512 mots de 36 bits). Ces mémoires sont dotées de la technologie double port. Le nombre maximum d'Entrées/Sorties utilisateur est de 1108. De plus, les composants Virtex-II possèdent aussi des multiplieurs 18x18 bits (de 4 à 168 multiplieurs), et jusqu'à 12 DCM (Digital Clock Manager) pour la gestion de l'horloge [5].

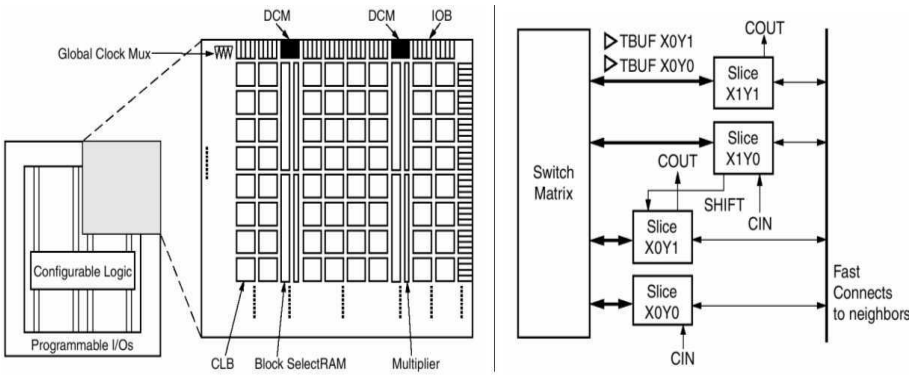


FIG. 2.2 – Architecture interne du Virtex-II

2.1.2.2 Virtex-II Pro

Il y a trois ans, Xilinx a mis sur le marché une nouvelle technologie de circuits FPGA intégrant de nouvelles fonctionnalités. Ces derniers permettent la réalisation d'architectures qui auparavant s'avéraient très compliquées à réaliser. Parmi ces nouveaux circuits : Le VIRTEX-II PRO (Figure 2.3) qui est un composant FPGA intégrant un coeur de processeur PowerPC 405 de chez IBM pouvant fonctionner à plus de 300MHz. Afin de rendre la conception mixte plus simple à réaliser, la famille de composant virtex-II-Pro possède plusieurs atouts tel que : l'intégration d'un ou de deux coeurs de processeur powerPC405. Un grand nombre de cellules logiques pouvant atteindre 44.096 unités (3008 pour le XC2VP4, 9240 pour le XC2VP20) permet d'implémenter des applications assez complexes. Des blocs de mémoire de

[1] Un slice est formé de deux cellules logiques

18Kb (28 pour le XC2VP4, 88 pour le XC2VP20) sont aussi disponibles ce qui diminue l'accès vers des ressources externes. Elle offre aussi des multiplieurs 18x18 bits, des DCM (Digital Clock Manager), et des blocs de Rocket IO transceivers (envoi de données à une vitesse de plusieurs GigaBits/s).

Le choix de ce circuit, pour une première étude au cours de cette thèse, s'est effectué en étudiant préalablement son architecture ainsi que toutes les opportunités qu'il nous offre. Le principe dans VIRTEX-II PRO est que le coeur du processeur PowerPC 405 se greffe directement sur le FPGA (figure 2.4). C'est à l'utilisateur de rajouter les périphériques selon ses besoins. Ce qui permet de moduler à son gré l'architecture à implémenter. Nous avons la possibilité avec le Virtex-II PRO de connecter nos accélérateurs matériels au niveau du bus OPB (On Chip Peripheral Bus) pour la logique lente ou bien sur le bus PLB (processor Local Bus) pour la logique rapide. Ceci nous offre une certaine flexibilité pour le partitionnement matériel/logiciel.

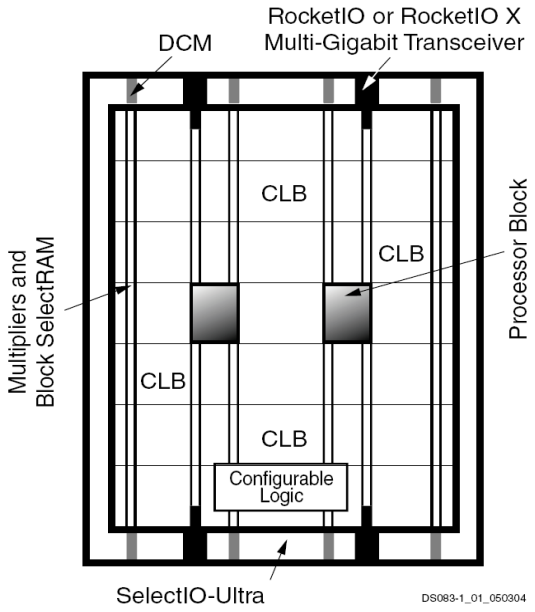


FIG. 2.3 – Disposition interne du Circuit FPGA VIRTEX-II PRO

Pour cela, XILINX fournit des IPs (Figure 2.4) décrites en langage VHDL qui permettent d'exploiter le PowerPC 405 et lui fournir tous les périphériques nécessaires à son fonctionnement. Les premières IP à implémenter autour du processeur sont le " processor Local Bus " et le " On Chip Memory Controller " : la première constitue le bus de communication sur lequel viennent se greffer d'autres IPs ; le "On Chip Memory Controller " permet de connecter le PowerPC 405 aux blocs RAM du FPGA ou bien des RAMs externes où sera logé le code à exécuter.

Une application peut se limiter à une configuration avec un PowerPC 405, une IP " processor Local Bus " et un " On Chip Memory Controller ". L'utilisateur peut

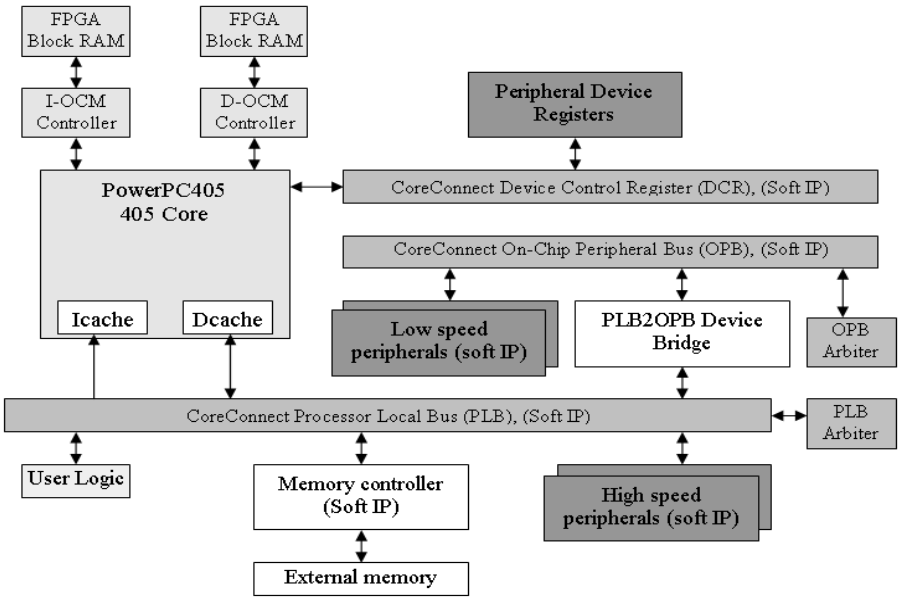


FIG. 2.4 – Architecture standard autour du PowerPC 405 dans le virtex-II Pro

rajouter des bus ou bien des contrôleurs de mémoires externes ou autres IPs, selon les exigences de son architecture. Le constructeur estime la consommation d'énergie du PPC 405 moyenne à 1mw par MHz. Nous basons nos calculs de consommation d'énergie sur cette donnée car il n'existe aucun autre moyen sur cette plateforme, mis à part la simulation (Consommation d'énergie estimée), de déterminer dynamiquement l'énergie consommée.

Les éléments du CoreConnect Les systèmes logiques sur puce sont souvent conçus de manière à être dimensionnés pour une seule application particulière. Chaque application a sa propre architecture. L'extension ou la réutilisation de telles architectures est difficile et nécessite un effort de réadaptation. Pour cette raison, IBM a proposé l'architecture CoreConnect qui offre trois bus permettant de connecter les coeurs de processeur, la logique dédiée, et les fonctions accélératrices. Ces bus sont :

- Processor Local Bus(PLB)
- On-chip Peripheral Bus(OPB)
- Device Control Register(DCR)

Généralement, le PLB fournit un chemin de données pour une communication rapide. En effet, il est utilisé pour connecter les composants tels que les coeurs de processeurs, les interfaces avec les mémoires externes, et les contrôleurs DMA nécessitant une communication rapide. L'OPB est utilisé pour réduire la charge du bus PLB. Il est plus adapté aux composants à communication plus ou moins lente tels que les ports séries, les ports parallèles, UARTs, GPIO. Un composant Maître

sur le PLB peut accéder au composant connecté à l'OPB via une macro passerelle (bridge). Cette passerelle est vue en tant que maître par l'OPB et en tant qu'esclave par le PLB ou réciproquement. Les registres d'états et de configurations de faibles performances sont généralement lus et écrits via le bus DCR. Xilinx a repris le comportement de tous les bus CoreConnect d'IBM et les a décrit en VHDL, ce qui permet d'avoir des bus paramétrables. Cette manière de procéder donne aussi la possibilité d'implémenter seulement la logique dont nous avons besoin pour notre architecture système, comme expliqué précédemment (les circuits Virtex II Pro).

Ajouter de la logique utilisateur (User IP) Pour simplifier la connexion d'un module utilisateur à un des bus du CoreConnect, le concepteur peut se servir de l'une des interfaces de bus portables préconçues par XILINX appelée IPIF (IP interface). L'IPIF prend en compte les signaux du bus, le protocole de la communication et l'ensemble des caractéristiques du bus. L'IPIF présente une interface pour la logique utilisateur dite IPIC, (IP Interconnect). Lorsque la logique utilisateur est conçue avec un IPIC, elle peut être portable et facilement réutilisable avec différents bus en changeant seulement l'IPIF (PLB vers OPB et vice versa). Des fichiers VHDL qui instancient l'IPIF et fournissent le code nécessaire à l'utilisateur pour ajouter ses modules simplifient la tâche de connexion de la logique utilisateur. Ces fichiers sont les User Core Reference Design. Dans les figures 2.5 et 2.6 suivantes, on présente les deux principales méthodes pour ajouter de la logique utilisateur. Dans la première figure, la logique utilisateur est externe au système de processeur : elle utilise l'IPIC pour se connecter au bus via l'IPIF. Dans la deuxième figure, la logique utilisateur fait partie du système de processeur. Elle est compactée avec l'IPIF et l'IPIC dans un seul module.

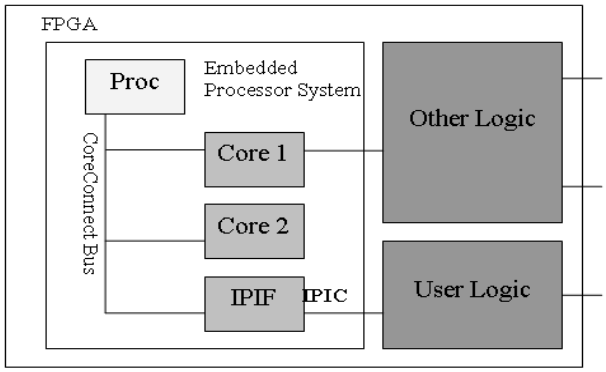


FIG. 2.5 – Logique utilisateur externe au système processeur

Intellectual Property Interface de XILINX (IPIF) L'Intellectual Property Interface (IPIF) que fournit Xilinx avec l'outil "Embedded Developement Kit" EDK

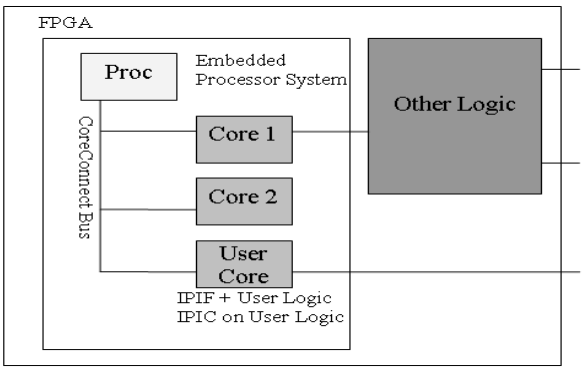


FIG. 2.6 – Logique utilisateur est interne au système processeur

simplifie la tâche à l'utilisateur pour connecter une IP à l'un des bus du CoreConnect PLB ou bien OPB, et cela en fournissant une interface standard pour les types de bus où l'utilisateur pourra connecter sa logique. L'IPIF nous permet donc d'implémenter de manière assez rapide une interface de connexion adaptable entre le bus utilisé et l'IP utilisateur. Avec le code VHDL générique que fournit XILINX, une multitude de composants et d'options peuvent être soient rajoutés ou bien enlevés de notre conception selon les exigences de notre application. XILINX fournit aussi une interconnexion "Intellectual Property InterConnect" (IPIC) qui est une interface commune entre le PLB IPIF et l'OPB IPIF permettant ainsi la connexion de notre IP au bus souhaité de manière rapide et directe. La figure 2.7 décrit l'ensemble des composants du PLB l'IPIF.

Le travail que nous avons effectué s'est fait autour du PLB IPIF. Cette interface nous offre la possibilité d'utiliser neuf services différents qui sont le Slave Attachment, Master Attachment, WRFIFO, RDFIFO, contrôleur d'interruption, la remise à zéro logiciel (S/W reset), le Direct Memory Acces (DMA), la détection d'erreur avec le SESR/SEAR et enfin le Byte steering. Dans les neuf composants qui forment le PLB IPIF, sept peuvent être utilisés ou non (optionnels) selon la description de notre architecture et deux d'entres eux forment les éléments de base du PLB IPIF et doivent par conséquent être toujours présents. Ces deux composants sont : le PLB "Slave Attachment" reprend les fonctionnalités de base pour les opérations esclaves entre le PLB et notre IP, cela en implémentant les protocoles et les timings entre le bus PLB et l'IPIC. Ce module "PLB Slave Attachment" peut être amélioré en lui ajoutant l'option de transfert en mode Burst nous permettant d'atteindre de meilleurs taux de transfert sur le bus. Le deuxième élément de base du PLB IPIF est la fonction "Byte Steering". Ce module se charge de guider les données sur le bus vers leur bonne destination et est nécessaire quand l'espace d'adressage de notre IP est plus petit que la largeur du bus PLB.

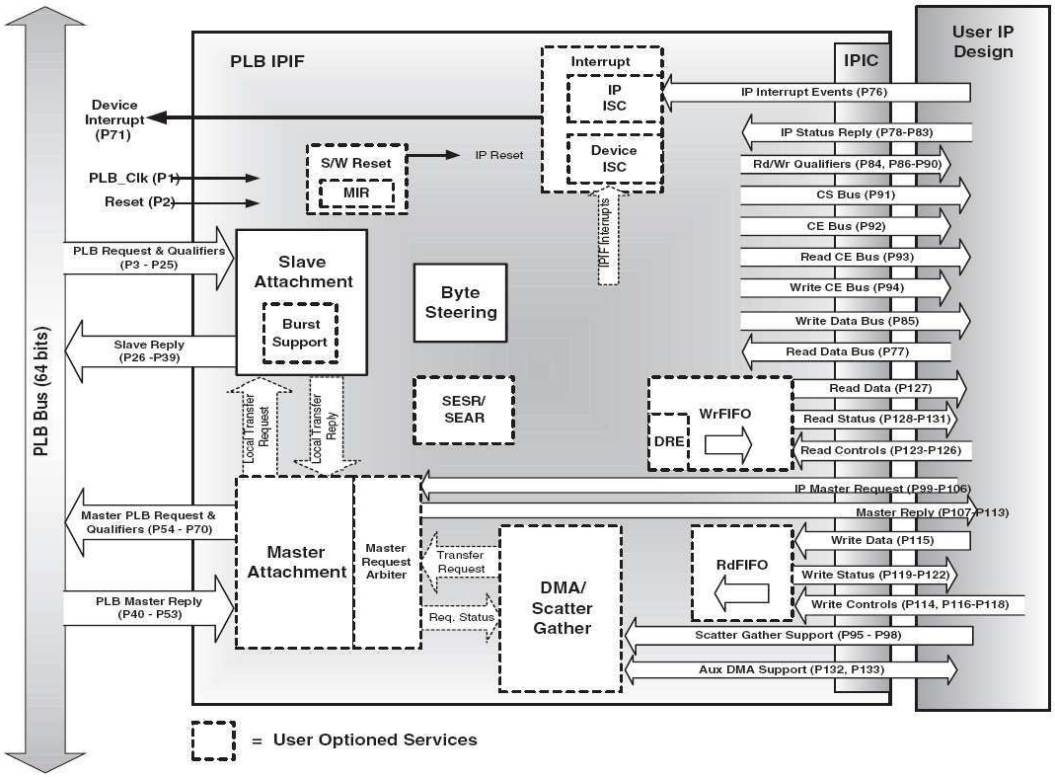


FIG. 2.7 – Composants du PLB IPIF

Résultats de différentes utilisations du PLB IPIF Xilinx a fourni des estimations sur les timings et l'occupation en nombre de cellules logiques concernant l'utilisation modulaire du PLB IPIF. Ces résultats (Tableau 2.2) ont été obtenus en exécutant ISE sur le fichier wrapper du PLB IPIF où tous les résultats de synthèse et d'implémentation ont été regroupés dans un tableau. On remarquera que la fréquence de notre module varie de 84MHz à 178 MHz selon les modules optionnels rajoutés dans notre PLB IPIF. Cette fréquence tend vers une valeur supérieure lorsque l'utilisation des modules du PLB IPIF est moindre. Pour l'obtention de ces résultats, les dimensions des FIFO d'accès en lecture et écriture sur les bus (PLB ou OPB), étaient fixées à 2048x64.

Les ressources du VIRTEX-II PRO La famille VIRTEX-II PRO dispose d'un nombre suffisant de ressources permettant la réalisation d'applications " System On Chip " et ce avec une électronique associée très réduite.

2.1.2.3 Flot de conception SoPC

Nous abordons dans cette partie le flot de conception SOPC. La conception SoPC est de plus en plus réalisée à partir de la réutilisation d'IP préalablement

Parameter value	Slices	Slice Flip-Flops	4-input Luts	Fmax
All services enabled	2487	1947	3947	84,9MHz
Reduce from packet scater Gather to simple scatter gather	2311	1865	364	84,6MHz
Reduce simple scatter gather to simple DMA	2233	1792	3498	84,9MHz
Removed DMA/SC Service	1670	1388	2506	84.3MHz
Removed packet features from packet FIFOs	1448	1278	2237	86,0MHz
Remov packet FIFO service	1078	1061	1749	93,7MHz
Removed IP master service	684	625	988	94,6MHz
Remov SESR/SEAR service	626	551	913	97,0MHz
Remov Interrupt service	550	470	767	105,5MHz
Remove RESET/MIR service	533	459	738	105,8MHz
Remove Burst support	345	327	455	153,8MHz
Remove 8-bit wide user IP address space	248	324	270	155,1MHz
Remove 32-bit wide user IP address space	190	296	166	166,8MHz
Remove data phase WDT service	176	286	138	178,8MHz

TAB. 2.2 – Résultats de synthèse (Performances et ressources) d'utilisation du PLB IPIF

définies. Le flot de conception SoPC est un flot standard Bottom-up où l'on débute la conception avec des IP préconçues. Cette méthodologie de conception s'impose d'elle-même. Effectivement, vu la taille des architectures à réaliser, le meilleur moyen de respecter la contrainte de temps est de se baser sur ces IP. Ces IP comportent des interfaces et des bus standards afin de faciliter leur réutilisation. Sinon, des wrappers sont utilisés afin d'adapter les interfaces non compatibles. Les deux grandes sociétés se partageant le marché des FPGA, à savoir Altera et XILINX, proposent toutes deux un flot de conception pour l'utilisation de leur produit. Leur flot de conception a une base commune présentée par les étapes suivantes :

1. Description du système
   - Schéma
   - Code VHDL
2. Compilation du code VHDL
3. Simulation fonctionnelle
4. Compilation du circuit
   - Conversion de la NETLIST
   - Placement/Routage des cellules
   - Retro-annotation temporelle
5. Verification finale
   - Simulation temporelle

– Configuration/Validation

Le flot de conception SoPC de XILINX sera donné plus en détail dans le chapitre 3.

2.2 Conception mixte et partitionnement matériel/logiciel

2.2.1 Conception mixte matériel/ logiciel

Vu la complexité qu'ont atteint les circuits, entreprendre la conception d'un système à un haut niveau d'abstraction est devenu une réelle nécessité. Cependant, pour être plus efficace, une telle méthodologie de conception doit pouvoir traiter et synthétiser aussi bien les composants matériels que logiciels. Effectivement, dans le domaine de la conception des systèmes électroniques, nous disposons de deux types de modules. Ces deux types sont les composants logiciels et les composants matériels. La partie logicielle représente l'ensemble des circuits processeurs, microcontrôleurs et DSP. Ces composants programmables peuvent exécuter différentes applications et procurent une certaine facilité dans l'implémentation grâce à la simplicité du portage et à la flexibilité de leurs programmes. Par ailleurs, les composants matériels programmables sont le plus souvent les circuits FPGA, CPLD, FPOA.

	Logiciel	Matériel
Portage	+	-
Flexibilité	+	-
Mise sur le marché	+	-
Performances	-	+
Debug	-	+
Coût	+	-
Effort de conception	+	-
Consommation d'énergie	-	+

TAB. 2.3 – Avantages et inconvénients Logiciel/Matériel

Sur ce type de circuits, on peut implémenter n'importe quelle application à l'aide d'un langage de description matérielle. Contrairement aux processeurs, les composants matériels sont des circuits taillés sur mesure pour une application donnée. Cela améliore nettement les performances par rapport à l'implémentation de la même application sur un quelconque processeur généraliste dans le même contexte technologique. C'est pour cette raison que les composants matériels sont le plus souvent utilisés, dans le domaine de la conception, comme étant des éléments pouvant venir en aide à l'exécution d'une tâche critique dans un système. La conception Matériel/Logiciel, appelée aussi conception conjointe, n'est autre qu'un processus décrivant de manière simultanée les parties matérielles et logicielles ainsi que la partie

communication qui les lie. Dans ce qui suit, nous présentons quelques outils pour la description conjointe des systèmes :

2.2.1.1 SystemC

SystemC [6] a été proposé par le groupe Synopsys à l'université d'Irvine en Californie au milieu des années 90. Le premier produit était appelé " Scenic " puis " Fridge ". La première version de SystemC 0.9 a vu le jour en 1999. Le langage SystemC consiste en une bibliothèque créée en langage C++, permettant la description de systèmes logiciels/matériels à l'aide de spécifications exécutables et de plusieurs niveaux d'abstraction, cela pour un même système. Le but de ce langage est d'employer les constructions à un niveau élevé pour décrire les systèmes hétérogènes et pour partitionner facilement le système en des composants logiciels et matériels. Le langage C++ est extrapolé avec une couche spécifique de matériel qui permet aux concepteurs de décrire le matériel en utilisant la description structurale et comportementale. Le langage C++ a été choisi du fait qu'il permet une description orientée objet rendant la réutilisation des blocs plus facile. Les constructions à niveau élevé telles que la modularisation, les processus, les types de données et les niveaux multiples d'abstraction permettent aux concepteurs d'écrire des descriptions de matériel d'une manière simple et concise. La figure 2.8 illustre les différents niveaux où peut opérer SystemC.

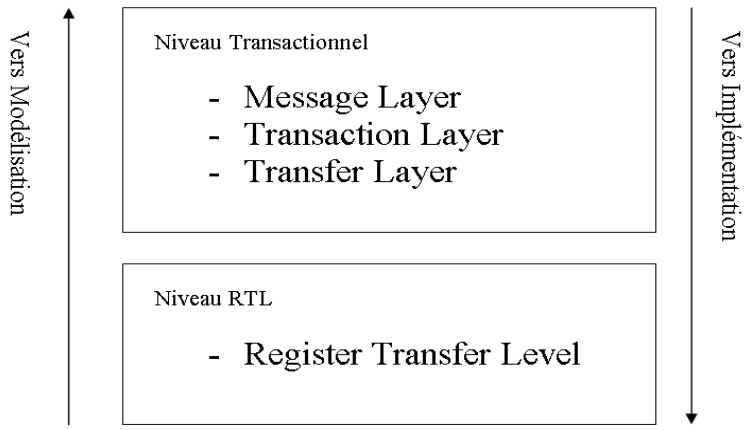


FIG. 2.8 – Niveaux d'abstraction SystemC

SystemC comporte plusieurs avantages dont :
- Simulation rapide
- Plusieurs niveaux d'abstraction
- Partitionnement Matériel/Logiciel à partir d'une même source
- Protocoles de communication
- Création de spécifications exécutables du système

La synthèse matérielle se fait à l'aide d'outils comme CoCentric SystemStudio ou bien Celoxica Agility Compiler. SystemC est devenu un réel standard dans le domaine de la conception électronique[2]. Le tableau suivant illustre les outils les plus utilisés pour le développement de systèmes à base de SystemC.

Société	Outil
CoWare	CoWare ConvergenSC
Synopsys	CoCentric System Studio
Synopsis	SCC Synopsys CoCentric systemC compiler
Mentor Graphics	Vstation TBX
Celoxica	Agility

TAB. 2.4 – Les outils pour SystemC

Cependant, SystemC n'a pas été proposé afin de remplacer le HDL. Aussi le passage d'un SystemC comportemental à un SystemC synthétisable demande beaucoup de travail manuel et de temps. Jusqu'à présent, les synthétiseurs SystemC proposés sur le marché n'ont pas encore prouvé leur entière efficacité. Ceci constitue l'argument qui nous a amené à ne pas utiliser SystemC dans cette thèse. Effectivement, vu l'orientation purement implémentation de nos travaux, SystemC ne pouvait pas encore répondre rapidement à nos besoins. De plus, la précision d'une simulation SystemC n'égale pas encore la précision des métriques extraites d'une implémentation réelle sur FPGA.

2.2.1.2 SpecC

SpecC [7] est une extension du langage de programmation C qui en plus fournit des constructions spéciales pour modéliser des processus concurrents, les transitions d'états, la hiérarchie structurale et comportementale, la gestion des exceptions, la notion de temps, la synchronisation et la communication. Ce langage a été créé à l'université d'Irvine en Californie pour permettre la description, la simulation et la synthèse de systèmes (matériel/logiciel). Une description SpecC se fait à partir de deux types d'objets :

1. le comportement
2. le média de communication

La force du langage SpecC est que tous les modèles sont décrits dans le même langage avec des détails différents. Durant la spécification, on définit la granularité de la phase d'exploration (étape suivante) par rapport au parallélisme disponible. Cette étape est exempte des détails d'exécution et de la notion de temps. Le modèle

[2]SystemC a été approuvé comme standard en 2005 par l'institution IEEE après cinq années d'utilisation (IEEE Std 1666-2005 SystemC Language Reference Manual (LRM))

de spécification inclut également les contraintes non fonctionnelles qui sont imposées à la conception. Pendant l'exploration architecturale, l'architecture du système est dérivée du modèle de spécification. Cette phase inclut l'allocation, le partitionnement et l'ordonnancement. Nous obtenons alors le modèle d'architecture qui reflète la structure de l'architecture du système qui est annoté avec des estimations de temps d'exécution. Dans la phase de synthèse de communication, les communications abstraites dans le modèle architectural sont traduites en des protocoles de communication.

2.2.1.3 Handel-C

Dans le même esprit que SystemC, Handel-C [8] a été créé par la compagnie Celoxica pour la description des systèmes numériques mixtes matériel/logiciel. Il est basé sur le standard ANSI-C pour l'implémentation de fonctionnalités en matériel, l'exploration architecturale et le codesign. Pour supporter la description matérielle, Handel-C gère les longueurs variables des structures de données (vecteurs de bits) ainsi que le traitement parallèle des événements et des communications. La différence principale avec SystemC est qu'il n'utilise pas d'automates à états finis (Finite State Machines) mais une méthode propriétaire de description des écoulements périodiques et parallèles.

Nous présentons dans les paragraphes qui suivent un état de l'art citant les travaux les plus importants effectués dans le domaine de la conception électronique.

2.2.1.4 Vulcan

Cette méthodologie [9] utilise HardwareC basé sur le langage de programmation C. Ce langage permet de modéliser un système en tant qu'un ensemble de processus concurrents communiquant les uns avec les autres. L'architecture cible se compose d'un processeur, d'une mémoire et d'un ensemble d'ASIC. Ce flot de conception commence par une étape de co-synthèse où le but est de mettre toutes les fonctionnalités qui sont susceptibles d'être implémentées en matériel dans la partie matérielle de l'architecture. Les autres parties sont implémentées dans la partie logicielle. Par la suite, l'algorithme Vulcan tentera de réduire les coûts relatifs à l'implémentation matérielle en mettant un maximum des parties matérielles dans les parties logicielles et ce, tant que la contrainte de performance reste satisfaite. Enfin, le partitionnement résultant sera injecté comme entrée pour des outils de synthèse et de compilation logicielle.

2.2.1.5 Polis

Cette méthodologie a été proposée au sein de l'université de Berkeley en Californie [10] afin d'assurer le partitionnement matériel/logiciel, la synthèse automatique

et la vérification des systèmes. L'architecture cible de Polis est composée d'un processeur et de plusieurs coprocesseurs définis par utilisateur (ASICs). La méthodologie POLIS est centrée autour d'une représentation à base de machines d'états finis. Effectivement, POLIS a été développé sur le modèle de calcul formel CFSM ou " Codesign Finite State Machine ". Chaque élément d'un réseau de CFSMs décrit un composant du système à modéliser. Dans Polis, le système est spécifié à l'aide du langage ESTEREL. Dans ESTEREL, le système est représenté comme étant un ensemble de modules concurrents communiquant entre eux à l'aide de signaux et d'événements.

2.2.1.6 Cosyma

COSYnthesis of eMbedded Architectures (COSYMA) [11] développé en Allemagne est une plateforme pour l'exploration du processus de partitionnement matériel/logiciel. La configuration de l'architecture cible est représentée par un processeur et un co-processeur. COSYMA couvre l'ensemble du flot de conception, de la spécification, le partitionnement matériel/logiciel jusqu'à la synthèse matérielle/logicielle. Cela fait de Cosyma une méthodologie complète pour la conception conjointe matérielle/logicielle. L'architecture cible de COSYMA se compose d'un processeur RISC standard, d'une RAM pour la mémorisation des instructions et des données avec un temps d'accès d'un cycle et d'un co-processeur, généré automatiquement, spécifique à une application. La communication entre le processeur et le co-processeur se fait par l'intermédiaire d'une mémoire partagée. Le système utilise le langage Cx qui découle du langage de programmation C avec en plus la gestion de la communication, des processus parallèles et l'introduction de la notion de temps. Dans Cosyma, le partitionnement matériel/logiciel est automatisé et utilise l'algorithme de recuit simulé pour l'optimisation de cette tâche.

Caract.	POLIS	COSYMA	Chinook	Ptolemy	MILAN
Langage de . spécification	Esterel	Cx	Verilog Parallel C	Graph.	Graph.
Abstr. de Comm.	Evén.	Env./Rec.	Signaux	Portholes	Buffers
Mod. de calcul	CFSMs	DFG on ESG	Modes	DF, FSM	SDF, ADF
Concurrence	Mod. concurrents	thread unique	Mod. concurrents	Domain dependent	Mod. concurrents
Partition.	Manual	Automated	Manual	GCLP	DESERT
Granularité	Coarse grained	Fine grained	Coarse grained	Coarse grained	Coarse grained
Vérif. formelle	VIS [12]	N/A	N/A	N/A	N/A
Co-simulation	Ptolemy	CoSim	Pia	Direct.	SystemC
Synth. matérielle	BLIF, VHDL	BSS	génère Netlist	VHDL, RTL	SystemC,
Synth. logicielle	S-Graphs à C	C	C	C, Java	C, Java, MATLAB
Simu. Soft	S-Graphs	Sparc	N/A	S-Graphs	HyperE, Simple Scalar
Simu. Hard	Single cycle exec.	List Scheduling	N/A	THOR	Active HDL
HW/SW Comm.	I/O Ports	mémoire partagée	I/O Ports	I/O Ports	I/O Ports
Arch. cible	Proc. +CFSMs en HW	Proc. Co-Proc.	Multi. proc.	Multi. proc.	Multi. proc.

TAB. 2.5 – Outils de conception pour systèmes embarqués

2.2.2 Partitionnement Matériel/Logiciel

Le partitionnement matériel/logiciel consiste à définir les fonctionnalités de la spécification système qui seront implémentées en matériel ou bien en logiciel tout en définissant de manière optimale leur architecture de communication. Le résultat est une architecture hétérogène optimisant les parties matérielles et logicielles de la plateforme, sans oublier que le but principal du partitionnement est l'amélioration de la performance, de la contrainte de temps de mise sur le marché tout en optimisant les coûts de fabrication du produit. Effectivement, les méthodes traditionnelles de conception de systèmes embarqués traitaient les parties logicielles et matérielles de manière complètement décorrélée. Une spécification décrite dans un langage non formel est envoyée au concepteur logiciel et matériel. Le code relatif à chaque partie sera créé et un partitionnement a priori sera effectué. Par conséquent, par manque d'automatisme dans cette approche, les concepteurs auront recours à un important travail manuel avec des retombées directes sur la qualité du produit en terme de performance et de temps de mise sur le marché. Le fait aussi de procéder à des décisions de partitionnement a priori implique des résultats avec des architectures sub-optimales. Les problèmes liés à une conception traditionnelle avec un partitionnement à priori sont cités dans les points suivants :

- Incompatibilité aux limites des parties matérielles/logicielles
- Résultats avec des architectures sub-optimales
- Beaucoup de travail manuel : flot de conception mal défini
- Mauvaise définition de la spécification : modélisation formelle absente
- Contrainte de temps de mise sur le marché difficilement respectée voir pas du tout

Le partitionnement Matériel/Logiciel est une activité interdisciplinaire optimisant l'implémentation d'une architecture en jouant sur un compromis Logiciel/Matériel.

Un important travail de recherche a été effectué ces dernières années dans le domaine de la conception conjointe. Différents algorithmes ainsi que des méthodologies de conception ont été proposés.

2.2.2.1 L'algorithme GCLP

Global Criticality/Local Phase (GCLP) est un algorithme [13] permettant d'effectuer un partitionnement matériel/Logiciel de manière automatique. Cet algorithme consiste à assigner et ordonnancer les différentes tâches (fonctionnalités) d'une application.

L'algorithme GCLP considère l'application à partitionner comme étant un ensemble de noeuds. Chaque noeud est scruté afin de déterminer son allocation ainsi que le début (Temps) de son exécution. La figure 2.10 présente l'aspect général de l'algorithme. GCLP a la possibilité d'adapter sa fonction " objectif " à chaque étape du traitement. Cette fonction dépend de deux facteurs qui sont : (1) la criticité globale (Global Criticality) qui est une estimation de la performance en temps

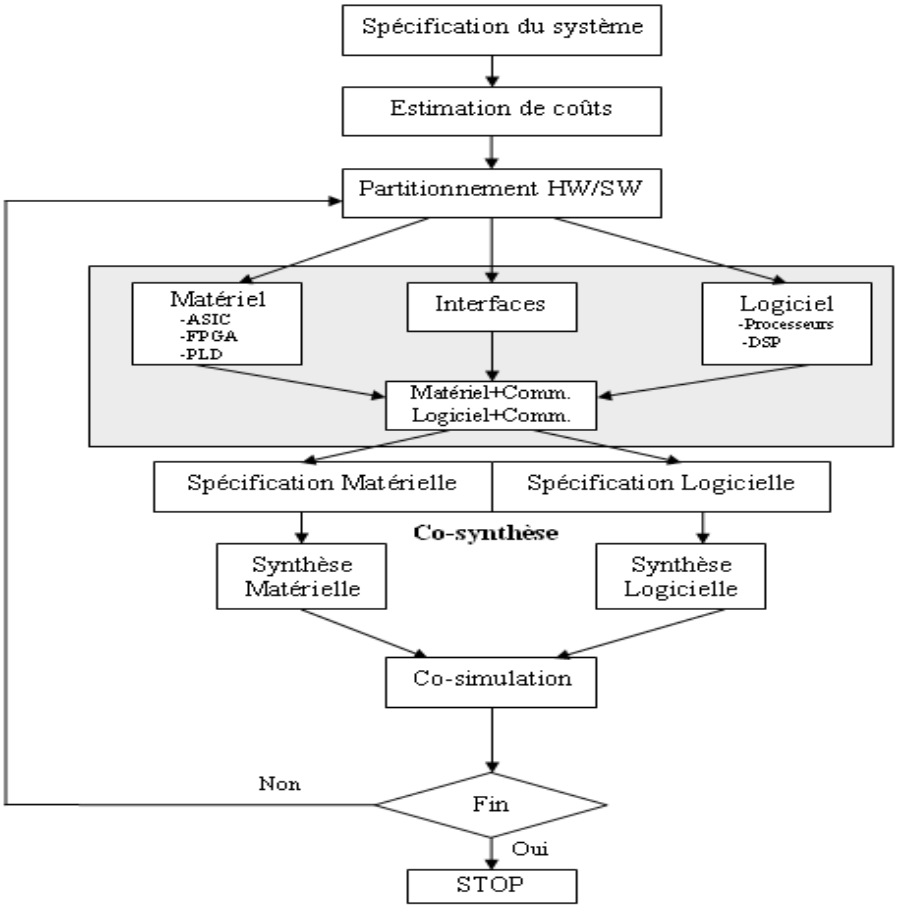


FIG. 2.9 – Flot général de partitionnement matériel/Logiciel Codesign

d'exécution à chaque étape de l'algorithme. (2) la Phase Locale (Local Phase) qui est une mesure qui fournit les caractéristiques d'hétérogénéité des noeuds. Grâce à l'exécution simultanée des deux points cités précédemment, l'algorithme GCLP fournit une solution faisable s'approchant de l'optimum [14].

2.2.2.2 L'algorithme MAGELLAN

L'algorithme MAGELLAN (Multiway HardwareSoftware Partitioning and Scheduling for Latency Minimization of Hierarchical ControlDataflow Task Graphs) rentre dans le cadre des outils d'aide à la conception niveau système pour des applications systèmes embarqués avec d'importantes demandes de calcul (JPEG, MPEG...) [15]. Cette méthodologie propose une heuristique permettant de mapper des graphes de tâches hiérarchiques de contrôle de flot de données sur des modèles architecturaux hétérogènes. L'objectif de la technique est de réduire au minimum la latence des pires cas du graphe de tâches sujets aux contraintes d'espace sur l'archi-

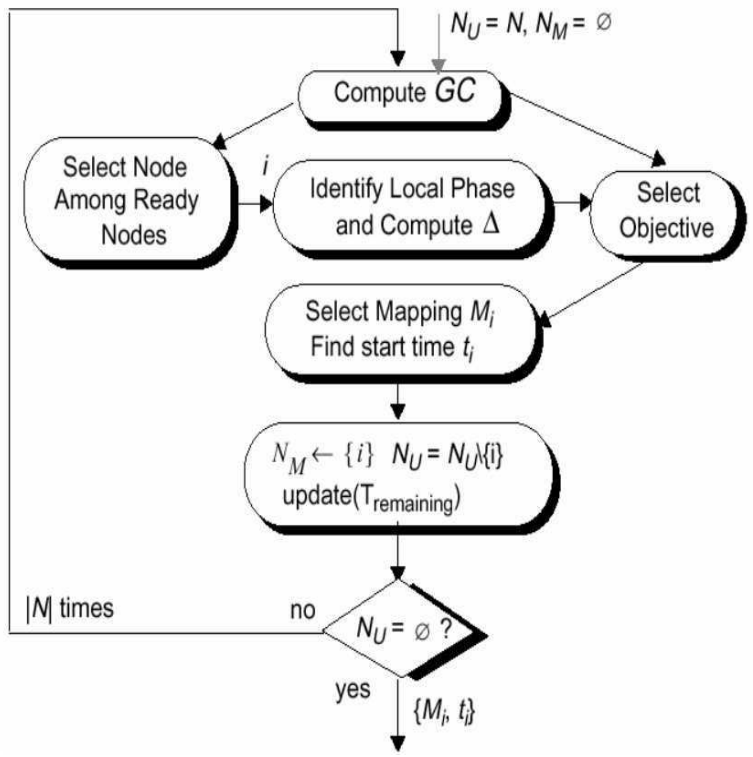


FIG. 2.10 – L'algorithme GCLP

tecture. La figure 2.11 illustre le schéma bloc de l'algorithme. On remarquera que l'algorithme est constitué de deux parties : (1) le partitionneur et (2) l'ordonnanceur. Ces deux parties opèrent sur le graphe de haut en bas de manière hiérarchique.

Une solution initiale est générée par l'ordonnanceur. Ce dernier effectue un mapping, une optimisation et un ordonnancement de l'application. Pour réaliser l'optimisation, l'ordonnanceur utilise le déroulage de boucle (loop unrolling), le pipeline logiciel, l'ordonnancement spéculatif des éléments du chemin d'une tâche cas (case task), et l'optimisation du nombre d'implémentations d'une tâche d'appel. Le répartiteur tente ensuite d'améliorer cette solution en effectuant plusieurs mouvements. Le répartiteur (partitionneur) exécute deux modes de mouvement : un mode hiérarchique, et un mode feuille (leaf). Le répartiteur reste en mode hiérarchique jusqu'à ce que il n'y ait plus d'amélioration puis il passe au mode feuille. MAGELLAN termine lorsque le répartiteur est incapable d'effectuer un nouveau mouvement.

2.2.2.3 L'algorithme COSYN

COSYN est un algorithme de partitionnement matériel/logiciel pour la conception de systèmes embarqués [16]. Il est principalement basé sur une modélisation à base de réseaux de PETRI (CIPN : coloured interpreted Petri net). La figure 2.12

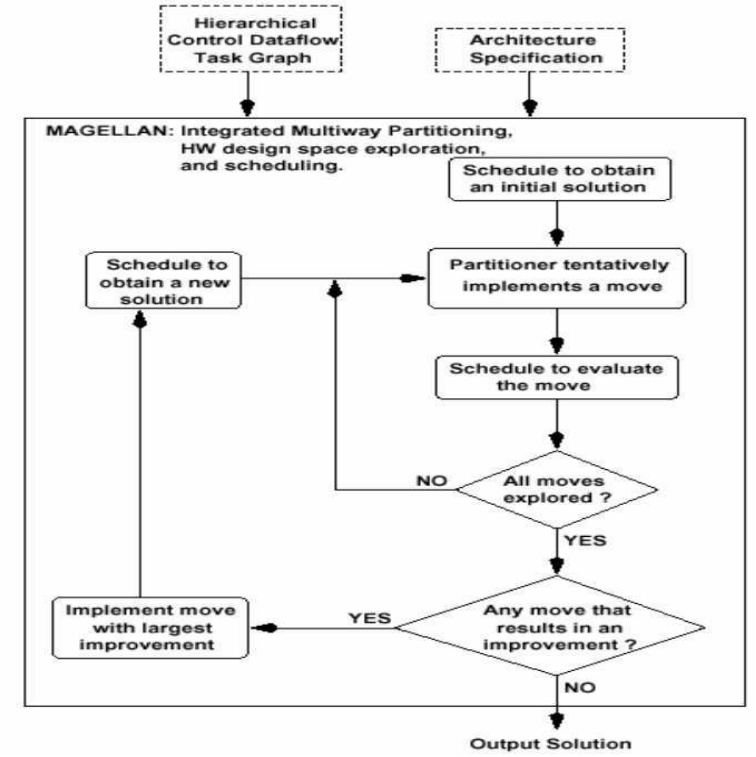


FIG. 2.11 – L'algorithme magellan

illustre le fonctionnement de l'algorithme.

L'entrée de l'algorithme est un graphe de tâches décrit avec des contraintes temps réel (FTE : Finish Time Estimate). COSYN prend en considération différents types d'éléments de communication communicant à travers différents types de médias de communication (Bus local, Point à point, LAN). Le point fort de cet algorithme est qu'il est le premier à prendre en compte l'optimisation de la consommation d'énergie.

2.3 Techniques d'exploration

Les concepteurs se heurtent de plus en plus à un problème très important d'exploration. Effectivement, la taille qu'ont atteint les circuits électroniques actuels rend l'exploration architecturale très compliquée et gourmande en temps d'exécution. Un système sur puce, comme expliqué précédemment, peut être composé de plusieurs processeurs ainsi que de différentes fonctions accélératrices matérielles, ce qui rend l'exploration impossible avec des méthodes traditionnelles exhaustives. L'alternative à cette question est de considérer le problème d'exploration comme étant un pur problème d'optimisation où le concepteur définira une ou même plusieurs fonctions objectif où le but sera de les optimiser selon différents paramètres. Ce domaine

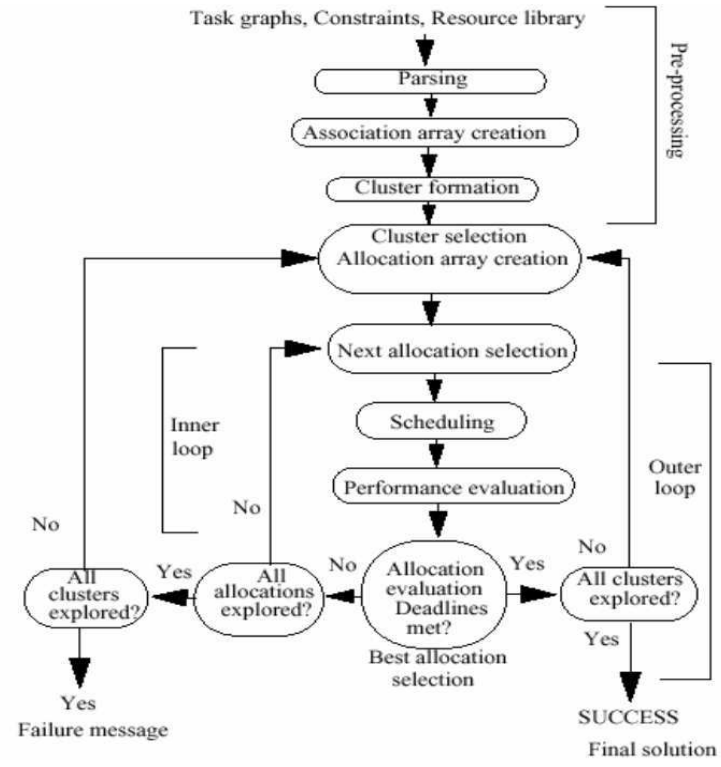


FIG. 2.12 – L'algorithme COSYN

d'optimisation est en pleine expansion dans les systèmes sur puce. Cela permet de trouver parmi un grand nombre de cas, celui qui sera le plus satisfaisant selon des critères donnés.

2.3.1 Algorithmes évolutionnaires

Cette section représente la partie sur laquelle nous avons travaillé et que nous avons utilisée dans notre travail de recherche. Les algorithmes évolutionnaires s'inspirent des lois de l'évolution naturelle en les simulant sur ordinateur. Ils représentent une méthodologie d'optimisation qui consiste à faire évoluer des solutions (individus) dans le temps (à travers plusieurs générations) afin d'arriver à la solution la plus optimale dans un environnement donné. Les algorithmes évolutionnaires ont été utilisés dans plusieurs domaines de recherche où ils ont prouvé une qualité de résultats que les autres algorithmes d'optimisation étaient incapables d'atteindre. Le domaine des algorithmes évolutionnaires se divisent en quatre parties :

1. les algorithmes génétiques
2. la programmation génétique
3. les stratégies d'évolution

4. la programmation évolutionnaire

Nous aborderons dans les paragraphes qui suivent les algorithmes évolutionnaires avec d'autres algorithmes (pour une meilleure comparaison) utilisés dans l'optimisation mono-objectif et multi-objectif.

2.3.2 Optimisation mono-objectif

Cette technique est appliquée à des problèmes relativement simples. Un seul objectif est à atteindre simplifiant considérablement la recherche de l'optimum. On pourra donner l'exemple d'une plateforme système embarqué où le but recherché est de minimiser le nombre de cycles d'exécution. Dans ce cas, l'objectif unique est la performance représentant le seul critère de comparaison entre les différentes solutions existantes dans l'espace d'exploration. Nous présentons dans ce qui suit quelques exemples d'algorithmes d'optimisation mono-objectif. Ceci nous permettra de mieux comprendre leur fonctionnement et justifiera le choix du type d'optimisation que nous avons choisi dans notre étude.

2.3.2.1 Recuit simulé

Pour expliquer la méthode du recuit simulé, on peut se baser sur un problème très connu dans le domaine de l'électronique, en l'occurrence le problème de placement et de routage des circuits électroniques. Le concepteur électronicien part d'un état sous optimal de son système. Le but est bien sûr d'arriver à un état où le placement est fait de telle sorte à avoir des connexions rectilignes tout en minimisant la longueur des fils. On passe alors d'un état de désordre à un état d'ordre optimal. On peut très bien situer ce problème dans un problème de recuit utilisé par les physiciens. En effet, les physiciens ont recours à ce genre de pratiques afin de modifier l'état d'un matériau moyennant un facteur température. En contrôlant ce paramètre température, le physicien peut plus ou moins faire atteindre à son matériau un état optimal (état cristallin optimal : réorganisation efficace de la structure physique). Cette idée de recuit a été utilisée dans des problèmes d'optimisation et a donné naissance à la technique appelée recuit simulé (simulated annealing) [17, 18]. L'organigramme de l'algorithme du recuit simulé est donné par la figure 2.13.

La méthode du recuit simulé procure généralement une bonne qualité de résultat. De plus, c'est une méthode générale facile à programmer et à mettre en oeuvre pour tous les problèmes qui relèvent des techniques d'optimisation itératives. Cependant, il existe des inconvénients liés au nombre important de paramètres (Conditions initiales, taux de décroissance de la température, durée des paliers, critères d'arrêt...) et à la lenteur des calculs dans certaines applications.

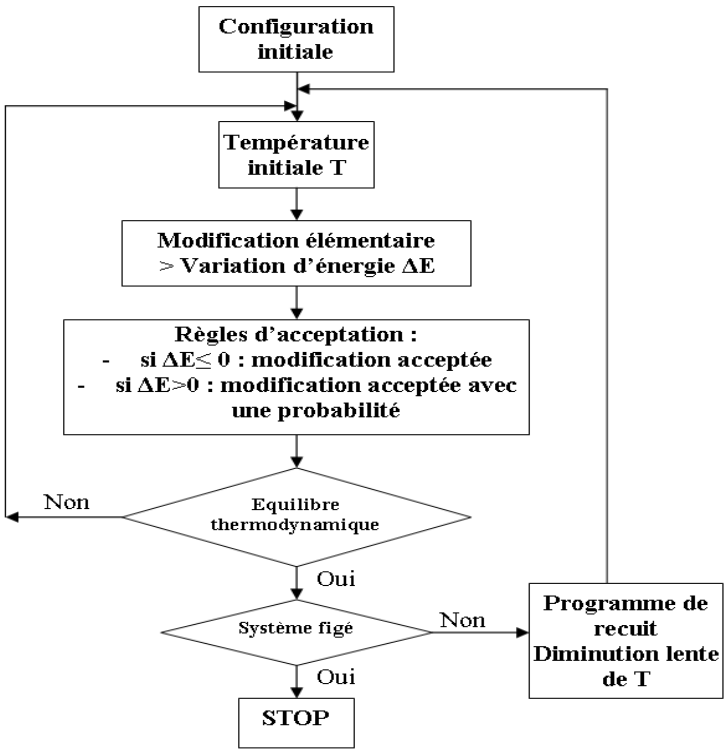


FIG. 2.13 – L'algorithme du Recuit Simulé

2.3.2.2 La recherche Tabou

La recherche avec tabous (Figure 2.14) est un algorithme d'optimisation existant depuis les années soixante qui a été repris sérieusement en 1986 par Fred Glover [19, 20, 21]. C'est un algorithme qui reprend l'esprit d'optimisation du recuit simulé à savoir, optimiser une fonction objectif, mais en plus, on introduit un aspect intelligent en transmettant à la recherche une connaissance du problème. Dans ce cas, la recherche de solutions dans l'espace à explorer ne sera plus effectuée de manière totalement aléatoire mais plutôt dirigée de manière intelligente en dotant la recherche d'une mémoire.

C'est une mémoire à court terme des solutions visitées récemment. On peut aussi bien doter l'algorithme d'une mémoire à long terme en introduisant deux mécanismes supplémentaires nommés intensification et diversification. Ces deux mécanismes testent la fréquence d'occurrences des événements sur une période plus longue. La méthode tabou a donné d'excellents résultats pour certains problèmes d'optimisation. C'est une méthode qui, dans sa forme de base, comporte moins de paramètres que le recuit simulé, facilitant ainsi considérablement sa mise en place. Cependant, elle peut voir sa complexité [22, 23] prendre de l'ampleur dès lors qu'on rajoute des mécanismes optionnels tels que l'intensification et la diversification [30,

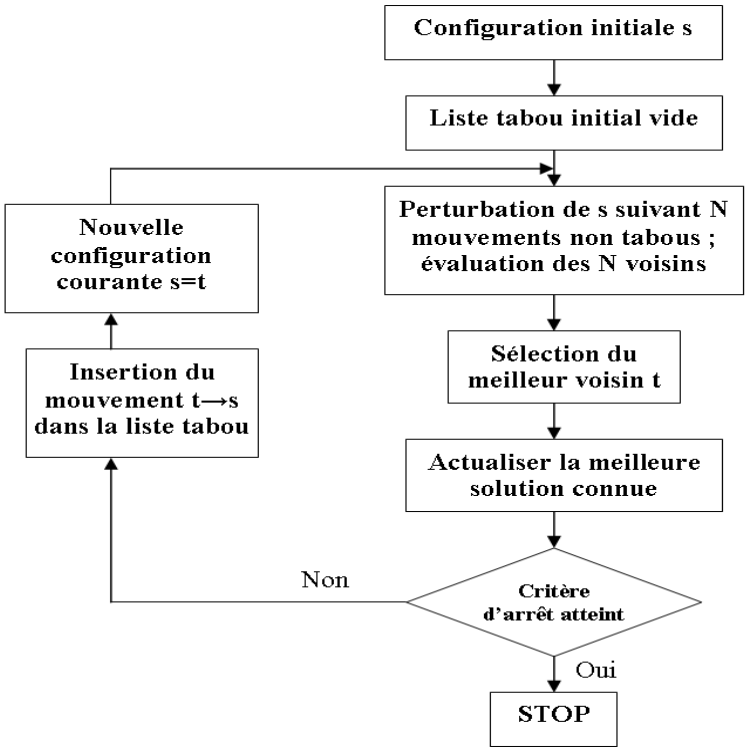


FIG. 2.14 – Algorithme recherche Tabou

31]. La méthode de la recherche tabou existe aussi en version multi-objectif : elle a été proposée par X. Gandibleux et al. en 1996 [24].

2.3.2.3 Algorithmes génétiques

Les algorithmes génétiques représentent plus précisément des algorithmes évolutionnaires inspirés des lois de l'évolution des espèces dans la nature. Ils ont été développés pendant les années soixante par Holland [25]. Ils ont été par la suite appliqués pour la première fois à l'optimisation paramétrique par De Jong en 1975 [26]. Cependant, le manque de puissance de calcul a rendu leur application très difficile. Ce n'est que pendant les années quatre vingt dix, et avec la puissance de calcul existante à ce moment que Goldberg [27] fait connaître les applications à base d'algorithmes génétiques dans la communauté scientifique. Ils représentent des heuristiques basées sur des opérations de variations génétiques, par des croisements, des mutations et des opérations de sélection naturelle. Ce sont des algorithmes d'optimisation intelligents très performants dotés d'une mémoire. Ils ont été utilisés dans des domaines divers et variés où les algorithmes d'optimisation classiques ont échoué. La même terminologie utilisée dans la génétique classique de l'évolution des espèces sert dans les algorithmes génétiques implémentés sur ordinateur.

- Individu : correspond au codage sous forme de gènes d'une solution potentielle à un problème d'optimisation.
- Génotype ou chromosome : c'est une autre façon de dire individu.
- Gène : un chromosome est composé de gènes. Dans le codage binaire, un gène vaut soit 0 soit 1.
- Phénotype : chaque génotype représente une solution potentielle à un problème d'optimisation. La valeur de cette solution potentielle est appelée phénotype.
- Population : représente un ensemble (de taille N) fini d'individus.
- Génération : signifie la population à une certaine itération.
- Performance : c'est une fonction basée sur l'objectif de l'optimisation afin de mesurer la qualité des différents individus
- Evaluation : cette fonction consiste à calculer la performance d'un individu.

Une population représente un ensemble d'individus (solutions) évoluant en parallèle sur plusieurs itérations (générations). Ces différentes solutions appartiennent à l'espace d'exploration du problème à optimiser. C'est un processus darwinien qui fera en sorte que la population de solutions évolue vers un optimum. L'algorithme est stoppé une fois qu'un critère d'arrêt, qui prend en compte à priori la qualité des solutions obtenues, est satisfait. L'algorithme génétique fournit une solution à un problème n'ayant pas de solution analytique. Les algorithmes génétiques ont la capacité de résoudre des problèmes très complexes sans avoir une connaissance a priori de l'espace d'exploration. La figure 2.15 donne plus en détail les différentes étapes d'exécution de l'algorithme génétique.

Les différentes solutions sont dissociées à l'aide d'un critère de qualité représentant une mesure objective qui permet de quantifier la capacité de l'individu à résoudre le problème en question. L'algorithme génétique n'exige aucune connaissance de la manière pour résoudre le problème : il est seulement nécessaire d'estimer la qualité d'une solution potentielle. Il s'agit d'une approche un peu brutale nécessitant une grande puissance de calcul mais présentant l'immense avantage pratique de fournir des solutions pas trop éloignées de l'optimal même si l'on ne connaît pas de résolution algorithmique. Il est également léger à mettre en oeuvre (le moteur est commun, il y a peu de code spécifique au problème à écrire). Nous donnons dans ce qui suit les détails concernant les différents opérateurs de l'algorithme génétique :

Sélection L'opérateur de reproduction est directement tiré de la sélection naturelle qui consiste à dupliquer chaque individu de la population proportionnellement à sa valeur d'adaptation "sélection proportionnelle". Le premier type de sélection utilisé dans les algorithmes génétiques est la technique de la roue biaisée (Roulette wheel Selection RWS) proposée par Goldberg [27]. Cette méthode consiste en une technique stochastique inspirée du principe des roulettes de casino. La figure 2.16 illustre son principe.

La roulette doit comporter autant de cases qu'il y a d'individus dans la population traitée. Chaque case de la roulette a une largeur x_i proportionnelle à sa per-

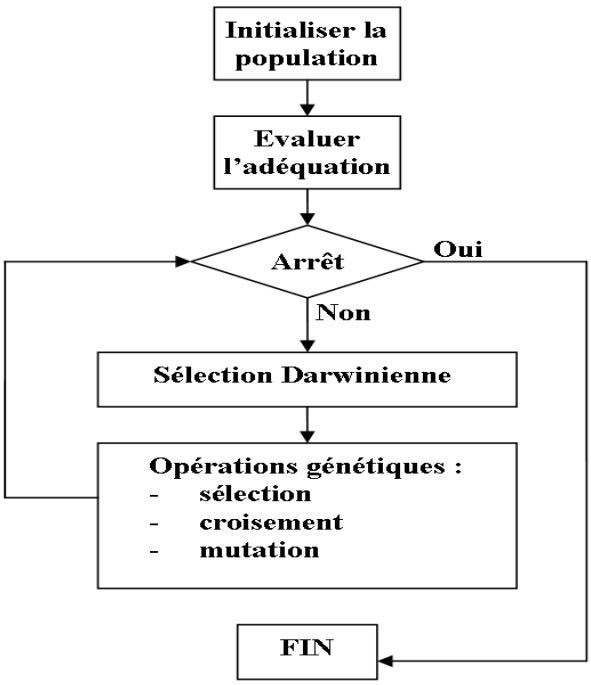


FIG. 2.15 – Algorithme génétique

formance $f(x_i) = f(x_i) / (\sum f(x_j))$. L'espérance n_i relative au nombre de tirages d'un élément x_i de la population en cours de traitement est donnée par l'expression $n_i = (N/\sum f(x_j)) \times f(x_i)$ où N représente le nombre d'individus. Pour réaliser la reproduction, il suffit de faire tourner la roue biaisée ainsi définie autant de fois qu'on a besoin de descendants. Le groupe ainsi obtenu constitue la nouvelle génération qui est ensuite soumise à d'autres opérateurs génétiques. La pression sélective correspond à l'espérance maximale dans une population donnée par $max(n_i)$. Bien que tous les éléments aient une chance d'être tirés, cette méthode favorise le tirage des meilleurs d'entre eux. Cela peut par conséquent faire perdre la diversité de la population si la pression sélective est trop élevée, ce qui implique l'apparition d'un super-individu. Afin d'éviter cette dérive génétique. Baker a proposé en 1987 la sélection à la roulette avec reste stochastique [28].

Croisement Le croisement (2.17) est un processus aléatoire appliqué à des parents tirés au hasard d'une population. L'opération a pour but de générer deux nouveaux individus enfants. Ce processus est stochastique dans le sens où le croisement répété de deux parents donnera à chaque fois des descendants différents. Cela se fait en échangeant les données relatives aux parents. Le croisement peut être avec remplacement et sans remplacement. Le croisement avec remplacement consiste à garder les parents déjà croisés afin de leur donner une nouvelle chance de croisement dans l'itération suivante (génération suivante). Par ailleurs, dans le processus de croi-

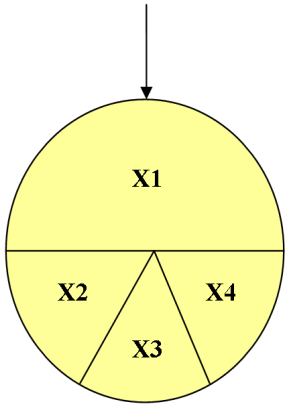


FIG. 2.16 – Roue Biasée

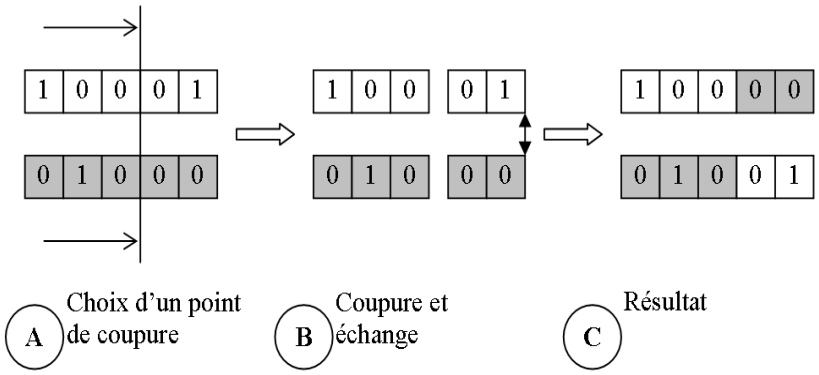


FIG. 2.17 – Croisement

sement sans remplacement, les parents croisés sont retirés de la population. C'est cette dernière solution qui est le plus souvent utilisée [29]. Le croisement respecte en général les règles suivantes :

- le croisement de deux parents identiques donnera des descendants identiques aux parents.
- Deux parents proches l'un de l'autre dans l'espace d'exploration engendreront des descendants qui leur seront proches.

Cependant ces règles ne sont pas toutes suivies de manière systématique.

Mutation La mutation, illustrée par la figure 2.18, consiste à altérer un gène choisi de manière aléatoire sur un individu. Cette modification se fait plus ou moins avec une faible probabilité de l'ordre de $0,1$ à $0,01$ par individu. La sélection et le croisement constituent des éléments indispensables pour assurer l'évolution et la convergence d'un algorithme vers l'optimum. Cependant, il peut arriver au cours de l'évolution que certaines informations se perdent lors du passage d'une génération à la suivante. La mutation a pour rôle de protéger les individus de cette défaillance.

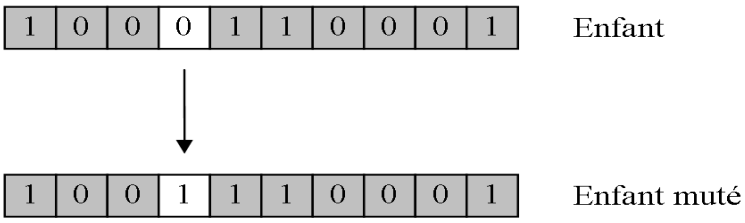


FIG. 2.18 – Mutation

Par conséquent la mutation permet l'apparition de nouveaux individus dans la recherche. La mutation sur une chaîne de bits se fait en remplaçant le bit pris au hasard par son inverse.

Elitisme L'élitisme dans les algorithmes génétiques consiste à conserver dans la population lors de l'évolution d'une génération à l'autre l'individu ayant le meilleur rendu de performance. La performance du meilleur individu de la population courante est ainsi monotone et croissante de génération en génération. Ça permet par conséquent la survie du meilleur parent à travers les générations. Cela suppose bien sûr de garder en mémoire la meilleure solution trouvée par l'algorithme depuis le début de l'évolution, sans que cette solution participe au processus évolutionnaire de l'exploration.

2.3.3 Optimisation multi-objectif

Contrairement à l'optimisation mono-objectif et comme son nom l'indique, l'optimisation multi-objectif consiste à explorer un système et tenter de trouver un optimum répondant à plusieurs fonctions objectif. Le fait de passer à plusieurs objectifs complexifie considérablement la tâche de recherche d'un optimum. En effet, non seulement la comparaison entre plusieurs solutions se fait sur plusieurs critères, mais en plus les différents objectifs d'un problème donné sont pour le plus souvent contradictoires. Par conséquent, l'amélioration de l'un des objectifs se fait le plus souvent au détriment d'un ou de plusieurs autres objectifs. Différentes méthodes ont été proposées afin d'adapter les algorithmes classiques d'optimisation mono-objectif à l'optimisation multi-objectif. Le problème est que la relation entre les différents objectifs n'est pas souvent linéaire. De plus, quand c'est le cas, le résultat est biaisé du fait d'avoir à faire une approximation non adaptée (exp. : somme pondérée des différentes fonctions objectif). Cette méthode s'est classée dans les algorithmes multi-objectif avec agrégation. Une autre astuce consiste à utiliser l'aspect de dominance (optimalité de Pareto). Une solution est dite dominante si elle est équivalente ou plus performante, pour chacun des objectifs, et si elle est plus performante pour au moins un objectif. Cette méthode s'est avérée très adaptée aux problèmes d'optimisation multi-objectif où la recherche dans l'espace d'exploration se fait sur une étude

simultanée de plusieurs solutions. Les algorithmes évolutionnaires se sont basés sur cette idée afin d'évaluer en parallèle les différentes solutions d'une génération donnée assujetties à de multiples critères. Un avantage avec les algorithmes évolutionnaires multi-objectif est qu'ils peuvent soit utiliser des approches basées sur la dominance de Pareto soit en passant par des méthodes tiers (agrégation des objectifs...). Le concept dominance et optimalité selon Pareto est illustré dans ce qui suit.

2.3.3.1 Objectifs multiples avec la méthode d'agrégation

C'est une méthode classique qui consiste à définir des fonctions objectif f_i, traduisant chaque objectif à atteindre. L'étape par la suite sera de les combiner au sein de la fonction d'adaptation. Le but ici est de ramener les différentes fonctions objectif à un compromis entre elles, le plus simple des compromis étant d'exprimer l'ensemble des fonctions objectif par une somme pondérée :

$$f = \sum \alpha_i f_i$$

les poids α_i sont tels que la fonction d'adaptation f soit bornée dans l'intervalle $[0, 1]$. On peut classer les objectifs par ordre d'importance mais les poids seront adaptés par tâtonnement jusqu'à l'obtention d'une solution acceptable. A la place d'une somme, on peut également utiliser un produit du type :

$$f = \prod \alpha_i f_i$$

ou d'autres expressions plus complexes. Dans ce type de méthodes d'optimisation multi-objectif, les résultats obtenus ne sont pas souvent satisfaisants. En effet, deux solutions potentielles dont les fonctions objectif n'ont pas la même valeur peuvent aboutir à une même valeur de la fonction d'adaptation. L'autre problème réside dans le fait que ces types d'algorithmes fournissent à la fin une seule et unique solution, alors qu'il peut y avoir toute une famille de solutions capables de répondre aux objectifs de recherche.

2.3.3.2 Objectifs multiples avec le critère de Pareto

Les algorithmes d'optimisation évolutionnaires basés sur le concept de dominance au sens de Pareto permettent une alternative aux techniques comme celles de la somme pondérée. Cette dominance (Pareto) permet de respecter l'intégralité de chaque critère au lieu de les comparer avec les valeurs des différents critères. Cette approche a été introduite pour la première fois par Goldberg. Elle utilise la notion de dominance de Pareto (figure 2.19) afin de déterminer la probabilité de reproduction de chaque individu dans le but de leur donner un ordre de classement selon ce critère. Cela revient à donner un rang 1 aux individus (solutions) non dominés et rechercher par la suite un autre ensemble de solutions Pareto qui seront classées dans le rang 2. L'opération est répétée jusqu'à la fin du traitement.

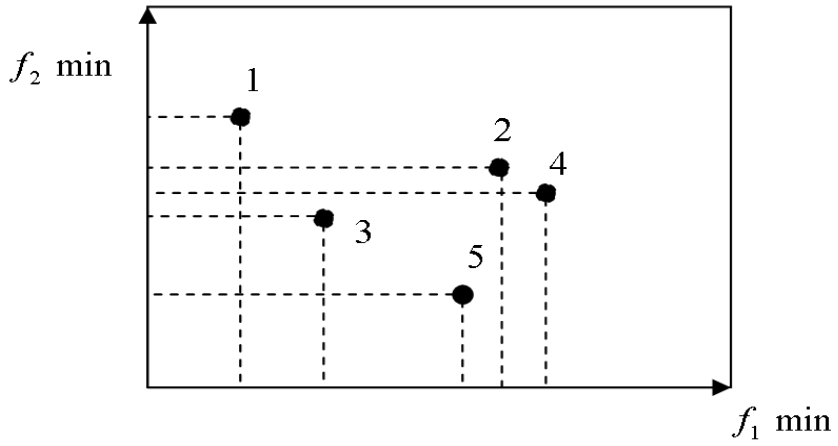


FIG. 2.19 – La dominance au sens de Pareto

Le principe de Pareto ne permet pas d'avoir une solution arrêtée mais par contre fournit au concepteur une aide précieuse lors de la prise de décision. C'est ce qu'a été exploité durant cette thèse.

- Définition 1 : Dominance entre deux solutions : Une solution A domine une solution B pour un problème de maximisation (resp. minimisation) si :

$$z_q(A) > z_q(B)(resp. z_q(A) < z_q(B)), \forall q \in Q$$

  La notation $A = B$ signifie que la solution A domine ou est équivalente à la solution B (i.e $zq(A) = zq(B), \forall q \in Q$).
- Définition 2 : Pareto optimalité : $X \in S$ est un optimum de Pareto strict pour un problème de maximisation (resp. minimisation) si : il n'existe pas $X^{'} \in S, z_q(X^{'}) = z_q(X)(resp. z_q(X^{'}) = z_q(X)), \forall q \in Q$ et
  $\exists q^{'} \in Q, z_{q'}(X') > z_{q'}(X)(resp. z_{q'}(X') < z_{q'}(X))$

2.3.3.3 Strength Pareto Evolutionary Algorithm

Cette méthode d'optimisation multi-objectif basée sur les algorithmes évolutionnaires a été proposée par Zitzler et Thiel en 1998 [30]. Les auteurs ont repris la technique au sens de Pareto afin de comparer les solutions. Ils ont en plus de cela introduit une notion d'archive. Le but est de stocker un ensemble de solutions qui sont Pareto-optimales dans une population externe appelée archive. Une fois cette population stockée, la fonction fitness de chaque individu sera calculée par rapport aux solutions stockées dans cette population externe. Une mise à jour de l'archive se fait à chaque fois que l'on passe d'une génération à une autre. Les individus dominés déjà présents dans l'archive sont supprimés et aussitôt remplacés par les nouveaux individus non dominés (dans la génération en cours de traitement). Il se peut qu'au cours du traitement, l'archive excède le taille fixée (choisie par l'utilisateur). Dans

ce cas, les auteurs ont utilisé une technique de clustering afin de réduire la taille de l'archive. L'utilisation de la technique de clustering garantit la diversité de la population externe. Cependant, le point noir de l'algorithme réside dans le fait qu'il manque de preuve de convergence. Effectivement, pendant la procédure de clustering, un individu de la population externe appartenant à l'optimal de Pareto, peut être remplacé par un individu qui n'appartient pas à l'optimal de Pareto.

2.3.3.4 Pareto Archived Evolution Strategy PAES

Knowles et Corne ont été les premiers à s'intéresser à cette méthode mono-objectif et, après études, ont remarqué que celle-ci fournissait des résultats supérieurs aux méthodes de recherche basées sur une population. Le but par la suite de ces deux chercheurs était de rendre cet algorithme multi-objectif. La particularité de cette méthodologie est qu'elle n'est plus basée sur une population. La recherche se fait sur un seul individu à la fois. Tout comme l'algorithme SPEA, le PAES stocke les solutions temporairement optimales dans une archive externe (population externe). Si la solution ne domine aucune autre solution dans l'archive, le parent et l'enfant sont examinés en terme de proximité avec les solutions présentes dans l'archive. Si dans l'espace des solutions relatif à l'archive, l'enfant réside dans la région de solutions la moins serrée, on l'accepte comme parent tout en laissant une copie dans l'archive. Si l'enfant et le parent ont le même ordre de proximité, l'un d'eux est pris au hasard. L'étalement (Crowding) de l'ensemble des solutions est maintenu en divisant de manière déterministe l'espace entier de recherche dans des sous-espaces de d^n (où d est le paramètre de profondeur et n est le nombre de variables de décision) avec une mise à jour dynamique de ces sous-espaces. D'autres méthodes améliorées ont été proposées, telles que le PESA (The Pareto-Envelope based Selection Algorithm), qui emploient la sous population stockée dans chaque cellule de la grille pour piloter la sélection et la diversité des membres de la population [31]. L'algorithme PAES est relativement simple à mettre en oeuvre et la technique de Crowding utilisée permet une mise à jour de l'archive plus rapide que dans l'algorithme SPEA.

2.3.3.5 L'algorithme NSGA-II

L'algorithme NSGA-II a été proposé par Deb [32]. L'algorithme est une modification de son prédécesseur NSGA [33]. En le modifiant, les auteurs cherchaient à résoudre les problèmes de complexité, de non-élitisme et du sharing. La complexité de l'algorithme NSGA est principalement liée à la procédure de création des différentes frontières. Suite à leur étude, les auteurs ont comparé l'algorithme NSGA-II avec le PAES d'où il ressort que NSGA-II est meilleur pour la répartition des individus sur le front de Pareto. La modification dans NSGA-II s'est faite de telle sorte à ce que la procédure de tri de la population se fait en plusieurs frontières. L'autre aspect gourmand en calcul est la procédure du sharing qui exige le réglage d'un

ou de plusieurs paramètres. Cette procédure a été remplacée dans NSGA-II par le crowding. Pour cet effet, deux caractéristiques sont attribuées à chaque individu :

- i_{rank} représentant le rang de non-domination de l'individu. Ceci dépend du front auquel appartient l'individu.
- $i_{distance}$ permet d'estimer la densité de la population autour d'un individu (distance de crowding).

Dans NSGA-II, la classification se fait sur un ensemble parents enfants en plusieurs fronts de Pareto. A partir d'une population initiale P, on commence par sélectionner tous les individus non dominés afin de former un front de Pareto d'ordre 2. Par conséquent, il ne reste dans la population que les individus non sélectionnés. La procédure est répétée jusqu'à ce qu'il n'y ait plus d'individus dans la population. La complexité maximale de NSGA-II est de MN. Effectivement, pour une population de N individus, il faudra N comparaisons où chaque comparaison est appliquée à M fonctions.

Algorithm 2.3.1: CLASSEMENT DES INDIVIDUS(1)

comment: Classement des individus

$P = population()$/*Ensemble total de la population*/
i=1/*i est le compteur des fronts de Pareto, initialisé à 1*/
while $p \neq 0$
do $\begin{cases} F_i = fastNondominatedSort(P) \\ P = P/F_i \text{/*supprimer les ind. non dominés de la pop. globale P*/} \\ i = i + 1 \text{/*incrémenter le compteur de front*/} \end{cases}$

Nous présentons ci-dessous en pseudo code la technique du crowding distance utilisée dans le NSGA-II pour préserver la diversité des solutions sur le front de Pareto. Cette procédure s'applique sur le dernier front de Pareto pour compléter la taille de la population parents pour la génération suivante. La complexité de cette procédure dépend de la répartition de la population. Quand tous les individus N de la population sont dans le même front I et pour M comparaisons (M tris suivant les différentes fonctions objectifs), $O(MNlogN)$ opérations sont nécessaires.

Algorithm 2.3.2: CROWDING DISTANCE(2)

comment: Crowding distance

$N \leftarrow |I|$ /*Nbre. de solutions dans le front de Pareto*/
for $i \leftarrow 1$ **to** N
  do $I[i]\,distance \leftarrow 0$/*Initialisation de la distance*/
for $m \leftarrow 1$ **to** M/*M est le nombre des fonctions fitness*/

$$\textbf{do}\ \begin{cases} I \leftarrow sort\,(I, m)\ \text{/*Tri des individus suivant fitness m*/} \\ I[1]\,distance \leftarrow \infty \\ I[N]\,distance \leftarrow \infty \\ \textbf{for}\ i \leftarrow 2\ \textbf{to}\ (N-1)\ \text{/*pour tous les autres points*/} \\ \quad \textbf{do}\ I[i]\,distance \leftarrow I[i]\,distance + (I[i+1] \cdot m - I[i-1] \cdot m) \end{cases}$$

Le pseudo code suivant représente la boucle principale de l'algorithme NSGA-II. L'algorithme commence par créer une première population parent P_t de manière aléatoire de taille N. Par la suite, on génère une population enfants Q_t à partir de cette population parents P_t. Cette génération se fait à l'aide des opérateurs génétiques de croisement et de mutation. Puis vient la fusion des deux populations en l'occurrence enfants et parents dans une seule population globale R_t. Après avoir obtenu la population globale R_t, on classe les individus de celle-ci dans un ordre croissant en plusieurs fronts de Pareto. La reconstruction de la population parents P_{t+1} pour l'itération suivante se fait en sélectionnant les individus appartenant aux fronts de Pareto $F1$, $F2$ et enfin $F3$.

Algorithm 2.3.3: BOUCLE PRINCIPALE DE NSGA II(3)

comment: Boucle principale de NSGA II

$R_t \leftarrow P_t \cup Q_t$
$F \leftarrow fastNondominatedSort(R_t)$/*F ensemble des fronts de Pareto*/
$P_{t+1} \leftarrow \phi$/*Initialisation*/
while $|P_{t+1}| + |F_i| \leq N$

$$\textbf{do}\ \begin{cases} CrowdingDistanceAssignement(F_i)\text{/*Calcul de la distance Crowded*/} \\ P_{t+1} \leftarrow P_{t+1} \cup F_i\text{/*Reconstruction de la population parent */} \\ i \leftarrow i+1\text{/*Incrémentation front suivant*/} \end{cases}$$

$Sort(F_i, <_n)$/*Tri des solutions dans l'ordre décroissant*/
$P_{t+1} \leftarrow P_{t+1} \cup F_i\,[1 : (N - |P_{t+1}|)]$ /*Compléter la population parent*/
$Q_{t+1} \leftarrow MakeNewPop\,(P_{t+1})$ /*Application des opérateurs génétiques*/
$t \leftarrow t+1$/*Génération suivante*/

La dernière étape consiste à compléter la population P_{t+1} à une taille de N. Cela se fait à l'aide de la technique du crowding distance sur le front de Pareto F_{j+1}.

2.4 Paramétrage de plateformes pour les systèmes embarqués

Les outils de conception proposés actuellement par des entreprises comme XILINX, Altera, Cadence..., permettent de réaliser assez rapidement un système sur puces programmables FPGA ou même sur ASIC. Par conséquent, n'importe quel concepteur peut désormais avec ces outils réaliser son propre système embarqué grâce à cette assistance logicielle. La prochaine étape logique après cela serait d'essayer d'avoir la meilleure configuration pour l'architecture en jouant par exemple sur les paramètres du processeur (Ex : Taille et fonctionnement du cache d'instructions et du cache de données) aussi sur le type et paramètres des mémoires et des IPs. Le but de tout cela est d'essayer de faire une exploration de tous les paramètres architecturaux afin d'obtenir la meilleure architecture pour notre plateforme. Le problème reste au niveau de la vérification car le concepteur ne dispose que d'outils de simulation pour l'obtention des résultats pour chaque paramétrage effectué et s'assurer du bon fonctionnement de sa conception. Cette simulation reste très lente malgré le fait que l'on dispose de machines très puissantes (Pentium 4 3.2GHz, 2GB RAM). Cette lenteur rend l'exploration presque impossible. Plusieurs méthodes ont été proposées ces dernières années afin de pouvoir explorer de manière efficace l'espace d'exploration d'une plateforme système embarqué paramétrable, l'objectif étant d'arriver à obtenir la meilleure configuration pour notre système. En général, ce travail consiste à trouver le compromis idéal, s'il existe, entre la consommation d'énergie, la performance et l'espace. Une plateforme système embarqué est constituée principalement de processeurs, dotés d'un ou de plusieurs niveaux de caches, de bus de communication, de mémoires embarquées et enfin d'IPs permettant de spécifier la fonctionnalité du système. Chacun de ces composants peut être paramétré dans le but d'atteindre de meilleures performances. Par exemple, le cache peut être paramétré en jouant sur sa taille et son fonctionnement. De même, on peut jouer sur les données qui doivent être mises dans le cache ou non. Toutes ces modifications apportées aux différents composants d'un système vont directement affecter la performance et la consommation d'énergie de ce dernier. Cela dit, le concepteur doit tout d'abord avoir une méthode lui permettant d'explorer efficacement son système d'autant plus que celle-ci dépend de chaque application. Notre objectif serait de permettre au concepteur d'avoir une méthodologie lui facilitant l'exploration de son architecture en lui fournissant en sortie la meilleure implémentation satisfaisant ses contraintes. Une méthode proposée consiste à explorer une application à partir d'une architecture paramétrable décrite en HDL en utilisant des outils standard de compilation, de debugage et d'émulation [34, 35]. Afin d'évaluer la performance et la consommation d'énergie d'une application s'exécutant sur une plateforme système embarqué, plusieurs méthodologies font appel à un environnement de paramétrage qui utilise des approches de mesure de performance et de consommation basées sur l'émulation, la simulation niveau portes (gate-level) ou la simulation RTL. Nous

aborderons par la suite avec beaucoup plus de détails un projet de paramétrage de plateformes[35].

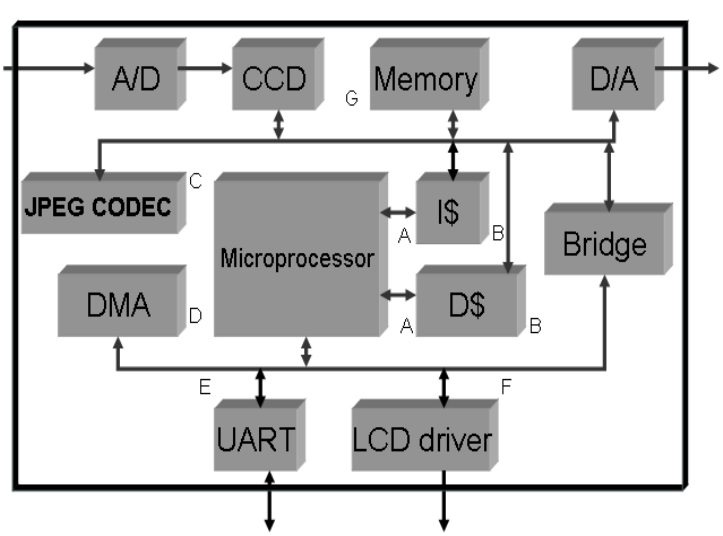
2.4.1 Les plates formes systèmes embarqués

Afin de permettre au concepteur de pallier le problème du temps énorme que prend la simulation lors d'une exploration, plusieurs travaux de recherche se sont penchés sur l'idée de proposer des plateformes pour systèmes embarqués avec des outils utilisés dans un environnement de développement logiciel pour gérer l'aspect compilation, débugage, simulation et bien sûr l'émulation. Ces plateformes peuvent être soient des IP ou bien carrément des circuits intégrés finis(IC). Les plateformes IP sont sous la forme d'un fichier décrivant le comportement d'une fonctionnalité. Cette description se fait à l'aide d'un langage de description matérielle tel que le VHDL. Une fois la spécification traduite en fichier, on pourra par la suite la modifier jusqu'à l'obtention de la meilleure description possible. Le deuxième type de plateformes concerne des circuits intégrés présentés dans des boîtiers. Dans ce casci, le concepteur fait abstraction de tout ce qui est synthèse, simulation et implémentation. Une plateforme à circuits intégrés peut être soit orientée directement pour la réalisation d'un produit fini (Product-oriented) ou bien vers un prototypage (Prototype-oriented). Cette dernière orientation vers du prototypage nécessite des plateformes assez encombrantes, qui consomment beaucoup d'énergie et coûtent énormément à intégrer dans un circuit final [36]. Après la vérification, on passe à l'intégration de l'architecture sur un circuit intégré spécifique (ASIC).

2.4.2 Paramétrage de plateformes

Comme cité précédemment, le prototypage de plateformes se fait sur des plateformes IP ou à partir d'une description comportementale. On arrive après synthèse à un modèle décrit en portes logiques (circuit intégré), ou bien sur des plateformes à circuits intégrés où le prototypage se fait en modifiant les valeurs des registres de configuration du circuit. Le concepteur va par la suite configurer puis exécuter son application en mappant sa fonctionnalité sur la plateforme et en l'exécutant. Le but ici est de lancer successivement plusieurs configurations (boucles d'exécutions) et exécutions en tentant de raffiner la conception et de mieux l'adapter aux exigences.

Pour cela, des fournisseurs proposent des plateformes prenant en compte un maximum d'applications où l'on peut modifier un bon nombre de paramètres. La figure 2.20 montre un exemple de plateforme où l'on peut jouer sur des paramètres tels que : la capacité du cache, la taille de ligne du cache, la précision de traitement du pixel, la taille de la mémoire, le "duty cycle" de l'afficheur LCD et enfin la taille du bloc de transfert du DMA. Une exploration sera appliquée à l'ensemble de ces paramètres afin d'avoir une exécution optimale de l'architecture.



Paramètres	Description
A	Cache Capacity (1-256K byte)
B	Cache line size (32,64,128)
C	CODEC resolution (12-16 bit)
D	Data block size (32,64 byte)
E	UART block size (2-16 byte)
F	LCD duty cycle (60,100 Hz)
G	Memory size (1-16M bytes)

FIG. 2.20 – Plateforme SoC pour caméra digitale

Le groupe de recherche de l'université de Californie Irvine cité dans les paragraphes précédents propose une méthode permettant d'évaluer la consommation d'énergie pour des plateformes paramétrables [36]. Ce groupe se base sur une architecture dont le schéma bloc est illustré par la figure 2.21. L'objectif est de paramétrer cette plateforme et essayer d'en tirer la meilleure implémentation du point de vue compromis entre performance et consommation d'énergie. Cette architecture est constituée d'un processeur, d'un bloc de cache d'instructions, d'un bloc de cache de données, une mémoire principale et de périphériques.

Dans cette analyse, il était question de jouer sur les paramètres du processeur, du cache de données, d'instructions, le bus et enfin de la mémoire. Ce choix a été fait sur la base que cette partie est la plus importante pour ce qui concerne le trafic d'information faisant d'elle la partie la plus gourmande en énergie. Les paramètres du cache d'instructions et de données sont : la taille du cache, l'associativité et la taille de la ligne du cache. Les valeurs que pourra prendre la taille du cache sont 32K, 16K, 8K, 4K, 2K, 1K, 512, 256 ou 128. Pour la ligne du cache, on aura 8, 16 ou 32 octets et enfin pour l'associativité, on aura 1, 4, ou bien 8. Le bus peut prendre différentes valeurs de largeur de données 32, 16, 8, ou 4. L'exploration a été basée sur une approche de simulation. La démarche suivie est représentée par la figure 2.22.

Cette méthode permet d'estimer la consommation d'énergie des composants sélectionnés pour être analysés et permet aussi d'estimer leur performance pour un ensemble donné de paramètres de cache et de bus. Dans un premier temps, il faut sélectionner un ensemble de paramètres, le code source est simulé afin d'obtenir l'activité du cache. L'analyseur de cache va prendre comme entrée cette activité et fournir en sortie la performance du cache, la consommation d'énergie du cache

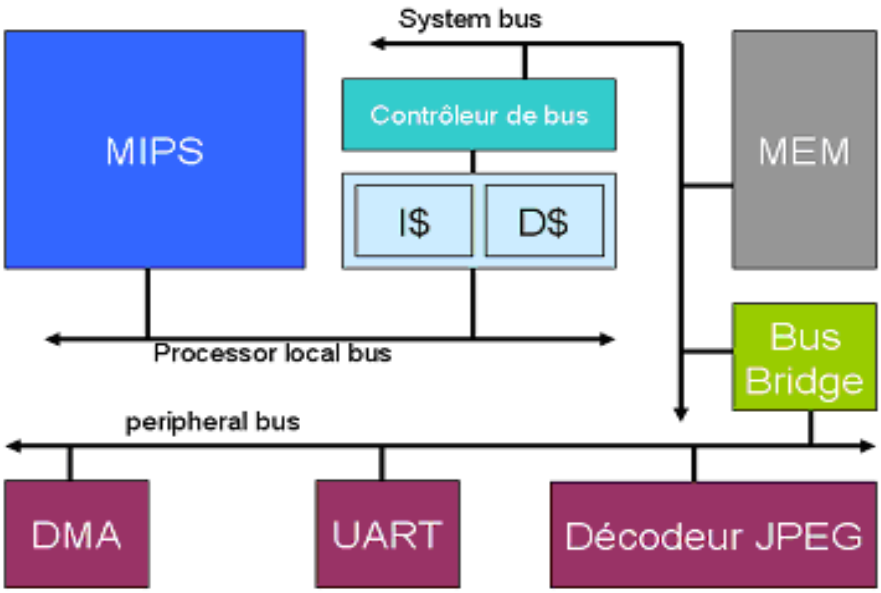


FIG. 2.21 – Architecture cible

ainsi que l'activité du bus. Après avoir obtenu ces données, on passe à l'analyseur de processeur qui lui va prendre en entrée les mesures de la performance du cache obtenu auparavant avec l'activité des instructions, cet outil nous fournira en sortie le temps d'exécution du système avec la consommation d'énergie relative au processeur. Après avoir obtenu les résultats de consommation et d'exécution pour le processeur et le cache, on passe par la suite à l'estimation de la consommation d'énergie du bus qui se fait à l'aide de l'analyseur de bus. Cet analyseur prend en entrée la dynamique du bus et fournit sa consommation d'énergie avec l'activité de la mémoire. Enfin on passe au dernier élément de l'espace d'exploration à savoir la mémoire. Celle-ci est analysée à l'aide d'un analyseur de mémoire en lui fournissant en entrée l'activité de la mémoire en sortie de l'analyseur de bus et en récupérant en sortie la consommation de la mémoire. Une fois la consommation d'énergie pour tous les composants de notre espace d'exploration déterminée, nous passons à une sommation de l'ensemble de ces résultats afin d'obtenir l'énergie consommée pour tout le système. Le tableau suivant donne un récapitulatif des résultats obtenus pour un ensemble de paramètres appliqués au système.

Cette même équipe de l'université de Californie (Irvine) a proposé en 2002 une méthodologie de conception et d'exploration de plateformes systèmes embarqués s'intitulant "Platune" [35]. Platune représente un environnement qui permet de faciliter la tâche du concepteur afin qu'il converge le plus vite possible vers la meilleure implémentation de son architecture. Nous rappelons que le but principal de ce type d'outil est d'essayer de trouver le meilleur compromis entre la consommation d'énergie et la performance du système. Pour cela, Platune est composé de deux éléments :

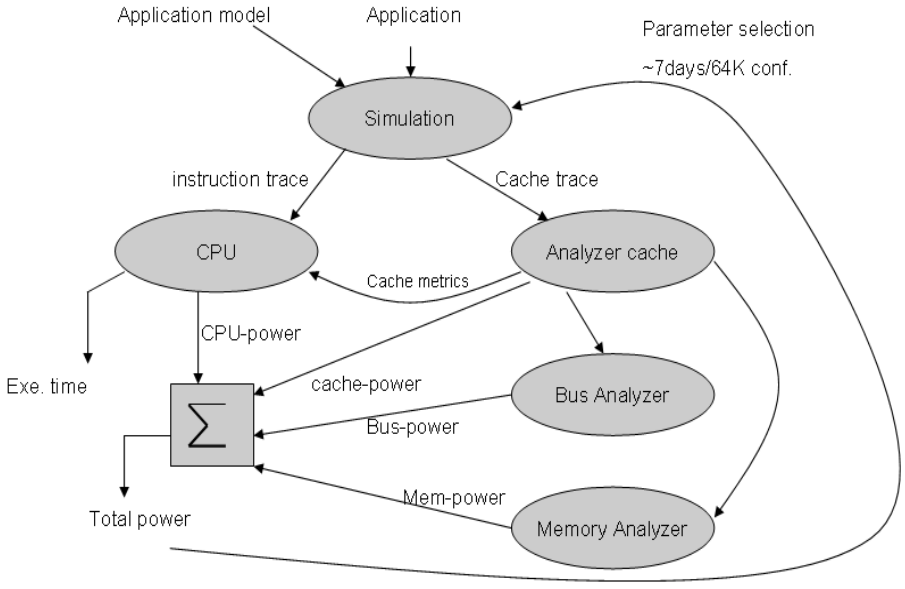


FIG. 2.22 – Analyse de performance basée sur la simulation

- Des modèles de simulation pour chaque composant de la plateforme SoC qui fourniront des résultats de performance et de consommation d'énergie.
- Des modèles de consommation pour chaque composant de l'architecture.

Platune se chargera par la suite de compiler le programme C puis de lancer la simulation afin d'obtenir les résultats de performance et de consommation d'énergie. Cette simulation se fait soit sur une configuration donnée par le concepteur ou bien sur un ensemble de configurations. Platune peut aussi exécuter une exploration automatique sur l'ensemble de toutes les configurations possibles du SoC. La simulation peut se faire à un niveau très élevé de précision afin de trouver la meilleure implémentation comme elle peut se faire à un niveau d'abstraction très élevé dans le but de gagner du temps en exploration. Après avoir sorti tous les résultats de consommation et de performance, on passe à l'exploration. Pour cela, l'équipe Platune propose deux manières de le faire : soit avec une méthode exhaustive ou bien avec une méthode plus efficace qui se base sur l'interdépendance des paramètres. Pour la première méthode, on commence d'abord par apprécier la consommation d'énergie et la performance pour toutes les configurations. Puis les résultats sont classés par rapport à la performance.

La dernière étape consiste quant à elle à sillonner tout l'espace pour éliminer toutes les configurations dont le résultat de consommation est au dessus d'un maximum. Cette méthodologie de conception parait juste et facilite considérablement la tâche du concepteur pour trouver le meilleur compromis entre consommation d'énergie et performance pour une architecture donnée. Seulement, cette méthodologie semble être très limitée dans son domaine d'application vu qu'elle ne peut

être appliqué que sur des plateformes systèmes embarqués où tous les composants doivent avoir leur modèle de consommation d'énergie et leur modèle de simulation. Sinon nous nous verrons dans l'impossibilité d'explorer notre architecture avec cette définition, à moins qu'on n'utilise que des composants dont les modèles de consommation et de simulation sont préalablement définis, ce qui implique directement une réduction considérable de l'espace de composants susceptibles d'être utilisés dans les systèmes embarqués. Sinon, on passe à une écriture systématique du modèle de consommation et de simulation pour chaque composant de notre conception avant de passer à une quelconque exploration. Même si cela est fait, la simulation reste un moyen très lent pour l'exploration et la vérification de n'importe quelle application système embarqué vu que ces systèmes sont tous constitués de processeurs et de composants de calcul assez complexes. Cette lenteur n'avantage pas du tout l'aspect " time to market " ce qui aura un résultat néfaste sur les prix de la technologie réduisant ainsi l'offre et la demande sur le marché et paralysant par conséquent le progrès.

D'autre part, le projet " Platune " reste idéaliste du fait aussi que la simulation constituera toujours un problème pour l'exploration et la vérification même si on part avec l'hypothèse qu'une simulation dure un temps infinitésimal, car cette vérification se fait à l'aide d'un modèle de simulation incapable de prendre en compte tous les facteurs d'une implémentation réelle. Par conséquent, on aura toujours dans notre implémentation une partie qui ne sera pas du tout contrôlée par le concepteur, formant ainsi une lacune dans le flot de conception des systèmes embarqués et ralentit elle aussi l'aspect " time to market " ne satisfaisant pas, comme il le faudrait, la demande du marché. Le projet " Platune " parle d'une manière redondante des paramètres qui sont toujours les caches et la taille des bus, mais ces éléments ne constituent malheureusement pas à eux seuls l'essentiel d'un système embarqué, car l'amélioration d'une implémentation SoC peut aussi se jouer sur les aspects communication et surtout sur la taille des FIFO des bus qui doivent idéalement s'ajuster à l'application en étant ni trop grandes pour ne pas occuper de l'espace inutilement, ni trop petites afin d'éviter les goulots d'étranglement. Ces améliorations peuvent se jouer aussi sur la partie de la zone mémoire qui doit être cachable et celle qui ne doit pas l'être.

Une première amélioration de la méthodologie d'exploration serait l'utilisation des circuits FPGA qui, comme dit précédemment, permettront d'éluder de manière intelligente la simulation et gagner ainsi en temps d'exploration pouvant passer de la journée de simulation à une seconde d'exécution. D'autant plus que les circuits FPGA actuels nous donnent la possibilité d'avoir au sein d'une même puce une matrice de cellules FPGA fusionnée avec des processeurs matériel enfouis au milieu des cellules logiques. On prendra au passage comme exemple le circuit FPGA Virtex-II Pro ou le virtex-4 de XILINX qui peuvent supporter jusqu'à deux processeurs PowerPC405 fonctionnant à plus de 300MHZ au sein d'un même boîtier. Ces circuits constituent à eux seuls des composants de qualité pour l'exploration et la finalisa-

tion des systèmes embarqués. On pourra aussi grâce à la matrice d'interconnexion de ces FPGA avoir des reconfigurations dynamiques nous permettant ainsi de gagner encore plus de temps lors de l'exploration du fait qu'on peut reprogrammer le circuit juste avec la partie affectée par de nouveaux paramètres et laisser le reste inchangé. Générant ainsi un fichier Bitstream plus petit qui sera par conséquent plus rapide à charger. Les circuits FPGA nous permettrons aussi d'explorer des systèmes multiprocesseurs sur puce (MPSoC) qui peuvent avoir jusqu'à deux processeurs PowerPC405 embarqués dans un même boîtier.

2.5 Exploration génétique : applications

Afin de comparer le travail d'exploration à base de recherche exhaustive et l'exploration à base d'algorithme génétique NSGA-II, nous présentons ici une application qui illustre assez clairement les avantages de la deuxième méthode. L'application que nous traitons ici base son travail d'optimisation d'une plateforme embarqué SoC paramétrable, sur l'algorithme d'exploration NSGA-II. De plus, cette application fait une comparaison directe avec la méthode proposée par [35] dans le projet Platune. Ce même projet a été abordé en détail dans la section précédente de ce chapitre. Les auteurs dans [37] proposent un flot de conception exécutant une exploration d'espace pour des plateformes matérielles SoC. Ce flot est proposé avec des outils logiciels permettant d'évaluer et de spécialiser la plateforme pour une quelconque application. Le package d'outils englobe un compilateur, un debugger, une librairie logicielle et des simulateurs. Dans le flot de conception proposé, les valeurs des différents paramètres de la plateforme à explorer sont variés et les solutions Pareto optimales sont graduellement cumulées. L'énorme avantage qu'on peut avoir avec l'algorithme NSGA-II est le fait qu'on puisse explorer les architectures en optimisant plusieurs objectifs à la fois. Les auteurs ont exploité cet aspect afin d'introduire un aspect multi-objectif à leur exploration. Comme mentionné précédemment, les auteurs ont utilisé la plateforme système paramétrable du projet Platune. L'exploration s'est effectée sur le processeur R3000 MIPS, la mémoire cache d'instructions et de données, la mémoire et les bus pour la communication CPU-cache et cache-mémoire. Avec l'ensemble de ces paramètres, l'espace à explorer est de $1,87 \times 10^{11}$ configurations. L'étude a été réalisée sur un ensemble de benchmarks[3] explorés avec Platune et avec la méthode génétique. Les auteurs ont choisi une configuration pour l'algorithme génétique afin d'explorer l'ensemble des applications. L'évaluation de performance s'est basée sur le temps en secondes qu'a mis l'évaluation de chaque benchmark, le nombre de simulations nécessaire par l'approche Platune, le nombre de simulations nécessaire dans l'approche génétique, et la distance moyenne en pourcentage entre l'approche génétique et l'ensemble de solutions Pareto optimales obtenues dans l'approche Platune. Le tableau suivant illustre une partie des résultats obtenus.

[3]Pour la description des Benchmarks voir référence [37]

Benchark	temps d'éval. (Sec)	Sim Platune	Sim	d%	s%
g3fax	2,61	27082	1143	0,88	95,8
jpeg	12,22	15686	1040	1,59	93,4
qurt	0,09	12994	1345	1,67	89,6
bcnt	0,13	22295	1071	1,25	95,2
adpcm	0,34	14442	1089	1,10	92,5
blit	0,17	37474	1024	1,44	97,3
compress	0.54	13707	1273	0,70	90,7
Moyenne	2.3	20526	1136	1,23	93,5

TAB. 2.6 – Résultats d'exploration et comparaison avec Platune, d : Distance moyenne par rapport au front de Pareto optimal, s : Gain en temps de simulation par rapport à Platune

On remarquera qu'en moyenne, avec l'exploration génétique on gagne 93.83% de temps d'exécution tout en gardant une précision correcte (1.23%) assurant une bonne qualité de solutions. Afin d'estimer l'extensibilité de la méthode génétique, les auteurs ont rajouté des paramètres à leur application. Les résultats ont montré que le nombre de configurations à prendre en compte a augmenté d'un facteur 10 alors que pour la méthode génétique il n'a été multiplié que d'un facteur de 1,7. Cette étude met en avant les avantages d'utiliser un algorithme génétique NSGA-II pour l'exploration multi-objectif des plateformes systèmes embarqués.

Un précédent travail de recherche (qui rentrait dans le cadre du travail d'une thèse [38]) s'est effectué au sein du laboratoire LEI (Laboratoire d'Electronique et d'Informatique à l'ENSTA). Cette étude concernait l'exploration de systèmes sur puce. Afin de mettre en évidence la démarche scientifique dans laquelle nous nous sommes positionnés, il est indispensable de revenir au début et aux besoins qui ont impliqué l'exploration architecturale dans notre équipe. Le travail dans [38] concernait un problème d'optimisation mathématique des différents paramètres des IPs soft constituant un système sur puce (SoC). Dans le cadre de ce travail, cela concernait l'architecture de processeurs soft. Le problème d'optimisation a été considéré comme étant un problème multidimensionnel dans le cadre des SoC pour systèmes embarqués. Cela est dû au fait de devoir considérer plusieurs paramètres à la fois, comme : la performance, la consommation d'énergie et la surface en silicium utilisée. Par conséquent l'exploration architecturale (Processeur soft) est ramenée à la résolution d'un problème multi-objectif. Le premier travail effectué dans cette thèse concernait l'exploration pour le dimensionnement d'IP de processeur SuperScalar. La recherche dans l'espace d'exploration s'est faite sous la contrainte de trois fonctions objectifs : la performance, la consommation d'énergie et la surface en silicium. Les résultats obtenus par des benchmarks multimédias " MiBench " de taille significative résultent dans un sous ensemble optimal au sens de Pareto, permettant de sélectionner une ou plusieurs solutions optimales pour les applications cibles. Dans

ce cas, l'exploration concernait les paramètres de la mémoire cache d'instructions, la mémoire cache de données ainsi que d'autres paramètres liés à l'architecture interne d'un processeur soft. Cela constitue la principale différence avec ce qui suit dans le travail de cette présente thèse. Effectivement, en plus des paramètres liés aux processeurs, nous sommes passés à l'optimisation de la communication entre un processeur et ses IP accélératrices puis entre plusieurs processeurs dans le cas d'une plateforme multiprocesseurs. L'objectif est le dimensionnement efficace des FIFO de communication afin d'atteindre un équilibre optimale entre la mémoire utilisée pour le FIFO et la mémoire utilisée pour les caches. La problématique est double car il faut dans ce cas optimiser la performance du processeur et celle de la communication.

Dans [39], les auteurs présentent une comparaison entre quatre différents algorithmes évolutionnaires puis exposent l'algorithme SPEA que nous avons présenté dans la section 2.3 de ce chapitre. Les algorithmes comparés sont 1) VEGA (Vector Evaluated Genetic Algorithm)[40], 2) Aggregation by variable Objective Weighting , 3) NPGA (Niched Pareto Genetic Algorithm) [41] et NSGA présenté plus haut. La comparaison s'est faite sur l'application knapsack appelée aussi problème du sac à dos qui est un problème d'optimisation combinatoire. Les auteurs ont mentionné le fait que tous les algorithmes évolutionnaires multi-objectif présentés surpassent nettement une stratégie aléatoire pure de recherche produisant aléatoirement de nouveaux points dans l'espace de recherche sans exploiter des similitudes entre les solutions. Concernant la comparaison entre algorithmes évolutionnaires les résultats ont clairement montré que le NSGA fournissait le meilleur rendu par rapport aux différents tests réalisés.

Un travail de recherche très intéressant exploitant les algorithmes génétiques est présenté dans [42, 43]. Les auteurs proposent une méthodologie de partitionnement logiciel/matériel basée sur une exploration génétique multi-objectif. L'étude a été réalisée sur une plateforme FPGA englobant un processeur enfoui. Le partitionnement se fait entre le processeur et une unité matérielle (tâches matérielles) implémentée sur les cellules logiques du FPGA, la communication entre les deux modules (matériel et logiciel) se fait à l'aide d'une mémoire bi-ports. Une comparaison a été effectuée avec les résultats obtenus à partir d'un algorithme glouton. Ces résultats représentent l'impact de la taille du FPGA par rapport au temps d'exécution où les auteurs illustres aussi, à travers des graphes, l'évolution du temps d'exécution, de communication, temps "Idle" et enfin temps d'exécution moyens à travers les générations d'exploration. Les auteurs ont démontré à travers cette étude l'utilité des algorithmes génétiques multi-objectif dans l'exploration architecturale et dans ce cas dans le partitionnement logiciel/matériel. Une extension plus complète est donnée dans le projet EPICURE [44].

Dans le cadre de MOGAC [45], à partir des spécifications d'un graphe de tâches, le problème de synthèse d'architectures distribuées hétérogènes temps réel est considéré sous trois objectifs : coût, performance et consommation d'énergie de l'archi-

tecture de cible. Pour réaliser ceci, un algorithme génétique adaptatif d'écrit afin d'éviter de se faire piéger dans des minima locaux à été proposé. La figure 2.23 illustre les étapes d'exécution de l'algorithme MOGAC.

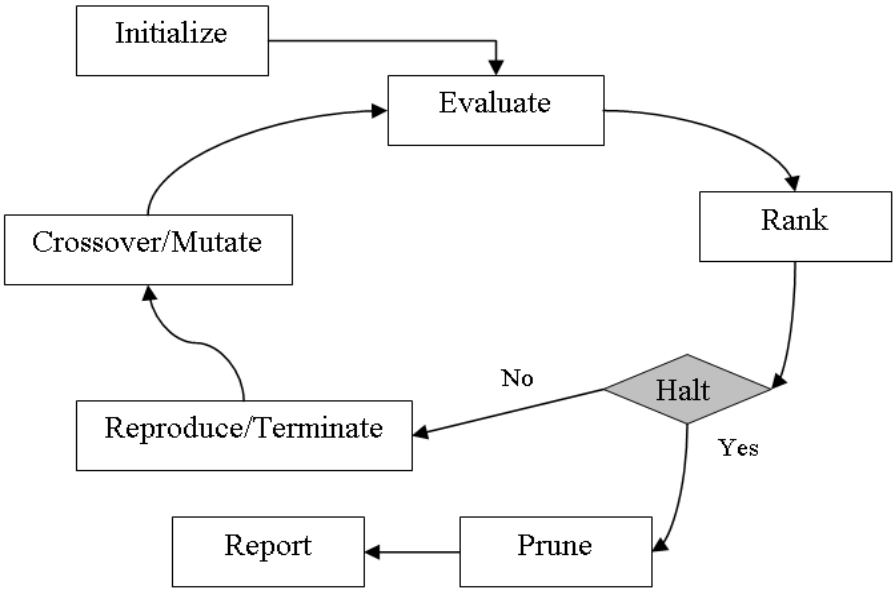


FIG. 2.23 – Algorithme MOGAC

MOGAC ne place aucune limite sur le nombre d'éléments de calcul matériels ou logiciels dans l'architecture à synthétiser. Les auteurs ont utilisé des modèles de bus et de communication point à point afin d'assurer une communication hétérogènes entre les différents éléments de l'architecture. Ils justifient aussi l'utilisation d'algorithmes d'optimisation multi-objectif par le fait que ces derniers fournissent un ensemble de solutions faisant le compromis entre différentes spécificités de la plateforme. Cependant, cette méthodologie exprime un manque par rapport à la gestion de solutions infaisables du fait qu'elle traite toutes les solutions non dominées de la même manière même si celles-ci violent les contraintes. Les individus non valides sont seulement retirés à la fin. La conséquence à cela est que MOGAC peut avoir des problèmes liés à la convergence, c-à-d. convergence vers un ensemble de solutions infaisables.

Afin de comparer les performances de l'algorithme NSGA-II, nous présentons une étude faite dans [40] où les auteurs exposent l'algorithme NSGA-II et le comparent avec d'autres algorithmes évolutionnaires multi-objectif. Dans cet article, les auteurs mentionnent différents points où l'algorithme NSGA-II excelle par rapport aux algorithme NSGA, PAES et SPEA. Ils reprochent principalement à ces algorithmes leur complexité de calcul et le manque d'élitisme à travers leurs explorations. Les résultats obtenus place l'algorithme NSGA-II comme étant le meilleur candidat pour traiter des applications complexes avec de réels problèmes d'optimisation

multi-objectif. Cet algorithme permet d'avoir une bonne estimation des fronts de solutions Pareto optimales sur plusieurs rangs. De plus, l'algorithme permet d'avoir des résultats d'exploration dans des délais assez rapides, comme ce qui a été démontré précédemment dans [45]. La densité de solutions existante autour d'un point, lors d'une exploration, donne aux systèmes embarqués une certaine particularité. Il se trouve que l'algorithme NSGA-II considère très bien cet aspect (crowding distance) le rendant particulièrement adéquat pour l'optimisation architecturale dans les systèmes embarqués.

2.6 Conclusion

Nous avons présenté dans ce chapitre les différentes méthodologies de conception des systèmes embarqués sur puce (SoC), tout particulièrement celles des systèmes sur puces programmables électroniques. Les systèmes sur puce sont de plus en plus performants en capacité d'intégration rendant par voie de conséquences les anciennes méthodologies de conception inadéquates. Effectivement, les systèmes sur puce actuels imposent des méthodologies assurant une exploitation efficace de leurs ressources avec des délais de mise sur le marché courts. Nous avons présenté pour cela les différents outils proposés auparavant et mis en avant leurs inconvénients par rapport aux exigences des nouveaux systèmes sur puce. Nous avons par la suite abordés les différentes techniques d'optimisations et justifié le choix de passer par des algorithmes évolutionnaires. Afin de démontrer l'efficacité de ces derniers, nous nous sommes basés sur des études traitant le paramétrage de plateformes systèmes sur puce. Dans la première étude, l'exploration de l'espace de solutions se fait de manière exhaustive puis par une extraction du front de Pareto. La deuxième utilise l'algorithme génétique NSGA-II. Toujours est il que même en utilisant des algorithmes évolutionnaires le temps que met l'exploration reste important du fait qu'on passe par la simulation. L'idée que nous avons introduite et qui sera illustrée dans les chapitres qui suivent, se base sur la large capacité d'intégration des circuits FPGA actuels et leur simple utilisation afin d'éviter le temps prohibitif de la simulation et réaliser une vérification basée sur une implémentation matérielle dans le flot de la boucle d'exploration.

Chapitre 3

Exploration SoPC monoprocesseur

3.1 Paramètrage d'architecture de communication pour plateforme SoC

Nous abordons dans ce chapitre l'exploration des plateformes embarquées et plus précisément de leurs architectures de communication. Celle-ci constitue une partie essentielle à optimiser afin d'améliorer les performances des systèmes sur puce [46]. Ces systèmes sont constitués de processeurs et de fonctions accélératrices matérielles permettant un gain de performance non négligeable. Cependant, une exploitation optimale des IPs matérielles ne peut se faire qu'avec une communication efficace avec le processeur.

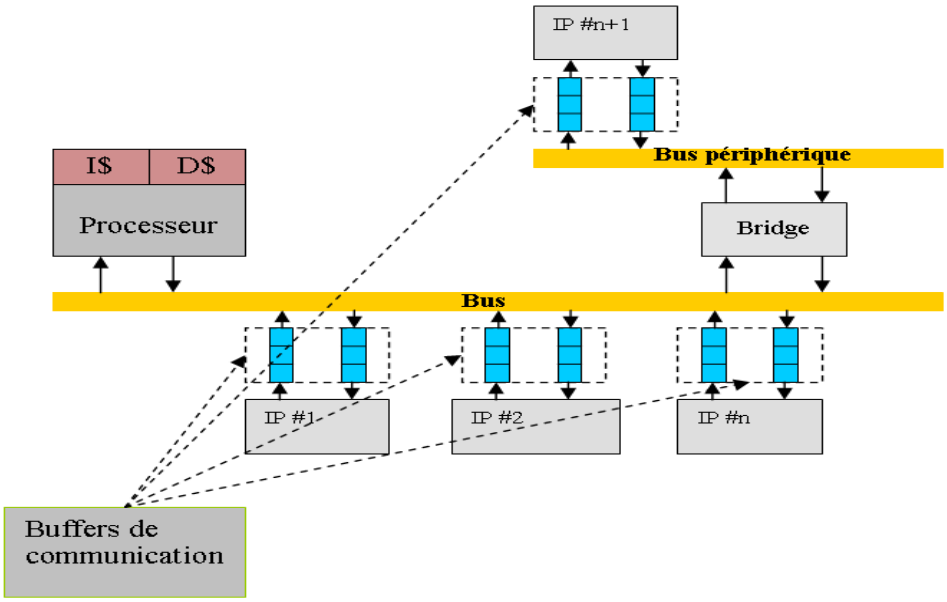


FIG. 3.1 – Système sur puce avec bus et FIFO de communication

Ce problème d'optimisation de la communication existe déjà depuis un certain nombre d'années et il est resté toujours ouvert. D'ailleurs, plusieurs travaux ont été menés tous basés sur des principes différents pour résoudre cette problématique. Cette question a été attaquée à différents niveaux d'abstraction et avec différents algorithmes. La figure 3.1 représente un système sur puce avec des IPs accélératrices sur le bus système et le bus périphérique. Ces IPs sont connectées au bus à l'aide de buffers de communication. Ces buffers permettent de stocker momentanément une donnée avant d'être prise en compte. Ils représentent un point essentiel pour la communication. Un mauvais dimensionnement de leur taille (profondeur, largeur) peut avoir des résultats directs sur la performance du système si leur taille n'absorbe pas efficacement le flux de données. Dans l'autre cas, où les buffers sont surdimensionnés, une mauvaise gestion de la ressource mémoire sur puce peut elle aussi affecter la performance si elle n'est pas allouée aux composants l'exigeant. Des modèles mathématiques ont été proposés afin de décrire une politique de communication [47]. Dans ce cas, la modélisation mathématique de la communication permet de choisir, dans le cas d'un bus, la politique de communication la plus appropriée. En général, l'étude analytique est appuyée par une simulation afin de mesurer les performances. Aussi une conception niveau système permet de générer des modèles fonctionnels de bus à partir des modèles du niveau transactionnel [48, 49]. Ces modèles permettent de procéder à un prototypage de l'architecture du bus systèmes. De plus, le principal avantage lié au fait de travailler à ce niveau d'abstraction nous permet d'avoir une vitesse de simulation avec laquelle on peut tester relativement rapidement différentes architectures de communication et analyser rapidement le trafic des bus.

La conception et l'analyse de l'architecture de communication forment une question cruciale dans la conception de systèmes sur puce [47], et le fait de se diriger vers des systèmes sur puce à l'échelle submicronique, l'incertitude dans la communication devient un facteur qui ne peut pas être négligé. Cela implique directement une adaptation des outils de conception avec une prise en compte pointue de l'implémentation physique. Les études analytiques et les simulations au niveau transactionnel donnent effectivement une idée précise sur le schéma de communication le plus approprié pour une application donnée. Cependant, dans ces études, les contraintes liées à l'implémentation physique ont été négligées ou seulement estimées.

Ce que nous proposons dans ce chapitre est une méthodologie permettant une exploration à base d'algorithmes génétiques et de techniques d'évaluation sur FPGA, de systèmes sur puce programmables afin d'optimiser leur architecture de communication et leur utilisation des ressources sur puce. Nous rappelons que nos deux principaux objectifs sont les suivants :

1. Réduction du cycle de conception (Contraintes time-to-market)
2. Garantir la qualité des résultats

Notre méthodologie n'impose pas de contraintes particulières et peut être appliquée à n'importe quel système sur puce.

Ce chapitre s'attaque à la problématique de communication sur les systèmes sur puce puis valide la proposition de notre méthodologie de conception à travers un exemple pratique.

3.2 Processeurs embarqués et processeurs embarqués soft IPs

Il y a quelques années, on pouvait trouver l'utilisation de processeurs que sur les stations de travail ou bien sur les grandes machines industrielles. A présent, les processeurs sont devenus des éléments essentiels et omniprésents dans toute l'industrie électronique grand publique. De fait, les processeurs sont utilisés dans les appareils électroménagers, l'industrie automobile, et sur tout sur les appareils portables tels que les téléphones et les assistants personnels (PDA : Personal Digital Assistants). Ces processeurs se présentent sous plusieurs architectures selon le besoin. Ils peuvent être de 8 à 64 bits ciblant des applications de traitement de signal numérique (DSP) ou pas. Comme conséquences du progrès dans le domaine de l'intégration sur silicium et dans l'amélioration de l'architecture processeur à travers les différentes techniques proposées ces dernières années (Les mémoires caches, parallélisme d'instruction, prédiction de branchement,...), on peut trouver en ce moment des processeurs très performants consommant un minimum d'énergie leur permettent d'être portables. Ces processeurs embarqués fournissent un rendu excellent concernant le rapport performance sur énergie consommée et de surcroît, ils sont facilement programmables. Il est important de signaler aussi que ces processeurs permettent pour le plus souvent, grâce à leurs performances, d'exécuter des systèmes d'exploitation temps réel, ce qui est un atout non négligeable concernant les applications (Audio, vidéo,..) de communication exigeant de fortes contraintes temps réel. Le tableau suivant récapitule le domaine d'utilisation des processeurs embarqués.

Domaine	Applications Embarquées
Ordinateurs Portables	Laptops Hand-Held PC PDA
Appareils Sans Fils	Téléphones portables GSM
Appareils multimédias	Lecteur MP3-MP4... Caméra digitale
Autres applications	Cartes à puce Badges électroniques

TAB. 3.1 – Domaine d'application des processeurs embarqués

3.2.1 Processeurs embarqués soft IPs

La capacité d'intégration des FPGA permet d'y implémenter des processeurs IPs soft. Ce sont des processeurs décrits à l'aide d'un langage de description matériel (VHDL, Verilog, SystemC...) pouvant être synthétisable. Ces processeurs reprennent exactement le fonctionnement et les performances des autres processeurs implémentés en dur. L'avantage ici réside dans leur capacité d'être paramétrés. Effectivement, ces processeurs peuvent être paramétrés, ouvrant ainsi la possibilité de les adapter à la spécification de l'application qu'on veut implémenter. Le fait de paramétrer un processeur permet d'avoir une flexibilité et une facilité à faire un compromis performance coût accélérant ainsi le cycle de conception. Parmi les produits existant en ce moment sur le marché on peut citer : NIOS/NIOS-II d'Altera [50], Microblaze de Xilinx [51] et Tensilica Xtensa V/LX [52]. En plus des processeurs, ces fournisseurs offrent un ensemble de modules de gestion de mémoire et de communication pour une réalisation rapide de systèmes sur puce programmables.

3.2.1.1 Processeur soft Microblaze

Xilinx a de son côté proposé le Microblaze, un processeur RISC "soft IP" 3 étages avec une architecture Harvard avec des registres internes de 32 bits. Il dispose de bus d'instructions et de données internes et externes. Le Processeur Microblaze, tout comme le Nios, est très facilement configurable occupant selon le choix des options de 900 à 2600 éléments logiques et pouvant fonctionner sur une fourchette de fréquences s'étalant de 80 à $180MHz$. La figure 3.2 illustre l'architecture interne du processeur Microblaze.

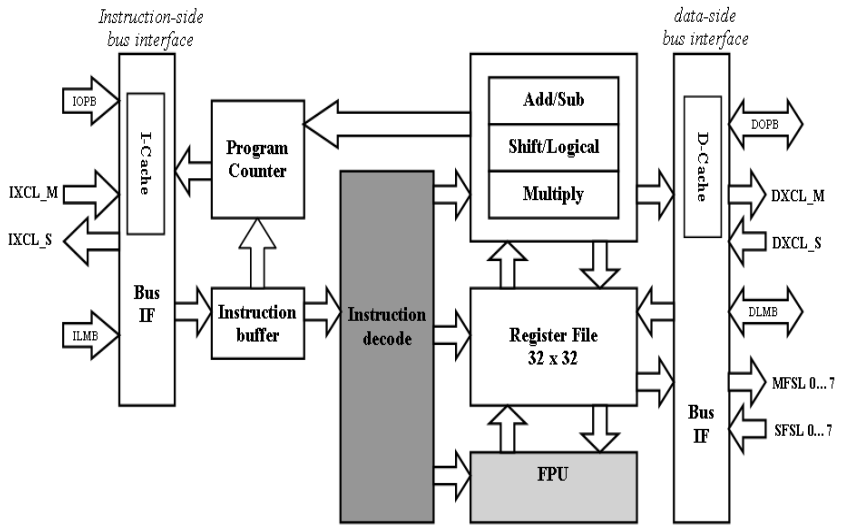


FIG. 3.2 – Architecture interne du Microblaze 4.0

Parmi les options configurables, on peut citer l'utilisation de multiplieurs (mul-

tiplieurs matériels sur FPGA), d'opérateurs de division, d'opérateurs de décalage (barrel shifter), d'unité de calcul flottant (FPU : Floating Point Unit), de mémoires caches d'instructions et de données et d'une unité de debug. Le bus utilisé avec le Microblaze est le bus OPB (On-Chip Peripheral bus). C'est un bus IBM Core-Connect utilisé avec les processeurs PowerPC. Ce bus autorise un maximum de 16 maîtres et un nombre d'esclaves illimité. Il dispose d'une politique d'arbitrage (Bus multi maîtres) paramétrable. Dans un système sur puce programmable à base de processeur Microblaze, le bus OPB est utilisé afin de connecter les périphériques dont les besoins en communication sont faibles. Autrement, Xilinx fournit le lien FSL (Fast Simplex Link) permettant des accès rapides (2 fronts d'horloge) des périphériques vers le Microblaze et vice versa (8 connexions FSL par Microblaze). Un autre type de bus LMB (Local Memory Bus) est utilisé pour accéder aux blocs RAM du FPGA. Dans le Microblaze, ce bus est utilisé pour les instructions et les données et il assure des accès rapides à la mémoire (1 front d'horloge). Dans cette thèse nous avons utilisé le Microblaze afin de mettre en place des plateformes systèmes multiprocesseurs et des réseaux sur puce. De plus amples informations seront fournies dans les chapitres 4 et 5.

3.2.2 Processeur Hard Core "PowerPC405"

Nous allons présenter maintenant la première proposition d'exploration architecturale pour un système sur puce programmable. Le processeur que nous avons utilisé est le powerPC405 d'IBM [53]. Il se trouve que le coeur de processeur matériel powerPC405 est enfoui dans le circuit FPGA Virtex-II Pro de Xilinx. C'est à l'utilisateur de rajouter les périphériques qui sont nécessaires à son fonctionnement. Le PowerPC405 utilisé dans les FPGA Xilinx est différent des PowerPC comme les G4 ou G5 car il ne comporte pas de registres flottants et ne possède pas non plus de module d'exécution d'instructions SIMD. L'architecture du processeur PowerPC405 est représentée par la figure 3.3.

Le powerPC405 est un processeur RISC 32 bits avec une architecture Harvard. Son architecture pipeline est de 5 étages avec une fréquence de fonctionnement de $400MHz$. Il dispose de 32 registres de 32 bits, une mémoire cache d'instructions et de données de 16 KB chacune et d'une unité de gestion de mémoire (Memory Management Unit). Sur les circuits FPGA Xilinx supportant le processeur PowerPC, l'accès aux mémoires internes (Blocs RAM du FPGA) se fait à travers une unité "On-Chip memory Controllers".

Le processeur PowerPC 405 ainsi que le Microblaze sont exploitables grâce à l'outil EDK de Xilinx. Cet outil est présenté dans la section 3.3 qui va suivre.

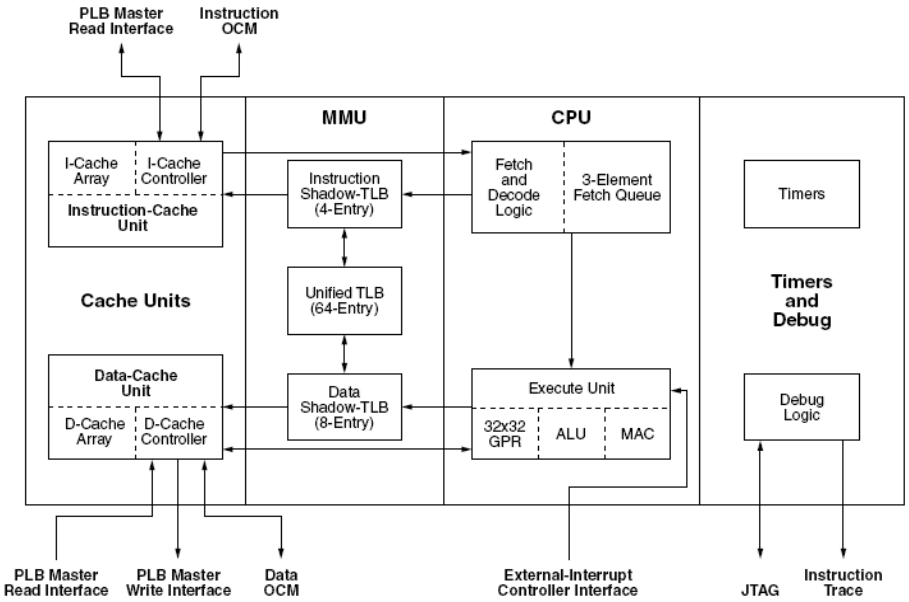


FIG. 3.3 – Architecture interne du PowerPC405

3.3 Outils de conception Xilinx

3.3.1 Flots de conception Xilinx

Pour la création et la vérification d'un système embarqué, Xilinx propose un ensemble d'outils qui sont regroupés dans deux grands logiciels qui sont ISE (integrated Software Environment) et XPS (Xilinx platform studio) [54]. Les outils de ISE sont utilisés par XPS.

La conception d'un système embarqué inclut typiquement les phases suivantes :

- Création de la plateforme matérielle
- Vérification de la plateforme matérielle
- Création de la plateforme logicielle
- Création et vérification de l'application logicielle

La plateforme matérielle est définie par le fichier MHS(Microprocessor Hardware Specification). C'est la connexion d'un ou plusieurs processeurs et périphériques sur les bus de ce dernier. L'utilisateur peut définir et ajouter ses propres périphériques. L'outil XPS fournit des moyens graphiques pour la création du fichier MHS. Le fichier MHS définit l'architecture, les connexions et le mapping des adresses du système. A partir du fichier MHS, l'outil Platform Generator (PlatGen) crée la netlist du système sous différents formats (NGC, EDIF) et les wrappeurs HDL des niveaux TOP pour que l'utilisateur puisse intégrer ses composants au système. Suite à l'application de PlatGen, les outils ISE sont utilisés pour compléter l'implémentation du système. La création d'une plateforme matérielle est résumée dans la figure 3.4 :

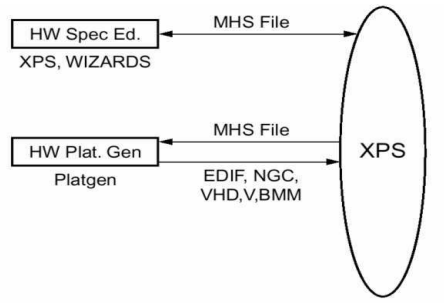


FIG. 3.4 – Création d'une plateforme matérielle

La plateforme de vérification nécessite la plateforme matérielle. Elle permet à l'utilisateur de définir le modèle de simulation pour chaque composant du système (processeur et périphériques). L'outil SimGen crée le fichier de simulation (HDL ou d'autres modèles compiler) à partir du MHS, ainsi que les fichiers de commandes. Si l'application logicielle est disponible sous sa forme exécutable, elle peut être utilisée pour initialiser les mémoires. La figure 3.5 représente la plateforme de vérification.

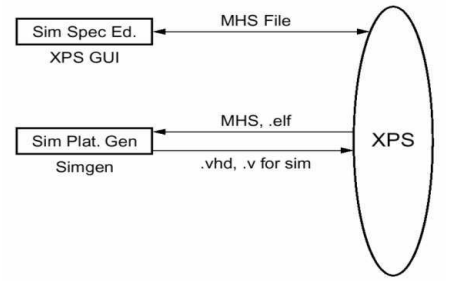


FIG. 3.5 – Vérification d'une plateforme matérielle

La plateforme logicielle est définie par le fichier MSS (Microprocessor Software Specification). Ce fichier regroupe les drivers et les bibliothèques pour l'adaptation des paramètres des périphériques et ceux des processeurs, les routines d'interruption et les composants d'entrées/sorties. Le fichier MSS est utilisé par l'outil Library Generator (LibGen) pour l'adaptation des drivers et bibliothèques et les routines d'interruption. Le processus de la création de la plateforme logicielle est donné dans la figure 3.6 :

La création et la vérification d'une application logicielle passe par différentes étapes : d'abord l'écriture du code en C, C++ ou assembleur qui va être exécuté sur les plateformes logicielles et matérielles. Ensuite ce code est compilé et linké à l'aide

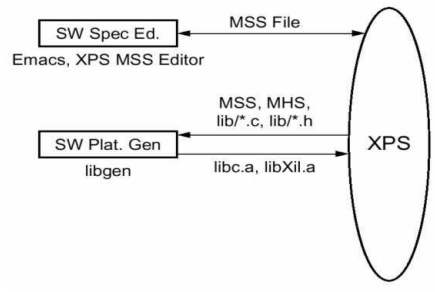


FIG. 3.6 – Création d'une plateforme logicielle

de l'outil GNU (d'autres outils peuvent aussi être utilisés) pour générer le fichier exécutable sous le format ELF (Executable and Link Format). Finalement, XMD et le débogueur de GNU (GDB) sont utilisés pour déboguer l'application. La figure 3.7 représente le processus de création et de vérification d'une application logicielle.

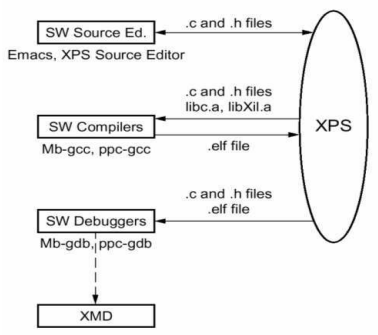


FIG. 3.7 – Création et vérification d'une application logicielle

Les processus supportés par l'outil XPS sont principalement :

- Création du fichier MHS
- Création du MHS
- Réalisation de la matrice de connexion des différents bus
- L'adaptation des drivers, des bibliothèques et des contrôleurs d'interruptions
- La gestion des fichiers sources

Les détails de ces différents processus sont représentés dans la figure 3.8 :

Le tableau 3.2 représente les différents types de fichiers utilisés dans l'outil EDK de Xilinx.

Les tableaux 3.3 et 3.4 représentent des exemples de fichiers MHS.

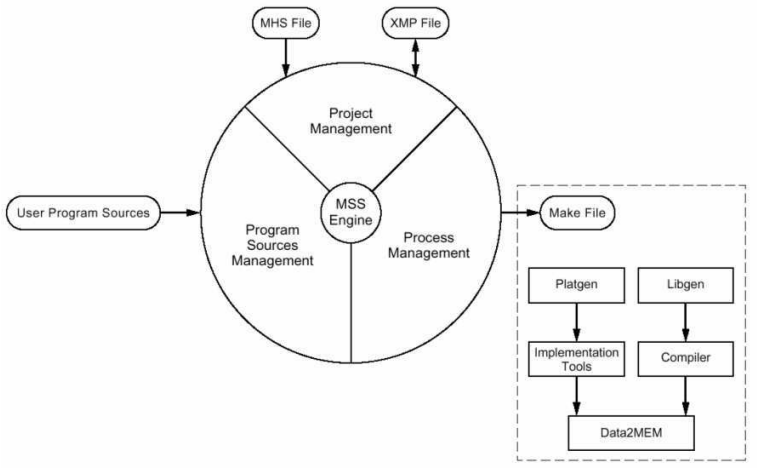


FIG. 3.8 – Les Processus supportés par XPS

Fichier	Description	Commentaires
MHS	Microprocessor Hardware Specification	Définit les composants matériels
MSS	Microprocessor Software Specification	Définit les bibliothèques logicielles et les drivers
MDD	Microprocessor Driver Definition	Permet d'adapter les drivers logiciels
MDD	Microprocessor Peripheral Definition	Définit les interfaces des périphériques
MLD	Microprocessor Library Definition	Contient des directives pour configurer la bibliothèque logicielle et l'OS
PAO	Peripheral analyze Order	Contient la liste des fichiers HDL et leur ordre de compilation

TAB. 3.2 – Les fichiers EDK

3.4 Cas d'étude

3.4.1 Plateforme d'exploration système sur puce monoprocesseur

L'architecture sur laquelle nous avons effectué notre étude d'exploration de système sur puce monoprocesseur est présentée dans la figure 3.9. Celle-ci est basée sur une plateforme FPGA avec un circuit Virtex-II Pro 2VP20 de chez Xilinx. Ce composant, comme expliqué dans le deuxième chapitre de cette thèse, est doté d'un coeur de processeur PowerPC 405, enfoui au milieu des cellules logiques du FPGA. Nous avons rajouté au processeur un bus local (PLB : Processor Local Bus), un bus de périphériques (OPB : On-chip Peripheral Bus), un pont (PLB2OPB Bridge) pour passer du bus PLB au bus OPB. Comme périphériques, nous avons utilisé

BEGIN MicroBlaze	BEGIN fsl_v20 BEGIN
PARAMETER INSTANCE = MicroBlaze 0	PARAMETER INSTANCE = fsl_v20_7
PARAMETER HW VER = 3.00.a	PARAMETER C FSL DEPTH = 8
PARAMETER C FSL LINKS = 2	PARAMETER HW VER = 2.00.a
BUS INTERFACE MFSL0 = fsl v20 2	PARAMETER C EXT RESET HIGH = 0
BUS INTERFACE SFSL0 = fsl v20 1	PARAMETER C IMPL STYLE = 1
BUS INTERFACE DLMB = dlmb0	PARAMETER C USE CONTROL = 0
BUS INTERFACE ILMB = ilmb0	PORT SYS Rst = lresetol
BUS INTERFACE DOPB = mb opb0	PORT FSL Clk = lclk
BUS INTERFACE IOPB = mb opb0	PORT FSL M Clk = lclk
PORT INTERRUPT = Interrupt 0	PORT FSL S Clk = lclk
PORT CLK = lclk	END
END	

TAB. 3.3 – Exemple MHS Microblaze et FSL

lmb bram if cntlr
PARAMETER INSTANCE = ilmb cntlr3
PARAMETER HW VER = 1.00.b
PARAMETER C BASEADDR
= 0 .. 00000000
PARAMETER C HIGHADDR
= 0 .. 00003fff
BUS INTERFACE SLMB = ilmb3
BUS INTERFACE BRAM PORT
= ilmb port3
END

TAB. 3.4 – Exemple BRAM

une IPs UART (Universal Asynchronous Receiver Transmitter) sur le bus OPB et trois IPs sur le bus PLB sur lesquelles s'est effectuée l'exploration. Ces IPs représentent des fonctionnalités de traitement d'images où le premier traitement consiste en un filtre médian, le deuxième est un filtre conservatif et le dernier représente une fonctionnalité de la chaîne de compression JPEG2000 qui est le codage entropique.

Ces trois composants ont été connectés au bus PLB grâce à l'IP IPIF (IP Interface) fournie par Xilinx. Cette fonctionnalité (expliquée dans le deuxième chapitre de cette thèse) permet de simplifier la connexion des IPs utilisateur au bus PLB en offrant plusieurs choix et options d'interfaçage. Parmi ces choix, on peut citer le fait de pouvoir choisir entre communiquer avec l'IP à travers des registres (si le flot de données n'est pas dense) ou à travers des FIFO d'écriture et de lecture. Dans notre cas, le choix s'est porté sur la deuxième option. Effectivement, nous avons choisi une application de traitement d'images où le flot de données est très dense et justifie une communication à base de FIFO entre le processeur et les IP accélératrices. La problématique est de paramétrer de manière optimale les différentes tailles de FIFOs pour avoir le meilleur compromis performance et ressources utilisées ainsi qu'une utilisation équilibrée de la mémoire interne du FPGA entre les différentes FIFOs de communication. Ce point représente l'objet de l'étude.

Dans les sections suivantes, nous présentons plus en détail les trois différentes IPs implémentées dans le cas de cette étude.

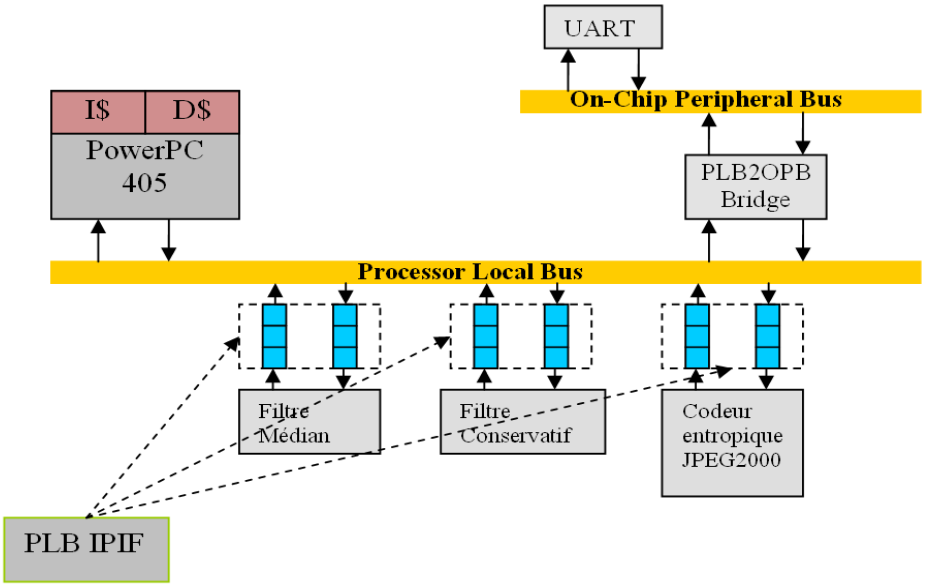


FIG. 3.9 – Architecture exploration Monoprocesseur sur Virtex-II Pro

3.4.2 Filtre médian

Ce filtre calcule la valeur médiane sur une fenêtre de 9 pixels. Obtenir cette valeur médiane implique la réalisation de nombreuses comparaisons. Une implémentation possible est d'appliquer un algorithme de tri à la fenêtre de 9 pixels et de prendre le cinquième élément du tableau trié, qui sera finalement la valeur médiane. La valeur du pixel courant sera alors remplacée par la valeur médiane. Le nombre d'opérations par pixel varie alors selon l'algorithme de tri, mais est quoi qu'il en soit plus élevé que pour un filtre conservatif ou moyenneur.

3.4.3 Filtre conservatif

Le filtre conservatif est quant à lui non linéaire. Il ne fait de calcul que sur les 8 voisins du pixel courant. Il extrait le maximum et le minimum du voisinage puis remplace le cas échéant le pixel courant par le minimum ou le maximum calculé. Une implémentation canonique de ce filtre réalisera donc dans le pire des cas 16 comparaisons et 15 affectations, et dans le meilleur des cas 8 comparaisons et 8 affectations.

3.4.4 Standard JPEG2000

Nous avons choisi de faire valider notre concept, en plus des filtres présentés ci-dessus, sur le codeur entropique du standard de compression d'images fixes JPEG2000 [55]. La particularité du codeur entropique JPEG2000 est qu'il reçoit

une quantité de données variable pour la même image traitée. Nous ne pouvons pas savoir au préalable le nombre de données envoyées du module de découpage en codes blocs au codeur entropique. Par conséquent, nous ne pouvons pas prévoir la profondeur de FIFO de communication qu'il faut avoir. Cet exemple permet de tester significativement notre méthodologie d'exploration d'architecture SoC. Une implémentation sur FPGA du codeur entropique a été réalisée. Nous donnons dans ce qui suit une brève explication sur le fonctionnement algorithmique du codeur entropique JPEG2000.

3.4.5 Codeur Entropique JPEG2000

Le codeur entropique constitue la partie réalisée et implémentée sur la plateforme FPGA Virtex-II Pro pour réaliser l'exploration architecturale. Cette partie représente le goulot d'étranglement de la chaîne de compression JPEG2000 du fait qu'elle exige énormément de ressources pour son exécution. Le codeur entropique se compose de deux parties principales la première est la modélisation binaire des coefficients, la deuxième est le codage arithmétique [56, 57].

Le codage entropique de JPEG2000 se compose des étapes suivantes :

- Création des code-blocs
- Conversion de la représentation des valeurs des coefficients en valeur absolue/signée
- Modélisation du contexte
- Codage arithmétique

La deuxième est la partie où se fait le codage. Nous présentons dans ce qui suit ces deux parties avec plus de détail.

3.4.5.1 Modélisation des coefficients

Nous partons d'un ensemble de codes blocs (après la transformée en ondelettes [58]), dont les échantillons sont ordonnés en Bit Planes [59, 60]. L'arrangement de ces plans se fait à partir des bits de poids fort. Par la suite chaque code bloc doit être codé. Pour cela, on fait passer les blocs par trois passes :

1. la passe de signification (Significance Pass) : codage des bits (0 ou 1) des échantillons non significatifs ayant au moins 1 voisin significatif plus éventuellement transformation d'un échantillon non significatif en significatif dans le cas du codage d'un bit à 1 (il s'agit alors du premier bit à 1 pour cet échantillon).
2. la passe d'affinage (Refinement Pass) : codage des bits (0 ou 1) des échantillons significatifs non encore codés
3. la passe de nettoyage (Cleanup Pass) : codage du reste des bits des échantillons non significatifs (0 ou 1). Si on code un bit à 1, on fait passer l'échantillon à l'état significatif. Cette passe contient aussi un mécanisme de codage par plage permettant de coder les séquences consécutives de zéros (Run-length coding).

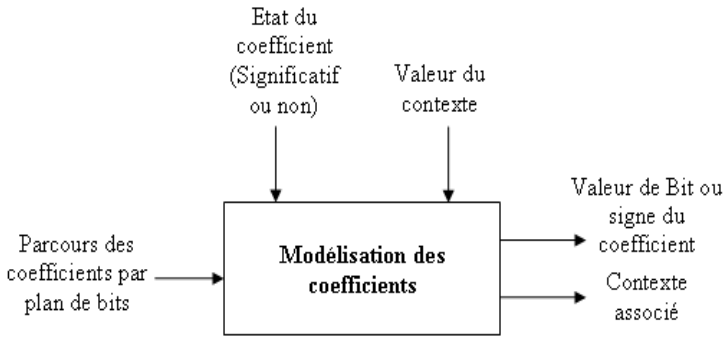


FIG. 3.10 – Modélisation des coefficients

La modélisation des coefficients se fait sur tous les codes blocs de l'image.

3.4.5.2 Codage arithmétique

Le codeur MQ proposé par IBM a été repris et utilisé dans la chaîne de compression JPEG2000. Il est connecté directement à la sortie du bloc précédent, en l'occurrence la modélisation des coefficients. Par conséquent, il prend comme entrées les valeurs binaires et les contextes associés en respectant l'ordre de passage de codage.

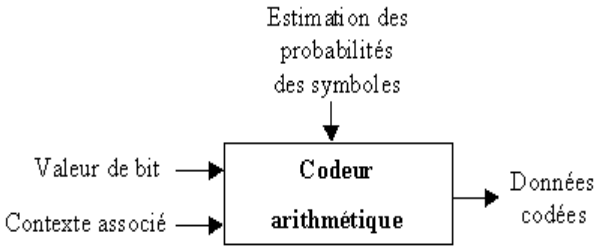


FIG. 3.11 – Codage arithmétique

Dans ce codeur, il a été choisi de représenter les données à l'aide des symboles LPS (Less Probable Symbol) et MPS (More Probable Symbol). Ces symboles représentent respectivement, la probabilité d'occurrence des données minoritaires et majoritaires. Cela se fait en gardant l'information sur la valeur 0 ou 1 minoritaire ou majoritaire. Ainsi l'intervalle courant est représenté par l'intervalle I que l'on divise alors en deux sous intervalles correspondant aux espèces minoritaire et majoritaire. D'un point de vue représentation, on donne toujours comme intervalle inférieur le LPS. Dans ce codeur entropique les hypothèses suivantes sont à respecter :

- $I - Qe$ est la longueur du sous intervalle pour le MPS
- Qe est la longueur du sous-intervalle pour le LPS

– Où Qe est la probabilité de l'espèce minoritaire

Cette approximation est vraie si I est maintenu dans un intervalle de représentation proche de 1. Ceci est réalisé notamment par des phases d'expansion. Le mot de code C généré est alors un pointeur qui pointe sur la base de l'intervalle courant. Ainsi si le bit suivant appartient à l'espèce majoritaire, en notant Qe la probabilité du LPS, alors il vient $C = C + Qe$.

Le pointeur pointe alors la base du sous intervalle correspondant au MPS dans l'intervalle courant. Si le bit est l'espèce minoritaire, alors le pointeur reste inchangé. MPS, LPS et Qe sont évidemment des fonctions du contexte.

3.5 Flot de conception automatique proposé

Nous avons défini dans ce travail une exécution automatique du flot de conception EDK (La Figure 3.12 représente le Flot EDK standard). Ceci nous permet la réalisation de systèmes embarqués autour du processeur PowerPC405 du circuit Virtex II Pro et aussi de lancer un ensemble de configurations successivement en changeant à chaque fois les paramètres pour les différents modules de notre architecture. Le fait d'itérer plusieurs exécutions avec différentes configurations nous permet de chercher le meilleur compromis entre les modules de notre architecture et d'avoir à la fin un ensemble de solutions optimales (Configurations optimales de la plateforme de départ). Nous avons pour cela écrit un script qui nous aide à choisir à chaque fois les composants de notre architecture et les paramètres qui lui sont associés. Nous récupérons en retour les informations de nombre de cellules logiques occupées, nombre de BRAMs, fréquence et enfin le nombre de cycle pour chaque configuration lancée à partir du script.

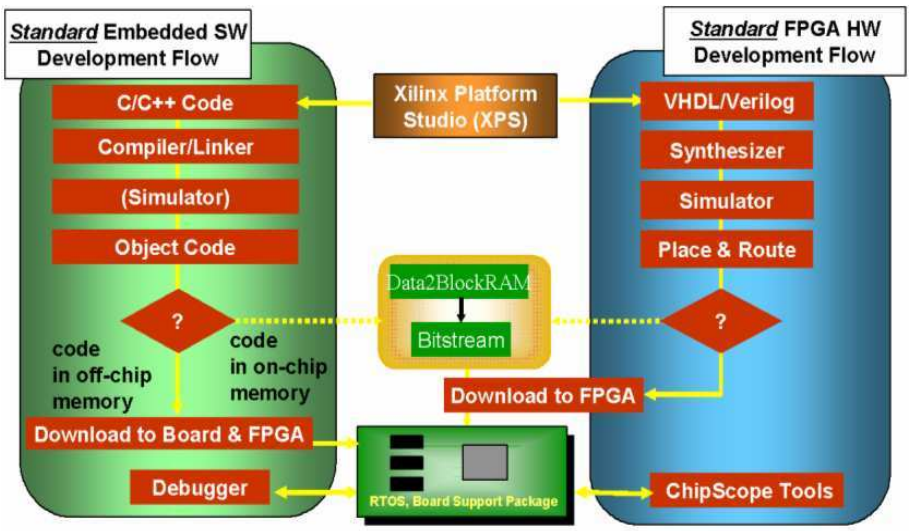


FIG. 3.12 – Flot EDK

Le programme que nous avons défini pour l'exécution automatique du flot EDK reprend toutes les commandes lignes de ISE pour la synthèse matérielle ainsi que toutes les commandes ligne XPS pour la jonction des deux flots (Logiciel et Matériel). Ce programme commence par une lecture du fichier script afin de fixer dès le départ le nombre de configurations à exécuter (si on se trouve dans une exploration exhaustive). Dans un second temps, il prend les paramètres relatifs à la configuration qui est en cours et redéfinit le "Microprocessor Hardware Specification" (MHS) pour chaque composant et le "Microprocessor Software Specification" (MSS) associé. Après avoir fixé ces deux fichiers d'entrée, on passe à la génération de la plateforme de notre système à l'aide de la commande PLATGEN : celle-ci crée les fichiers VHDL wrappés pour chaque composant de notre architecture en fixant dans leurs paramètres génériques les variables à partir du script d'exploration puis exécute une synthèse de tous ces fichiers avec l'outil de synthèse XST de Xilinx. Nous obtenons en sortie les netlists associées aux composants de notre conception. Une fois la synthèse de la spécification matérielle achevée, on se dirige vers son implémentation. Cette partie est constituée de trois principales étapes qui sont : le NGDBuild (Translation), le MAP (Mapping) et enfin le PAR (Place and Route). Dans la première étape d'implémentation (NGDBuild), on effectue une lecture des différentes Netlist à l'aide de la commande NGDBuild qui va par la suite créer un fichier NGD décrivant chaque Netlist relative à un composant de manière logique avec des éléments tels que des portes AND, OR, décodeurs, flip-flop et RAMs. Le fichier de sortie NGD (Native Generic Database) peut être mappé sur n'importe quel circuit FPGA.

Dans notre cas, c'est le circuit Virtex II Pro. Dans la deuxième étape (MAP) de la partie implémentation, on mappe notre architecture sur un FPGA Xilinx avec la commande MAP. L'entrée pour cette commande est le fichier NGD qui contient, comme cité précédemment, la description logique de notre Netlist. Le MAP commence par l'exécution d'un DESIGN RULE CHECK dans le fichier NGD puis mappe la logique sur le composant (cellules logiques, entrées/sorties). Nous obtenons en sortie un fichier NCD (Native Circuit Description) qui est une représentation physique de l'architecture mappée sur le composant. Enfin, on passe à la dernière étape (PAR) de l'implémentation qui consiste à placer et à router l'architecture avec la commande ligne PAR. Cette commande prend en entrée le fichier de description physique NCD ainsi que les différents paramètres liés au type de placement routage que nous voulons effectuer. En sortie, on aura un fichier NCD placé routé prêt à être utilisé pour la génération du Bitstream. Le Bitstream représente un fichier de configuration englobant l'architecture synthétisée à implémenter sur FPGA. Après avoir obtenu le fichier NCD (Native Circuit Description) placé et routé, on exécute la commande ligne BITGEN qui va nous créer un Bitstream pour les composants Xilinx reconfigurables. Cette commande prend comme entrée le fichier placé routé NCD. Toutes les étapes (Synthèse, implémentation, génération Bitstream) qu'on vient d'expliquer précédemment reprennent le flot de développement matériel standard pour les

FPGA Xilinx.

Après avoir défini et généré la partie matérielle de l'application, on passe à l'exécution du flot de conception logiciel nous permettant de compiler le code C ou C++ à exécuter sur le processeur PowerPC405. Pour cela, nous avons introduit dans le flot automatique les commandes lignes assurant l'exécution automatique de ce flot logiciel. Ce flot se décompose en deux étapes : génération des bibliothèques du système et compilation du code C (code objet et link) pour l'obtention d'un fichier exécutable sur PowerPC405. Dans la première étape du flot de conception, on lance l'exécution de la commande ligne nous permettant la mise en place des bibliothèques (drivers) pour les différents modules communiquant avec le processeur PowerPC405. Cela se fait à l'aide de la commande LIBGEN. Les différentes bibliothèques sont décrites dans notre projet sous forme de fichiers .h appelés par le code C principal qui les utilisera pour accéder aux fonctions d'une IP connectée au processeur. Une fois les bibliothèques relatives à notre projet générées, on passe à l'exécution de la deuxième étape permettant la compilation du code C pour le PowerPC405. La compilation se fait avec l'outil GCC. Dans un premier temps, nous fournissons en sortie un code objet puis dans un second temps, nous passons à la création de liens en spécifiant l'adresse de début de notre mémoire et en donnant les différents découpages pour l'adressage (plage de données, plage d'instructions).

Enfin on obtient le fichier exécutable sur PowerPC405 avec une extension .elf. Nous avons alors exécuté un flot matériel et un flot logiciel pour le PowerPC405. Ces deux derniers nous ont fourni respectivement un fichier Bitstream relatif à l'architecture de notre plateforme et un fichier "executable.elf" s'implémentant sur le PowerPC405 du circuit Virtex II Pro. Il faut maintenant faire la jonction entre ces deux fichiers afin d'avoir un seul fichier capable de s'implémenter directement sur une cible Virtex II Pro. Cette jonction du bistream avec le code executable se fait dans le flot de conception avec la commande ligne BITINIT. Cette commande sert aussi à l'initialisation de la mémoire du PowerPC405. On se retrouve alors avec un fichier .bit implémentant tout notre système. Ce fichier sera chargé sur une carte ALPHA-DATA sur port PCI munie d'un circuit VIRTEX II Pro 2VP20 ff896. A partir du flot de conception automatique, on lance notre exécution et on récupère en retour le nombre de cycles, le nombre de slices, nombre de blocs et enfin la fréquence. Ces résultats sont sauvegardés dans un fichier résultats pour chaque configuration lancée.

3.5.1 Carte FPGA ADM-XRC II

3.5.2 Présentation de la carte ADM-XRC-II

La plateforme ADM-XRC-II [61] utilise un FPGA de type Xilinx Virtex-II Pro 2VP20. Le Virtex-II Pro possède des entrées/sorties compatibles pour les interfaces LVDS et le PCI. Le boîtier utilisé pour le Virtex-II Pro dans cette plateforme est

du type ff896. La famille des FPGA Virtex-II possède d'énormes avantages. Parmi eux la fréquence à laquelle peuvent être implémentées les applications, l'espace et la mémoire fournis et la possibilité d'effectuer une reconfiguration dynamique partielle. Cette particularité nous permet d'éviter de re-synthétiser une application lorsque le changement ne s'effectue que sur une partie de l'architecture.

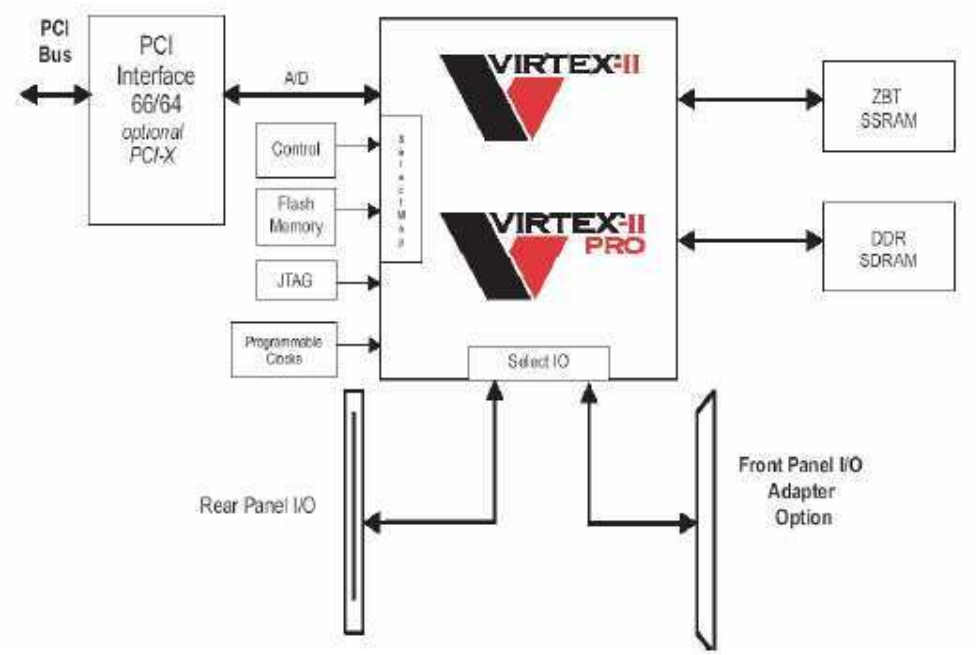


FIG. 3.13 – Carte FPGA Alpha Data

Par ailleurs, la carte ADM-XRC II se connecte sur le bus PCI d'un ordinateur via un connecteur PMC. Cela permet d'avoir une configuration rapide du FPGA ainsi qu'une communication cadencée à la vitesse du bus PCI. Les drivers et bibliothèques permettant d'exploiter les différentes fonctionnalités de la carte sont fournis, permettant par conséquent au PC de communiquer avec le FPGA sous les environnements Linux, Windows ou VxWorks.

3.5.3 Spécifications techniques du module ADM-XRC-II

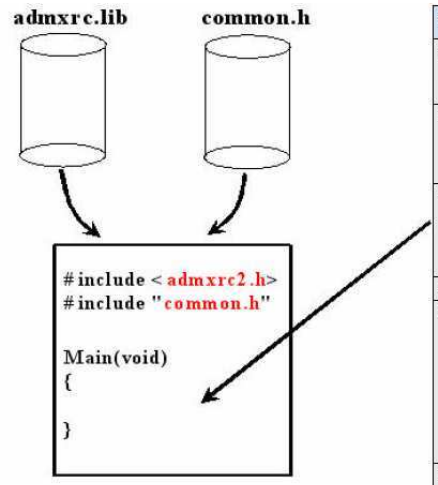
Le module ADM-XRC-II supporte les hautes performances du bus PCI à l'aide d'une interface PCI matérielle PLX 9656 [62] qui gère les communications entre le FPGA et le PCI. Cette carte permet aussi d'avoir accès à une horloge programmable (1 MHz à 100 MHz) pour le FPGA.

Les caractéristiques techniques de la carte sont :

- Bus PCI 64bits - 66 Mhz
- 6 bancs indépendants de ZBT SSRAM 256K x 32bits

- 64 entrées/sorties via un connecteur PMC
- 146 entrées/sorties via un connecteur XRM
- Une autre horloge programmable fournit une fréquence au FPGA : 1 à 100 Mhz
- Possibilité de connecter 256 MO de DDR SDRAM via le connecteur XRM

La figure 3.14 représente l'ensemble des fonctions fournies afin d'exploiter la carte ADM-XRC-II.



Group	Consists of...
Initialization	ADMXRC2_CloseCard ADMXRC2_OpenCard ADMXRC2_OpenCardByIndex ADMXRC2_SetSpaceConfig
Information	ADMXRC2_GetBankInfo ADMXRC2_GetCardInfo ADMXRC2_GetSpaceConfig ADMXRC2_GetSpaceInfo ADMXRC2_GetVersionInfo
FPGA configuration	ADMXRC2_ConfigureFromBuffer ADMXRC2_ConfigureFromBufferDMA ADMXRC2_ConfigureFromFile ADMXRC2_ConfigureFromFileDMA ADMXRC2_LoadBitstream ADMXRC2_UnloadBitstream
Clock generation	ADMXRC2_SetClockRate
Data transfer	ADMXRC2_BuildDMAModeWord ADMXRC2_DoDMA ADMXRC2_DoDMAImmediate ADMXRC2_MapDirectMaster ADMXRC2_Read ADMXRC2_ReadConfig ADMXRC2_SetupDMA ADMXRC2_SyncDirectMaster ADMXRC2_UnsetupDMA ADMXRC2_Write ADMXRC2_WriteConfig
Interrupt handling	ADMXRC2_RegisterInterruptEvent ADMXRC2_UnregisterInterruptEvent
Error handling	ADMXRC2_GetStatusString ADMXRC2_InstallErrorHandler ADMXRC2_StatusToString

FIG. 3.14 – Drivers pour carte FPGA

Le code source utilisateur sensé communiquer avec le FPGA doit être compilé avec les bibliothèques fournies par le constructeur.

3.5.4 Exploration exhaustive

Nous avons effectué une première exploration de manière exhaustive. Les modules de l'architecture à explorer concernaient les FIFOs de communication des différentes IPs (Filtre conservatif, filtre médian et le codeur entropique JPEG2000) connectées au bus PLB et sensées communiquer avec le processeur PowerPC405. L'autre paramètre à explorer concerne l'utilisation ou non de la mémoire cache de données du processeur. Le tableau 3.5 résume les différente IPs pouvant être utilisées dans le flot de conception proposé ici.

Afin de permettre, par la suite, une comparaison des résultats obtenus par la méthode exhaustive et ceux obtenus par l'exploration génétique multi-objectif, nous avons limité la recherche aux FIFOs en Entrées du bus PLB vers les IPs accélératrices. Il n'y a donc que 512 configurations à réaliser. Si on avait gardé les FIFOs

Modules	Choix d'utilisation	Cache Données
PowerPC 405	Non (Utilisation obligatoire)	
PLB	Oui	Largeur de bus 32,64
Bloc RAM	Oui	8K, 16K, 32K, 64K
UART	Oui	9600, 19200, ...
DCM	Oui	CLKMultiply, CLKDivide
IPIF WRFIFODepth	Oui	2...2048
IPIF RDFIFODepth	Oui	2...2048
IPIF WRFIFOwidth	Oui	32, 64
IPIF RDFIFOwidth	Oui	32, 64

TAB. 3.5 – Modules paramètrables dans la plateforme proposée

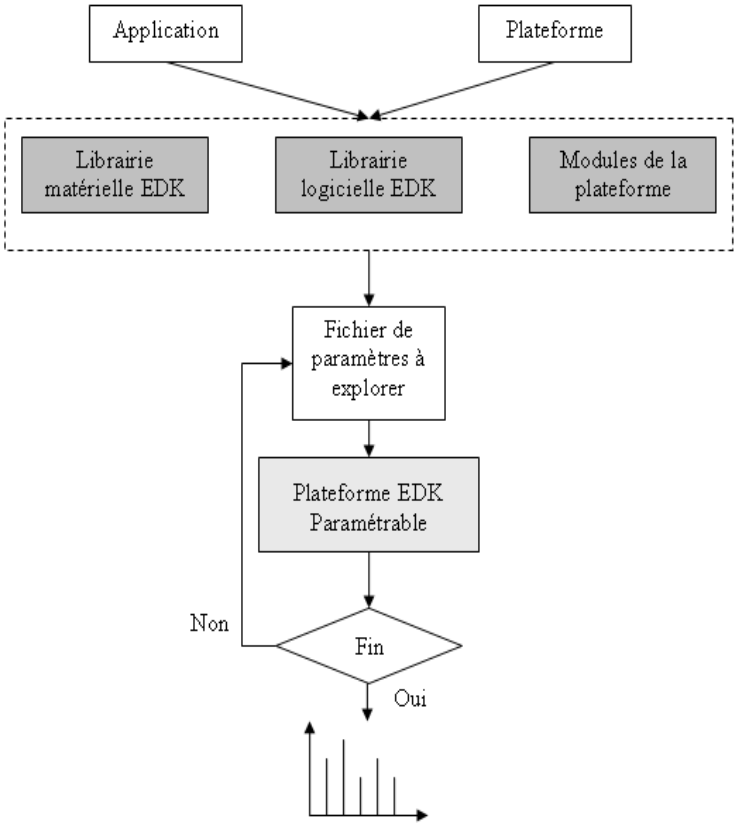


FIG. 3.15 – Flot de l'exploration exhaustive

en entrées et en sorties, l'exploration exhaustive aurait été irréalisable (39,9 années pour l'ensemble des cas).

Dans notre cas, nous avons fait varier la profondeur des FIFOs de l'IPIF de 16 à 2048 cases et gardé la largeur à 32 bits. Nous avons lancé une première exploration

Paramètres	Valeurs
Utilisation du cache de données	0, 1
Profondeur FIFO écriture IP1	16, 32, 64, 128, 256, 512, 1024, 2048
Profondeur FIFO écriture IP2	16, 32, 64, 128, 256, 512, 1024, 2048
Profondeur FIFO écriture IP3	16, 32, 64, 128, 256, 512, 1024, 2048

TAB. 3.6 – Paramètres appliqués à la plateforme

avec 125 configurations. Les résultats relatifs à cette recherche sont présentés dans la figure 3.16.

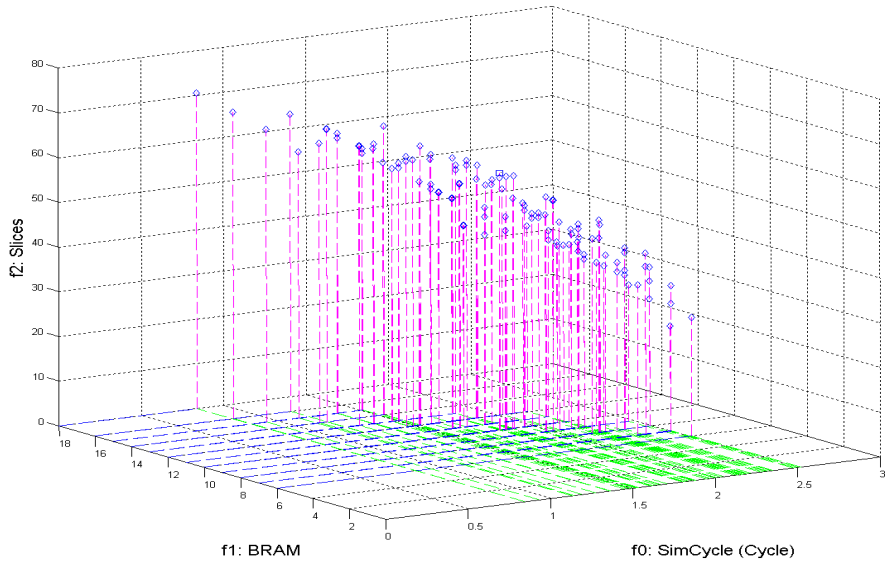


FIG. 3.16 – Exploration exhaustive 125

Une deuxième exploration a été lancée avec dans ce cas-ci la totalité des configurations possibles (512 configurations). La figure 3.17 illustre les résultats de la deuxième exploration.

On remarquera que les résultats obtenus, dans les deux explorations, sont trop rapprochés ce qui rend leur exploitation difficile. Le but d'une exploration est de montrer les résultats sur l'étendue de l'espace de solutions possibles. Nous rappelons que nous cherchons à utiliser de manière optimale les ressources du FPGA. Nous partons d'une application ayant un taux de communication important entre le processeur et ses fonctions accélératrices. Ces données transitent par des FIFOs. Dans notre application, les FIFOs sont réalisées à base des blocs RAM fournis par le circuit FPGA. Le problème est amplifié du fait que les FIFOs ne présentent pas les seuls éléments dans l'application à utiliser les ressources en mémoire. La mémoire cache de données en utilise aussi. Les RAM et l'espace sur puce sont des ressources

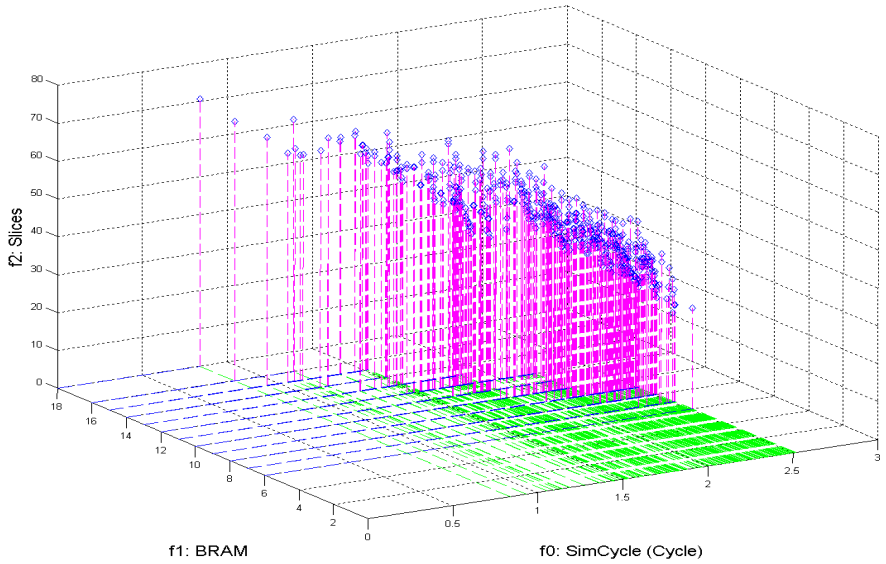


FIG. 3.17 – Exploration exhaustive 512

précieuses. L'objectif à atteindre est de rééquilibrer l'utilisation des RAM entre les différents modules afin d'avoir une exploitation efficace des ressources. Nous définissons ici une exploration qui nous fournira à chaque configuration le nombre de cycles ainsi que le nombre de ressources consommées en slices et Blocs RAM sur FPGA. Dans ce genre d'exploration, nous ne disposons d'aucune règle afin de diriger la recherche dans la direction des solutions susceptibles de répondre aux exigences de l'implémentation. Les résultats seront exploitables que si nous sommes en possession de toutes les solutions relatives à toutes les possibilités de configuration. Autrement, nous ne pouvons statuer sur une solution en disant qu'elle est la seule plus adaptée à nos besoins.

Dans notre cas, nous avons pu visiter tout l'espace d'exploration après une exécution sans interruption du flot pendant 7 jours. Si nous avions introduit en plus les paramètres de profondeur liés aux FIFO de lecture, une exploration complète de tout l'espace aurait été impossible. Effectivement, le nombre de configurations possibles dans le cas de notre système sur puce programmable, qui est relativement simple, serait de 524288 cas. Sachant que la synthèse de chaque configuration dure en moyenne 20 minutes le temps total qui faudrait pour avoir l'ensemble de toutes les solutions possibles est de 19,9 années.

Cette méthode exhaustive nous permet d'aborder l'exploration de manière empirique en ne se basant sur aucune théorie. Le but est d'observer les résultats de l'expérience afin de tirer les leçons des difficultés qui lui sont associées et proposer

une solution adaptée. Aussi cette première expérience nous a permis d'asseoir un flot de conception et le rendre plus robuste en utilisant des outils du marché, en l'occurrence ceux que propose Xilinx.

3.5.5 Exploration génétique à base de l'algorithme NSGA-II

Nous avons réalisé notre exploration multi-objectif à l'aide de l'algorithme NSGA-II déjà introduit dans le chapitre 2 de cette thèse. L'exploration génétique, tout comme pour l'exploration exhaustive, s'est faite sur la mémoire cache de données et les différentes tailles de FIFO de communication entre le PowerPC et ses IP. Nous rappelons que le but est de comparer les résultats obtenus avec les deux méthodes (Exhaustive, génétique multi-objectif) en terme d'exploitation. L'algorithme que nous avons utilisé a dû être adapté à nos besoins d'implémentation sur FPGA. Le nombre de cycles que prend l'exécution de chaque configuration est lu de la carte FPGA puis injecté directement dans le flot d'exploration génétique multi-objectif.

3.5.5.1 Représentation chromosomique

Notre espace d'exploration est constitué de 4 variables. Ces variables sont : l'utilisation du cache et la profondeur des FIFOs de communication. Ces différents paramètres ont une représentation binaire.

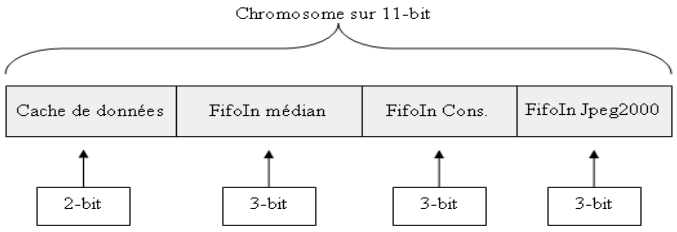


FIG. 3.18 – Chromosome exploration multi objectif

La variable relative au paramètre cache est codée sur 2 bits du fait que cette variable peut prendre que deux états : soit en active l'utilisation du cache de données soit on la désactive. Concernant les variables liées à la profondeur des FIFO de communication, celles-ci sont codées sur 3 bits (8 valeurs pour chaque FIFO). Le chromosome décrivant ceci est représenté par la figure 3.18.

Le tableau 3.7 récapitule les options appliquées à l'algorithme génétique NSGA-II lors de l'exploration du système monoprocesseur sur puce programmable. Les fonctions objectif celles fixées pour l'exploration exhaustive : performance en nombre de cycles, nombre de BRAM et le nombre de slices utilisés.

Le flot de l'exploration génétique multi-objectif est représenté par la figure 3.19. On commence par définir l'application monoprocesseur à implémenter en décrivant

Paramètres	Valeurs
Taille de la population	24
Nbre de générations	5
Nbre de fonctions objectif	3
Nbre de variables dans l'espace d'exploration	4
probabilité de croisement	0,8
probabilité de mutation	0,015

TAB. 3.7 – Paramètres appliqués à NSGA-II

les modules nécessaires. Cette étape est exactement la même que celle pour l'exploration exhaustive. Par la suite, on passe à la définition des options et paramètres génétiques que nous appliquerons à l'algorithme NSGA-II ((3.7).

Lors du premier lancement du flot, l'algorithme NSGA-II génère une première population nécessaire pour son initialisation. A cette étape, l'algorithme fixe de manière aléatoire des valeurs aux différents modules paramétrables (faisant partie de l'espace d'exploration). La deuxième itération concernera la première génération. À partir de cette génération les valeurs des modules paramétrables sont fixées selon les résultats intermédiaires liés aux trois fonctions objectif. Les résultats d'occupation en ressources (BRAM et Slices utilisés) sont directement collectés des fichiers de placement routage. Concernant le résultat de performance, ce dernier est obtenu par exécution sur FPGA.

La figure 3.20 représente le front de Pareto rang 1 relatif à l'exploration NSGA-II multi-objectif. On déduit de la forme de cette courbe de Pareto que l'algorithme d'exploration a bien convergé. Celle-ci a exigé 144 implémentations sur FPGA (24 individus, 5 générations+une génération d'initialisation). On remarquera que cette exploration a visité les régions critiques de l'espace d'exploration nous permettant ainsi une interprétation plus efficace des résultats. Nous remarquerons aussi que l'utilisation des mémoires, liées dans notre cas au cache de données et à la profondeur de FIFO de communication, avait un impact direct sur la performance.

Le concepteur peut à présent, avec l'aide de cette courbe de Pareto représentant un ensemble de solutions optimales, décider de la configuration la plus adéquate afin de répondre à ses exigences d'implémentation selon ses besoins en performance et en mémoire.

3.5.6 Comparaison entre exhaustive et génétique

Afin de mieux comparer les résultats obtenus par exploration exhaustive et ceux obtenus par exploration génétique multi-objectif, nous avons rassemblé ces deux résultats dans un même graphe représenté par la figure 3.21. La région 1 repré-

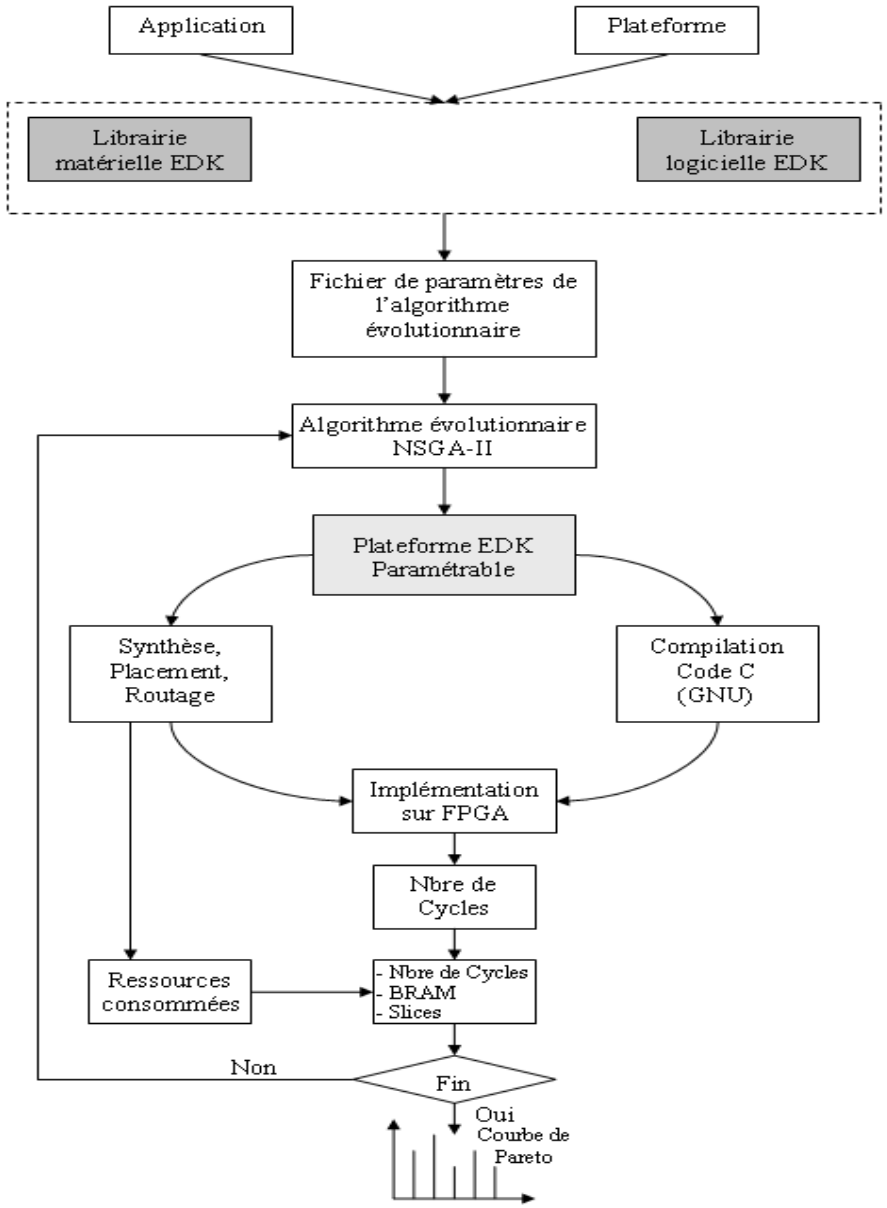


FIG. 3.19 – Flot exploration multi objectif

sente les résultats de l'exploration exhaustive sur 512 configurations après lui avoir appliqué une recherche de solutions optimales par critères de Pareto. La région 2 représente les résultats d'exploration relatifs au flot génétique multi-objectif NSGA-II. Ces deux régions ont été tracées sur une échelle différente, mais sur un même plan, afin de pouvoir les dissocier visuellement. On remarquera que dans le cas de la deuxième méthode (NSGA-II), l'exploration a pu, avec peu d'individus et de générations, visiter des configurations critiques exprimant beaucoup d'informations sur

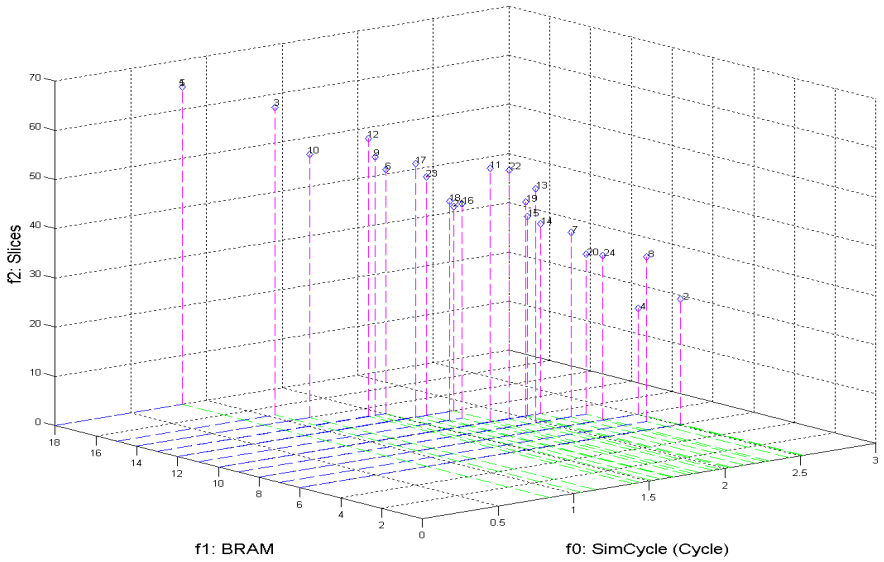


FIG. 3.20 – Exploration génétique nbre population=24, nombre génération=5

l'implémentation de notre application traitement d'images sur cette plateforme système sur puce. L'étendue sur l'espace d'exploration de cette méthode évolutionnaire avec ce nombre d'individus et de générations est assez satisfaisant sachant que ces paramètres peuvent facilement être améliorés pour une meilleure convergence.

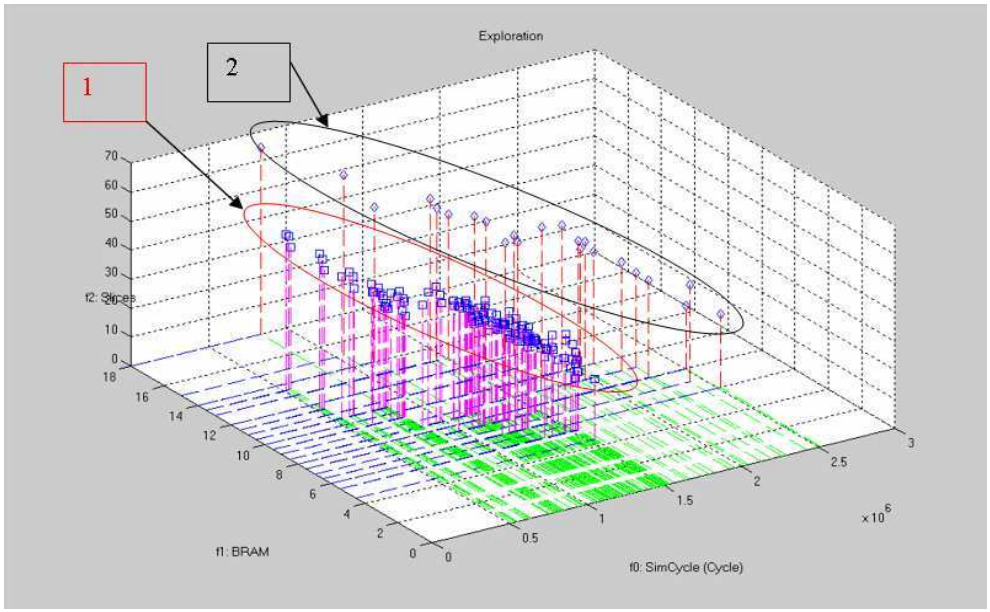


FIG. 3.21 – Comparaison résultats exhaustive Vs NSGA-II

En comparant encore plus les deux graphes, nous remarquons l'importance de la notion d'élitisme et de diversité dans l'exploration. Les résultats de l'exploration NSGA-II sont plus représentatifs du fait de leur diversité. Contrairement à l'ensemble de Pareto extrait à partir de l'exploration exhaustive qui élimine des résultats importants sur les extrémités de l'espace d'exploration.

Le nombre de paramètres sur lesquels on peut jouer est assez important rendant ainsi l'utilisation d'une méthode d'exploration exhaustive inefficace. Ceci est très bien illustré dans la région 1 de la figure 3.21. Les résultats dans ce cas sont très proches les uns des autres du fait que nous n'avions aucune information pour définir un espace d'exploration.

La comparaison illustrée par la figure 3.21 confirme définitivement l'efficacité d'une exploration génétique multi-objectif dans le cas d'une plateforme système embarqué. Il est à noter que nous avons choisit dans ce flot de conception de considérer la surface utilisée par le circuit comme résultat des rapports de synthèse et placement routage. Ceci nous différencie des approches basées sur l'estimation des ressources à la compilation comme dans [63] car l'impact du placement routage ne peut être ignoré dans les circuits de grande taille FPGA et pour lesquels les ressources (BRAM, multiplieurs, DSP...etc) se trouvent à des emplacements topologiques fixes et en nombre limité.

3.6 Implication pour les MPSoC

Nous essayons, par le graphe représenté par la figure 3.22, de montrer l'accroissement du nombre de configurations selon l'architecture que nous utilisons. Ces architectures multiprocesseurs utilisent des FIFOs comme médium de communication. Sur les deux architectures multiprocesseurs, nous avons bien sûr gardé à chaque fois les mêmes paramètres que pour l'architecture monoprocesseur que nous avons étudiée dans ce chapitre.

Nous précisons que ces architectures multiprocesseurs ont été d'abord définies afin de vérifier que leur taille leur permet d'être implémentées sur FPGA. Le nombre de configurations varie d'une architecture à une autre de manière exponentielle et drastique. Cela renforce l'idée d'utiliser les algorithmes évolutionnaires multi-objectif pour l'exploration de plateformes multiprocesseurs sur puce.

3.7 Conclusion

Nous avons étudié dans ce chapitre la possibilité d'effectuer une exploration basée sur l'algorithme évolutionnaire multi-objectif NSGA-II d'une architecture monoprocesseur sur puce. Une définition efficace (Compromis performance Vs ressources) des interfaces de communication (Le bon dimensionnement des FIFO de communication) dans un système sur puce, entre le processeur et ses IP accélératrices constitue un

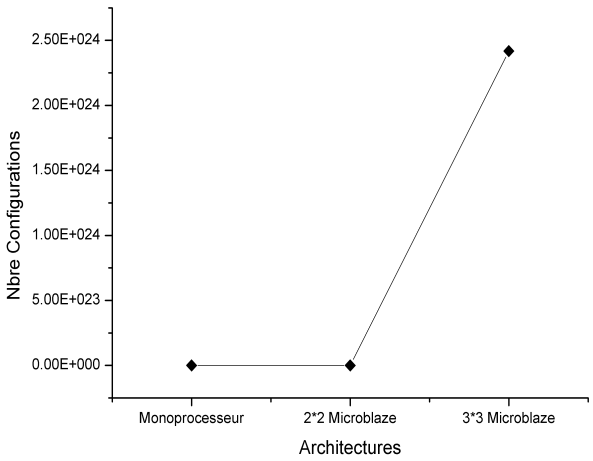


FIG. 3.22 – Nombre de configurations selon l'architecture

point essentiel pour l'optimisation des performances. Afin de mettre en évidence les améliorations apportées par la méthodologie introduite dans ce chapitre, nous l'avons comparée avec une exploration classique exhaustive. L'étude a bien montré que le nombre important de paramètres sur une architecture embarquée rendait difficile l'exploration exhaustive. Les résultats obtenus par exploration génétique multi-objectif ont bien prouvé l'efficacité d'une telle méthode d'exploration. Les résultats obtenus dans ce cas représentent un ensemble de solutions optimales classées sur le rang 1 du front de Pareto. Grâce à ces résultats, on facilite considérablement la tâche du concepteur dans sa prise de décision. Nous avons aussi introduit dans le flot une exécution directe de l'architecture à évaluer sur une plateforme reconfigurable, afin de collecter les résultats de performance en nombre de cycles. Cela permet essentiellement d'éviter le temps prohibitif de simulation et bien sûr d'avoir des résultats d'exploration correspondant à une exécution réelle et non pas à une estimation par simulation. Le but de ce chapitre est de conforter l'idée de passer par des explorations basées sur des algorithmes évolutionnaires multi-objectif pour l'optimisation de l'utilisation de la mémoire dédiée à la communication au sein des systèmes multiprocesseurs sur puces programmables. Nous pouvons ainsi aborder, dans les chapitres suivants, la conception de systèmes multiprocesseurs sur puces programmables sur des bases solides et fondées.

Chapitre 4

Multiprocesseur et exploration MPSOPC

Nous abordons dans ce chapitre les architectures multiprocesseurs sur puce en exposant un bref état de l'art sur leur apparition et sur les différentes architectures de communication utilisées. Différents types de réseaux sur puce (NoC : Network on Chip) seront présentés et comparés. Nous avons aussi dans ce chapitre réalisé une architecture multiprocesseur sur puce programmable implémentant une couche de communication logicielle qui a fait office de cas d'étude pour notre proposition de méthodologie. Le point central de ce chapitre a été de proposer une méthodologie de conception pour systèmes multiprocesseurs sur puce répondant à différents problèmes auxquels se heurtent les concepteurs.

Une architecture multiprocesseur sur puce de tendance actuelle a été réalisée et utilisée comme cas d'étude pour le point central de ce chapitre en l'occurrence la méthodologie d'exploration multiobjectif pour multiprocesseurs sur puces programmables.

4.1 État de l'art des architectures multiprocesseur

La figure 4.1 illustre très bien l'augmentation considérable de la performance brute du calcul enfoui (2GOPS avec des technologies 90 nm à 77 GOPS avec des technologies 45 nm) tout en gardant une puissance crête quasi constante (100 mW). La figure 4.2 présente une partie des différentes classes d'applications existantes ainsi que la capacité en performance que celles-ci exigent. Nous constatons que l'utilisation à tous les niveaux possibles du parallélisme pour augmenter la puissance de traitement à consommation énergétique donnée est devenue nécessaire afin de garantir l'implémentation de ces applications. En effet, le problème de la puissance dissipée (ordinateurs de bureau) et de l'énergie consommée avec son impact sur la durée de vie des batteries (ordinateurs portables, systèmes enfouis et embarqués) est maintenant commun à toutes les classes de systèmes informatiques. Au niveau

Process Technology (nm)	130	90	65	45	32	22
Operation voltage (V)	1.2	1	0.8	0.6	0.5	0.4
Clock frequency (MHz)	150	300	450	600	900	1200
Application (MAX performance required)	Still Image Processing	Real Time Video Codec MPEG4/CIF		Real Time Interpretation		
Application (Others)	Web Browser Electric mailer Scheduler	TV phone (1:1) Voice recognition (input)		TV phone (>3:1) Voice recognition (operation)		
		Authentification (Crypto engine)				
Processing Performance (GOPS)	0.3	2	14	77	461	2458
Parallelism factor	1	4	4	4	4	4
Communication speed (Kbps)	64	384	2304	13824	82944	497664
Energy Efficiency (MOPS/mW)	3	20	140	770	4160	24580
Peak Power Concumption (mW)	100	100	100	100	100	100
Stand-by Power Concumption (mW)	2	2	2	2	2	2
Battery Capacity (Wh/kg)	120	200		400		

FIG. 4.1 – La "roadmap" du calcul enfoui (Source ITRS Design ITWG July 2003)

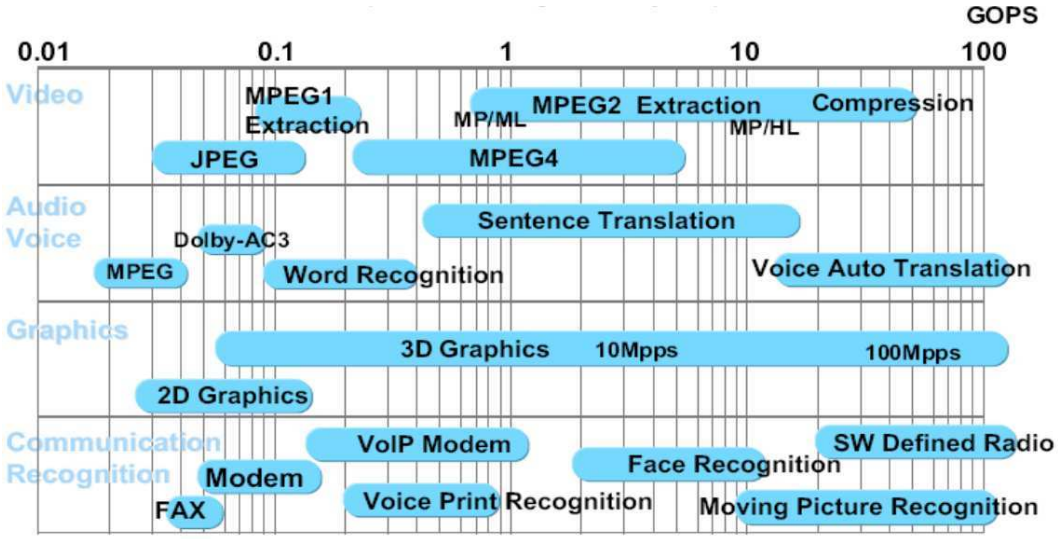


FIG. 4.2 – La "roadmap" des application enfouies (Source ITRS Design ITWG July 2003)

des processeurs d'usage général, c'est l'augmentation exponentielle de la densité de puissance (figure 4.3) qui a conduit à renoncer à l'augmentation continue des fréquences d'horloge au profit du parallélisme (double coeur et multi coeur). Dans le cas des processeurs pour applications enfouies, c'est la meilleure efficacité énergétique qui est la raison de l'utilisation croissante du parallélisme. Nous citons ci-dessous une comparaison proche de celle effectuées par P. Paulin de ST [64]. Exemple de processeurs " logiciels " pour l'enfoui et l'embarqué :

- Processeur " bas de gamme "
  MIPS M4K : 367 Drystone MIPS à 240 MHz. Puissance : 0,05 mW/MHz
- Processeur " haut de gamme "
  MIPS 24K : 900 MIPS à 625 MHz : 0,58 mW/MHz.

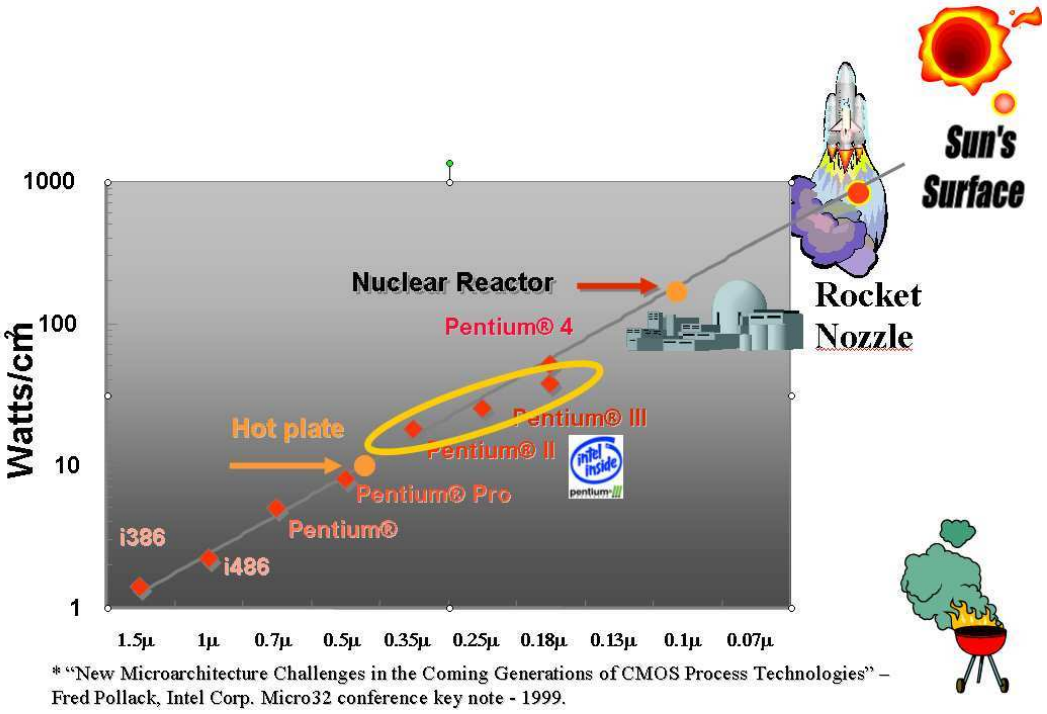


FIG. 4.3 – Densité de puissance

3 processeurs 24K délivrent 1100 MIPS (contre 900) en consommant 36 mW au lieu de 362 mW. Gain énergétique supérieur à 10 !
C'est la raison essentielle du succès des multiprocesseurs sur puce.

D'après le " survey " du journal " Embedded Systems Programming magazine " cité dans [64], près de la moitié des circuits considérés utilisaient au moins deux processeurs.

De plus, les progrès qui se font sur les architectures monoprocesseurs afin d'améliorer le parallélisme au niveau instructions, ne peuvent pas être maintenus indéfiniment vu la complexité croissante engendrée au niveau circuit lors de leur réalisation. Reste que, le fait de se diriger vers des architectures multiprocesseur nous expose à une multitude de problèmes, relatifs aux processeurs utilisés et surtout aux moyens de les faire interagir. Les multiprocesseurs sont utilisés depuis longtemps dans les serveurs et les machines parallèles haut de gamme et de nombreux progrès on été réalisés pour faciliter leur programmation et leur utilisation. Les systèmes embarqués ajoutent des contraintes particulières en temps d'exécution, en consommation d'énergie et en coût de réalisation d'architecture monopuce.

Les processeurs parallèles représentent des systèmes se composant de plusieurs unités de traitement reliées par l'intermédiaire d'un certain réseau d'interconnexion en plus des logiciels requis pour faire fonctionner ensemble ces unités de calcul. Il y a deux facteurs principaux employés pour classer de tels systèmes : les unités

de traitement, et le réseau d'interconnexion qu'elles utilisent pour leur communication. Les unités de traitement peuvent communiquer et interagir les unes avec les autres en utilisant une mémoire partagée ou bien la méthode de passage de messages. Le réseau d'interconnexion pour les systèmes à mémoire partagée peut être classifié comme étant des communications basées sur des bus ou bien sur des commutateurs. Dans les systèmes à passage de messages (Message Passing systems), le réseau d'interconnexion est divisé en réseau statique et dynamique. Les réseaux statiques ont une topologie fixe qui ne change pas pendant l'exécution des programmes sur la plateforme. Les réseaux dynamiques, quant à eux, créent des liens de manière dynamique pendant que le programme s'exécute. Ce qui encourage l'utilisation de multiprocesseurs est la possibilité d'avoir un système puissant en connectant simplement plusieurs processeurs. De plus, un système multiprocesseur permet d'atteindre des performances meilleures que celles d'un monoprocesseur tout en restant à des fréquences d'exécution raisonnables. Ce facteur est très important dans les systèmes embarqués vu que ces derniers exigent une basse consommation et un bon rendu en performance.

La taxonomie la plus populaire de l'architecture d'ordinateur englobant l'architecture multiprocesseur a été définie par Flynn en 1966. Cette classification est basée sur la notion de flot d'information. Deux types de flot d'information existent dans un processeur : le flot d'instructions et le flot de données. Le flot d'instructions est défini comme étant l'ensemble des instructions à exécuter dans un ordre précis par l'unité de traitement. Le flux de données est défini comme étant le trafic de données échangé entre la mémoire et l'unité de traitement. D'après la classification proposée par Flynn, le flot l'instructions ou de données peuvent être mono ou multiple. Cette classification est donnée par le tableau 4.1.

	Unique	Multiple
Unique	SISD (Von Neumann)	SIMD (Tab de processeurs)
Multiple	MISD (Pipeline)	MIMD (Multiprocesseur)

TAB. 4.1 – Classifiaction des machines

Les systèmes SISD sont les architectures monoprocesseurs standards. Les architectures parallèles sont les architectures SIMD et MIMD. L'architecture SIMD représente un système où il existe une seule unité de commandes où tous les processeurs exécutent la même instruction. L'architecture MIMD représente, quant à elle, un ensemble de processeurs où chaque unité de calcul a sa propre unité de commandes et peut exécuter différentes instructions sur des données différentes. Cette architecture représente la réelle architecture multiprocesseur où chaque processeur est indépendant par rapport à ses voisins.

4.1.1 Multiprocesseur sur puce

Connaissant les avantages des systèmes parallèles et les avancées technologiques réalisées dans le domaine du silicium, les systèmes sur puce (SoC) se sont rapidement transformés en systèmes multiprocesseurs sur puce (MPSoC : multiprocessor system on chip). Ces systèmes combinent les avantages du parallélisme à celle de la large capacité d'intégration des systèmes sur puce. La figure 4.4 illustre l'architecture standard d'un système multiprocesseur sur puce.

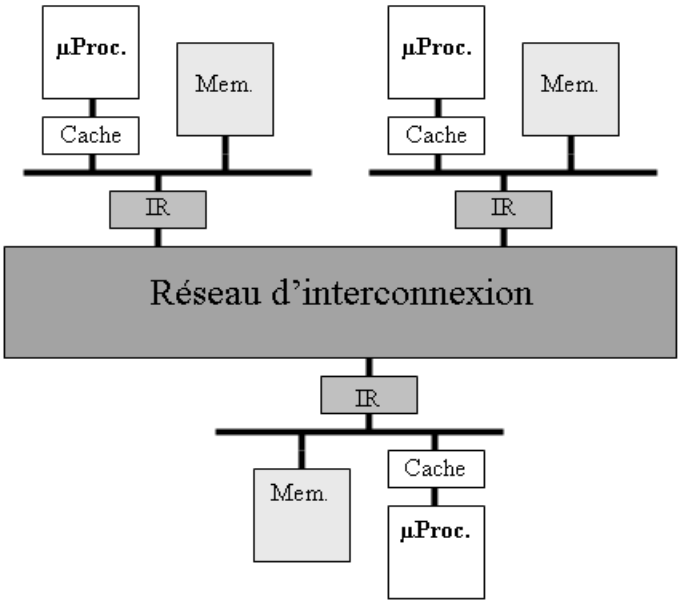


FIG. 4.4 – Architecture MPSoC standard

Cette architecture est composée de plusieurs grappes de processeurs où chaque processeur a sa propre mémoire, son propre bus et peut avoir aussi ses propres IP accélératrices.

Qu'est ce qui rend les MPSoC nécessaires ? Cette question permet la réflexion et justifie les efforts consacrés du passage des systèmes monoprocesseurs aux systèmes multiprocesseurs [65]. La quantité d'effort et la nature de l'effort pour réaliser un système monoprocesseur est loin d'atteindre la complexité de conception des systèmes multiprocesseurs. Cependant, les systèmes monoprocesseurs présentent des lacunes évidentes faces aux applications actuelles. La première exigence des applications est de pouvoir s'exécuter en plusieurs processus en même temps. La deuxième exigence est de pouvoir exécuter plusieurs processus concurrents avec des contraintes de temps réel [66]. Les systèmes monoprocesseurs ont clairement des limites faces à de pareilles contraintes, et augmenter la fréquence pour combler ces lacunes n'est guère compatible avec la condition sine qua non de basse consommation. Les systèmes multiprocesseurs disposent, quant à eux, de cette capacité de manager des processus concurrents avec des contraintes temps réel.

Cependant, le passage aux MPSoC engendre la considération d'un certain nombre de problèmes. Ces problèmes sont liés à des facteurs divers et variés. Commençant par le plus évident, en l'occurrence la gestion d'une programmation parallèle. Depuis toujours, les programmeurs ont été habitués à poser leurs problèmes de manière séquentielle et ce, au delà du fait que les systèmes monoprocesseurs étaient les premiers à exister et qu'ils soient plus répandus que les systèmes multiprocesseurs [67]. Effectivement, il est beaucoup plus naturel de décrire un algorithme de manière séquentielle en fixant les besoins à réaliser à chaque étape. L'exécution d'une application sur une architecture multiprocesseur implique le fait que plusieurs tâches s'exécutent de manière concurrente tout en communiquant entre elles et partageant, pour le plus souvent, des ressources. Dans un tel schéma il peut y avoir à tout moment une perte de la gestion des priorités, des blocages (deadlock, starvation) où bien une incohérence dans les données. On est passé de la simple gestion d'exécution de tâches concurrentes sur un monoprocesseur aidée par des systèmes d'exploitation temps réel à un schéma parallèle où plusieurs tâches s'exécutent et communiquent en même temps de manière imprévisible [68]. Plus de quarante année de recherche dans ce domaine n'ont pas permis d'avoir une méthodologie ou un standard unique facilitant l'exploitation des systèmes multiprocesseurs. Les concepteurs se sont plus basés sur des aspects ad-hoc que sur une méthodologie de modèle de programmation et de synchronisation standard. Arriver à améliorer les aspects cités plus haut (modèle de programmation, synchronisation) ne veut pas dire que le travail de mise au point d'une architecture MPSoC est achevé. Effectivement, le concepteur aura besoin de débugger son application et savoir clairement et précisément comment celle-ci s'exécute. Cette tâche, dans le domaine multiprocesseur, est considérablement complexe pour le concepteur qui doit comprendre l'exécution de plusieurs processus en même temps mais aussi à la mise au point d'une plateforme permettant de le faire. Les MPSoC offrent la possibilité de mettre au sein d'un même paquetage plusieurs processeurs et même différents types de processeurs afin de répondre précisément aux contraintes et demandes en performance de l'application à implémenter. De plus, chaque processeur fournit des possibilités de configuration aussi bien que pour les réseaux de communication. Cette diversité des systèmes MPSoC engendre un problème lié à la taille de l'ensemble de solutions. Ceci a pour conséquences une explosion de l'espace de configurations du fait du nombre de paramètres : quel schéma de communication choisir ? quel mapping ? et quelle configuration appliquer à chaque processeur et à chaque élément de communication ? Le nombre de solutions, la complexité des MPSoC et le coût de la conception font en sorte que les anciennes approches ad-hoc sur lesquelles se basaient les concepteurs ne sont plus viables. Ce que nous proposons dans cette partie de la thèse est une méthodologie de conception pour MPSoC offrant la possibilité de répondre aux problèmes rencontrés dans la conception MPSoC cités dans les paragraphes précédents. Le temps de cycle de conception et la qualité des solutions représentent les deux valeurs que nous tenterons d'optimiser.

4.1.1.1 Multiprocesseurs homogènes ou hétérogènes

La première question est celle du type de multiprocesseur utilisé sur puce. Les multiprocesseurs homogènes (ou symétriques) utilisent des processeurs identiques, et sont la " transcription " au niveau des systèmes sur puce des multiprocesseurs symétriques (SMP) largement utilisés pour les serveurs et sous forme de multi-coeurs dans les ordinateurs de bureau et ordinateurs portables. Les multiprocesseurs hétérogènes (ou asymétriques) utilisent différents types de processeurs. Cette approche existe depuis plusieurs années sous forme de circuits du commerce, comme par exemple OMAP (Figure 4.5) chez Texas Instruments, Nomadic chez ST, etc. La figure 4.5 présente le schéma fonction de la version 2420 d'OMAP, dans lequel on reconnaît un processeur RISC (ARM 11) servant de contrôleur, un processeur de traitement du signal (TMS320C55x), et deux accélérateurs matériels spécifiques : un accélérateur pour le graphique et un autre pour la vidéo. S'ajoutent tous les composants nécessaires pour gérer les transferts mémoire (DMA), les entrées sorties et même des opérateurs de cryptage. Ce qu'il est intéressant de noter ici, c'est l'utilisation de processeurs différents, chargés plus spécifiquement de répondre aux besoins différents des différentes parties ou des différentes phases d'une application. D'après [Paulin] citant le " survey " de 2005 déjà mentionné, près de 2/3 des multiprocesseurs sur puce sont hétérogènes. Les générations successives de plateformes de ST utilisent cette approche " multiprocesseurs hétérogènes ". Dans cette thèse, nous utilisons des multiprocesseurs homogènes.

Nous rappelons que notre objectif principal est de montrer l'apport de l'approche "algorithmes génétiques" pour l'exploration de l'espace de conception en fonction des paramètres architecturaux. L'utilisation de plusieurs processeurs identiques sur FPGA (en l'occurrence les PowerPC405) nous permet déjà de fournir des résultats significatifs. De plus, l'utilisation d'un multiprocesseur symétrique n'implique pas obligatoirement un mode de fonctionnement SPMD (où tous les processeurs exécutent le même programme sur des données différentes). Si les différents processeurs exécutent des programmes différents (par exemple, dans un parallélisme de type "pipeline" où chaque processeur exécute une tâche correspondant à une étape du pipeline), alors on peut "simuler" une architecture hétérogène : dans ce cas, l'hétérogénéité est introduite par le programme et conduit à des temps d'exécution différents, des débits de transfert de données différents dans l'ensemble des processeurs utilisés. Par ailleurs, y compris en utilisant un support FPGA, l'existence d'IP spécialisée n'interdit pas de poursuivre dans le futur, dans un contexte multiprocesseur hétérogène, les travaux que nous avons menés dans un contexte multiprocesseur homogène.

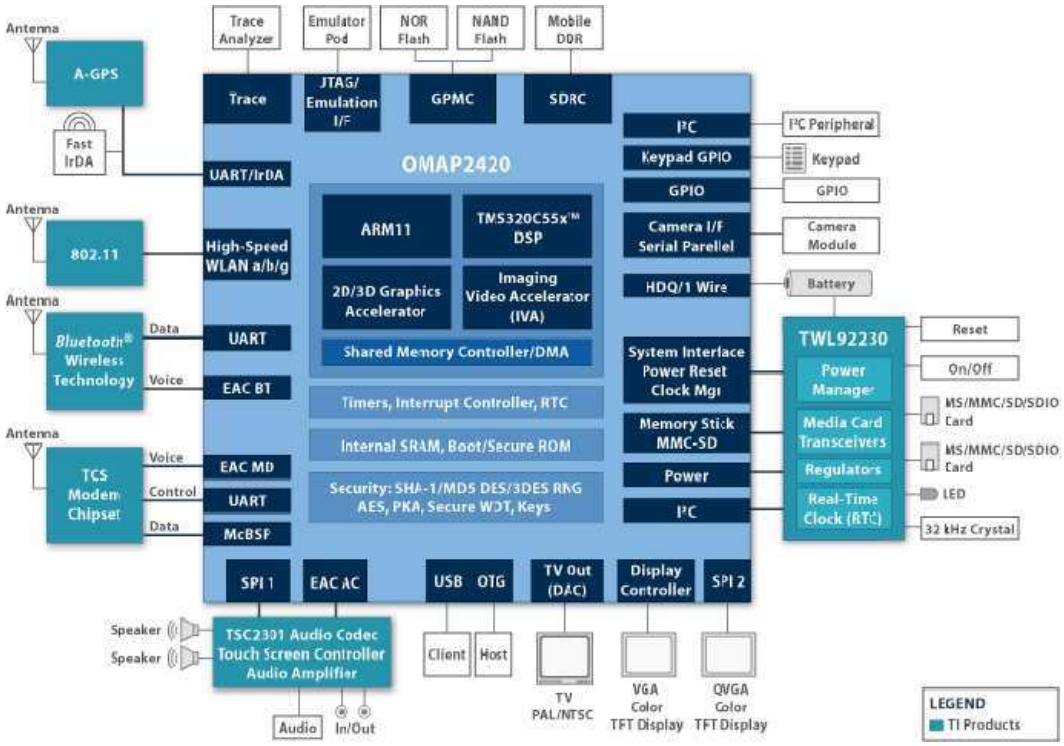


FIG. 4.5 – Exemple de multiprocesseur hétérogène : OMAP de Texas Instruments

4.2 Méthodologies de conception MPSoC

Du fait de la complexité des applications parallèles et de l'architecture multiprocesseur sur puce, plusieurs méthodologies de conception pour MPSoC ont été proposées ces dernières années. Ces propositions ont pour but de définir une règle commune à l'implémentation de toutes sortes d'applications parallèles en cherchant à faciliter la tâche de conception tout en ayant un maximum de performances. La taille des MPSoC ne permet pas une conception au niveau RTL. C'est pour cela que la plupart des méthodologies présentées actuellement tentent de monter en abstraction afin d'acquérir un gain en facilité et en temps. Il est à signaler que la monté en abstraction est nécessaire aussi bien du côté matériel que logiciel. Effectivement, la partie logicielle d'un système multiprocesseur est énormément dense en terme de lignes de code. Il est par conséquent, impossible d'utiliser dans ce cas un outil de description logicielle bas niveau. On peut citer comme exemple l'outil ROSES proposé par le laboratoire TIMA [69]. Il permet la création d'une plateforme multiprocesseur sur puce à partir d'une bibliothèque de composants matériels/logiciels configurables pouvant être adaptés à différents standards. Le modèle de simulation est aussi tiré de cette bibliothèque. ROSES permet de connecter automatiquement à partir de la spécification de l'application, les tâches du processeur aux accélérateurs matériels et les mémoires. L'outil offre aussi la possibilité de valider l'architecture

à différents niveaux d'abstraction grâce à un mécanisme de co-simulation. L'entrée du flot ROSES est une spécification de l'application décrite avec un langage nommé VADeL (Virtual Architecture Description Language) défini par le groupe SLS [70]. Ce langage n'est autre qu'une extension de SystemC. Cette méthodologie permet une exploration rapide pour le partitionnement logiciel/matériel du système, du fait qu'elle soit basée sur une spécification à haut niveau. Le concepteur peut dans ce cas évaluer assez rapidement l'implémentation du système et avoir une idée au niveau transactionnel de la communication.

Un excellent travail a été proposé et réalisé au sein du laboratoire TIMA traitant cette problématique de méthodologie de conception pour MPSoC [71]. L'auteur, dans le cadre de sa thèse, a proposé une approche consistant à implémenter l'algorithme d'encodage MPEG4 [1]sur une plateforme MPSoC. Pour une implémentation efficace de cet algorithme, il a été question de donner la possibilité à l'encodeur MPEG4 d'avoir différentes configurations d'algorithmes et d'architectures. Dans ce travail, une architecture multiprocesseur à base de deux processeurs connectés sur un modèle d'architecture SMP (Symetric multiprocessors) a été utilisée pour l'implémentation d'un encodeur MPEG4 flexible. La description de la plateforme s'est faite à un haut niveau d'abstraction avec SystemC. Une couche logicielle MPI (message passing interface) a été choisie pour la communication entre processeurs. L'auteur a proposé une exploration, en même temps, d'algorithmes et d'architectures, afin d'extraire les bons paramètres pour l'implémentation de l'encodeur MPEG4. Ce travail d'exploration s'est effectué à un haut niveau d'abstraction ce qui a permis une exploration rapide par rapport à une simulation niveau RTL. Cependant, des paramètres importants ont dû être ignorés afin de favoriser la rapidité d'exploration.

Nous retenons de ce travail, ainsi qu'avec d'autres travaux de proposition de méthodologies de conception MPSoC [72, 73], deux principales différences avec ce que nous présentons ici dans cette thèse. La première différence est liée au fait d'abstraire certains éléments importants à l'implémentation afin d'avoir un gain en vitesse de simulation. La deuxième différence réside dans le fait que nous revendiquons un aspect Pareto et multiobjectif aux systèmes multiprocesseurs sur puce. Pour une application parallèle donnée, il existe un ensemble de solutions optimales faisant un réel compromis entre les ressources sur puce et la performance.

4.2.1 La communication dans les MPSoC

La capacité d'intégration impliquant la possibilité d'implémenter sur puce un système multiprocesseur, nécessite l'utilisation d'un médium de communication atteignant les différents modules du système sans affecter l'intégrité des messages y transitant. Seulement, à cause de la taille du silicium (de la puce) les fils se heurteront forcément à un problème de délai [74]. Il doit certainement exister des méthodes

[1]MPEG-4 (ISO/CEI 14496), introduit en 1998, est une norme de codage d'objets audiovisuels spécifiée par le Moving Picture Experts Group

contournant le problème comme par exemple pipeliner les liens de communication, sauf que celles-ci, demanderont une connaissance précise et exacte du délai du signal en question. Ce problème représente le côté négatif de l'accroissement de la capacité d'intégration. Synchroniser les systèmes sur puce sur un même signal d'horloge sera une tâche particulièrement difficile à réaliser [75].

Le passage à une architecture de communication réseaux à commutation de paquets sur puce permet de résoudre les problèmes liés aux délais des fils, à la propagation du signal d'horloge, à l'intégrité des messages et de la bande passante[76, 77]. Par conséquent, la communication sur puce sera vu comme étant un micro réseau avec différentes couches de communication permettant une abstraction des propriétés électriques, logiques et fonctionnelles de la communication [78].

L'avantage de la communication par paquets réside dans l'importante modularité impliquée sur les réseaux. Comme le cycle de conception des systèmes sur puce implique d'importantes contraintes de temps, cet aspect devient un réel plus. Il existe une standardisation dans les réseaux sur puce mais celle-ci ne concerne que les interfaces à l'entrée du réseau (Interface entre le module de calcul et le réseau). Concernant le reste, et plus précisément la topologie de communication, le concepteur a la possibilité de l'adapter à son application. Cet aspect a fait émerger un point essentiel dans les systèmes sur puce concernant la synthèse de réseau sur puce en adaptant le schéma de communication aux exigences de l'application. Cet aspect sera abordé dans le chapitre 5 de cette thèse.

4.3 Réseaux d'interconnexion

Dans ce paragraphe, nous ne prétendons pas présenter un état de l'art des réseaux d'interconnexion sur puce. Notre présentation s'inspire de la présentation de J. Xu et W Wolf [79] and De Micheli [80] pour justifier le choix, de l'architecture de communication, effectué pour notre étude d'exploration d'espace de conception MPSoC sur FPGA. Nous rappelons que les systèmes multiprocesseurs sont essentiellement constitués d'unités de calcul, de composants de mémorisation et enfin de composants de communication. Cette dernière partie (composants de communication) étant la plus critique vu que la plupart des circuits logiques sont maintenant limités par leurs communication et interconnexions et non pas, comme on le penserait, par leur vitesse de calcul (logique) ou mémoire [81, 82]. Ces composants de communication vont assurer l'interconnexion entre les différents modules de traitement ainsi que les mémoires. A la base de leur fonctionnement il y a des modules de routage de transport de l'information ainsi que des composants permettant de gérer l'ensemble, en attribuant à chaque unité de calcul et mémoire les informations qui lui sont destinées et la concernant.

Ces composants de communication peuvent être simples, c'est à dire, qu'ils vont juste se charger de transmettre l'information, ou l'acquittement ainsi que le contrôle

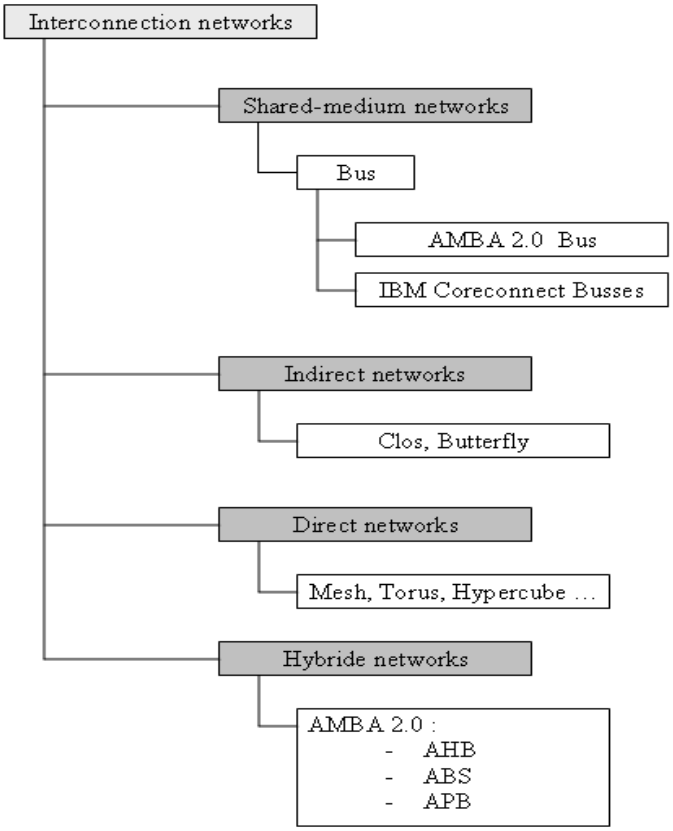


FIG. 4.6 – Classification des réseaux d'interconnexion sue puce

de la bonne transmission de données se feront sur les composants de calcul et sur les mémoires. Le deuxième type est plus compliqué, du fait qu'il se charge non seulement de la transmission de l'information, de l'acquittement et de l'espionnage du bus grâce à des modules supplémentaires, un arbitre permettra aussi de décider à qui donner la main et quelle donnée il doit faire passer à chaque unité de calcul. On peut classer ces différents types de réseaux d'interconnexion par rapport à leur mode de fonctionnement car ils peuvent être soient synchrones ou asynchrones, ou bien par rapport au modèle de contrôle du réseau puisque ce dernier peut être soit centralisé, décentralisé ou distribué. La figure 4.6 nous donne un résumé de tous les types de réseaux d'interconnexion existants [83]. Cette classification est constituée principalement de quatre catégories de réseaux d'interconnexion : les réseaux à ressources partagées, les réseaux directs, les réseaux indirects et enfin les réseaux hybrides (pour les réseaux hybrides, nous avons donné un exemple dans la figure 4.6 à base de spécification AMBA[2]) ou hétérogènes [85].

[2]La spécification AMBA [84] permet d'implémenter trois types de bus (protocoles) sur une même puce : AHB (High Performance Bus), ABS (Advanced System Bus), APB (Advanced Peripheral Bus)

4.3.1 Réseaux à ressources partagées (Bus)

Ce type de réseaux est le plus utilisé pour la communication au sein d'un système embarqué : cela est dû principalement à sa simplicité topologique et aussi à sa simplicité d'implémentation. Dans ce type de réseaux, le système de transmission est partagé entre les différents éléments du réseau où tout le trafic partage un seul chemin de transmission de données, par conséquent un seul élément peut seulement accéder à la fois au réseau. Cela crée un gros désavantage pour la communication car on est obligé d'augmenter la fréquence pour satisfaire une bonne transmission. Cela fait de ce type de bus un élément où la consommation d'énergie n'est pas gérée de manière efficace d'autant plus que ce réseau effectue une diffusion (broadcasting) systématique réquisitionnant tous les éléments du réseau d'interconnexion [86].

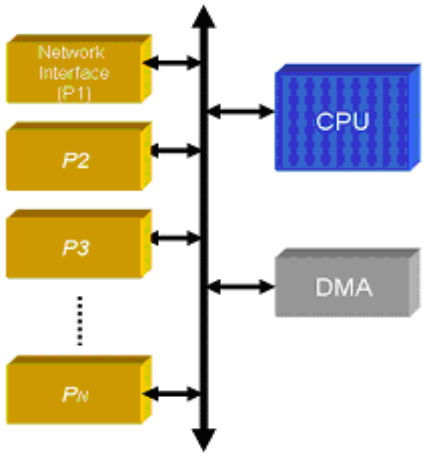


FIG. 4.7 – Bus

Ces réseaux d'interconnexion à ressources partagées sont largement utilisés mais comportent un réel problème d'extensibilité. Ce modèle de réseaux basés sur une architecture de bus, illustré par la figure 4.7, reste très bien utilisé dans les systèmes embarqués où l'on n'utilise que quelques processeurs [87]. Comme expliqué aussi dans le paragraphe précédent, chaque transmission de donnée est naturellement effectuée dans un mode diffusion (Broadcast) insinuant l'envoi de l'information à chaque récepteur dans le réseau même si ce dernier n'en n'a pas besoin. On comprendra facilement que pour les futurs systèmes embarqués avec des centaines de récepteurs, ces réseaux seront peu efficaces et consommeront énormément d'énergie ce qui est inapproprié pour des systèmes embarqués.

4.3.2 Réseaux directs

Les systèmes embarqués à multiprocesseurs basés sur des réseaux à bus ne peuvent s'agrandir car la bande passante diminue à chaque fois que l'on rajoute

un nouvel élément au réseau. Les réseaux directs (figure 4.8) ou bien réseaux point-à-point arrivent à compenser ce problème majeur des réseaux à ressources partagées qui limitent considérablement l'élargissement des applications. Ce réseau est constitué d'un ensemble de noeuds qui peuvent représenter plusieurs fonctionnalités ; ils peuvent être composés de processeurs graphiques, de processeurs d'entrées sorties ou d'un autre type de processeur. La façon dont les noeuds sont connectés entre eux dans un réseau change d'une machine à une autre. Dans l'architecture des réseaux directs, chaque noeud du réseau a une connexion point-à-point ou bien directe avec un ensemble de noeuds appelés voisins. Les réseaux directs sont beaucoup utilisés dans de larges architectures multiprocesseurs impliquant un nombre très important de processeurs parallèles, car plus le nombre de noeuds dans le réseau direct augmente plus on augmente la bande passante [88].

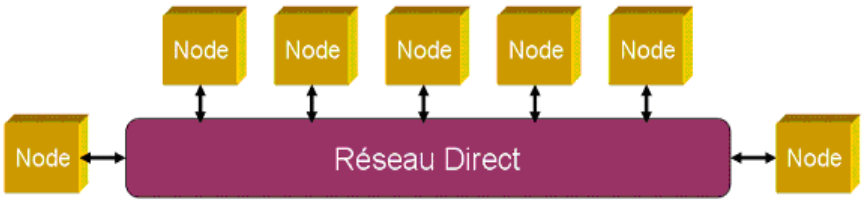


FIG. 4.8 – Multiprocesseur connecté à l'aide d'un réseau direct

On remarquera que dans les réseaux directs, les différents noeuds les constituant n'ont pas accès à une mémoire partagée, ce qui oblige la communication entre les noeuds à se faire en mode "Message Passing". Dans un réseau direct, un noeud est constitué principalement d'un processeur, d'une mémoire liée à son fonctionnement et d'autres fonctionnalités, comme le montre la figure 4.9. Ces différents éléments sont connectés sur un bus d'interconnexion local. Ce même bus connecte le dernier élément constituant le noeud, qui est le routeur, à la mémoire du processeur, cela à travers une paire de canaux internes (DMA : Direct memory access). Certaines implémentations utilisent plusieurs canaux internes minimisant ainsi le goulot d'étranglement pour la communication entre la mémoire et le routeur. Quant au routeur, il possède des canaux d'entrées et de sorties sur une interconnexion externe lui permettant ainsi d'accéder aux noeuds voisins. Comme nous l'avions cité auparavant, une information (message constitué d'un ensemble de paquets) doit passer par une multitude de noeuds (routeurs) afin de parvenir à sa destination. Les différents canaux externes traversés forment le chemin de l'information et le nombre de canaux externe forme la longueur du chemin (path length). Nous rappelons que l'avantage avec les réseaux directs est que la bande passante augmente avec l'adjonction d'autres noeuds dans le système, ce qui donne aux systèmes embarqués une extensibilité très élevée [89].

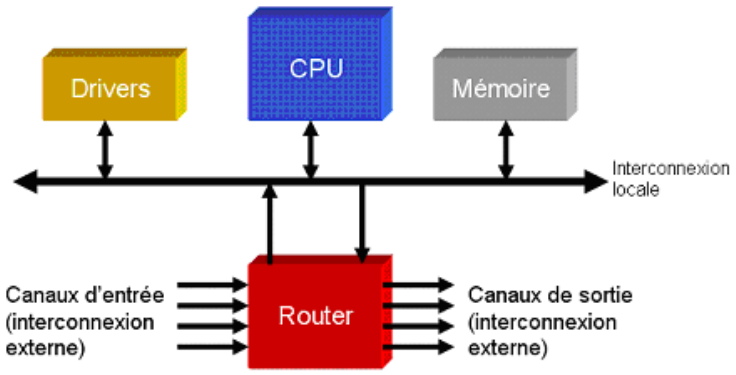


FIG. 4.9 – Architecture générique d'un noeud d'un réseau direct

4.3.3 Réseaux indirects

Dans ce type de connections, les liaisons entre deux noeuds ne se font pas de manière directe. Dans le cas des réseaux indirects, on passe par des commutateurs (switches). La différence qui existe alors entre les réseaux directs et indirects, est que la première catégorie de réseaux passe par des routeurs alors que la deuxième passe par des switches (interrupteur) [80]. Les commutateurs n'effectuent pas d'opérations de calcul, ils se chargent juste de fournir une connexion programmable entre les différents ports pour mettre en place une configuration de communication qui peut être changée selon le besoin durant la durée d'exécution. Parmi ces types de réseaux, on peut citer les réseaux " Crossbar " (figure 4.10), qui ont la possibilité de connecter n'importe quel élément à un autre élément dans le réseau sous réserve que les ports d'entrées et de sorties soient disponibles afin de mettre au point la connexion. Les réseaux " Crossbar " permettent d'avoir la possibilité de mettre en place plusieurs communications simultanément. Cet avantage rend les réseaux " Crossbar " très intéressants et très utilisés dans les systèmes multiprocesseurs.

Les entrées sorties du réseau sont interconnectées à l'aide de Cross-points. Ces Cross-points peuvent avoir, comme le montre la figure 4.11, deux états, un premier état Cross-state passant et un deuxième état Bend-state bloquant. On pourra donc à l'aide de ces Cross-points établir ou pas une connexion entre les entrées m et les sorties n.

Seulement, dans ces réseaux il existe malheureusement une limitation physique en ce qui concerne le nombre de Cross-points qu'on peut avoir au sein d'un réseau. Par conséquent, ce type d'interconnexion ne pourra pas satisfaire des systèmes multiprocesseurs importants en terme de nombre de processeurs. Cependant, des chercheurs ont mis au point un autre type de réseau en s'inspirant toujours des réseaux Crossbar afin de combler cette lacune. L'idée était d'utiliser plusieurs petits réseaux Crossbars permettant ainsi d'avoir un plus ample intervalle de connexion. Ce type d'interconnexion est dit " Multistage Interconnection Networks " (MIN),

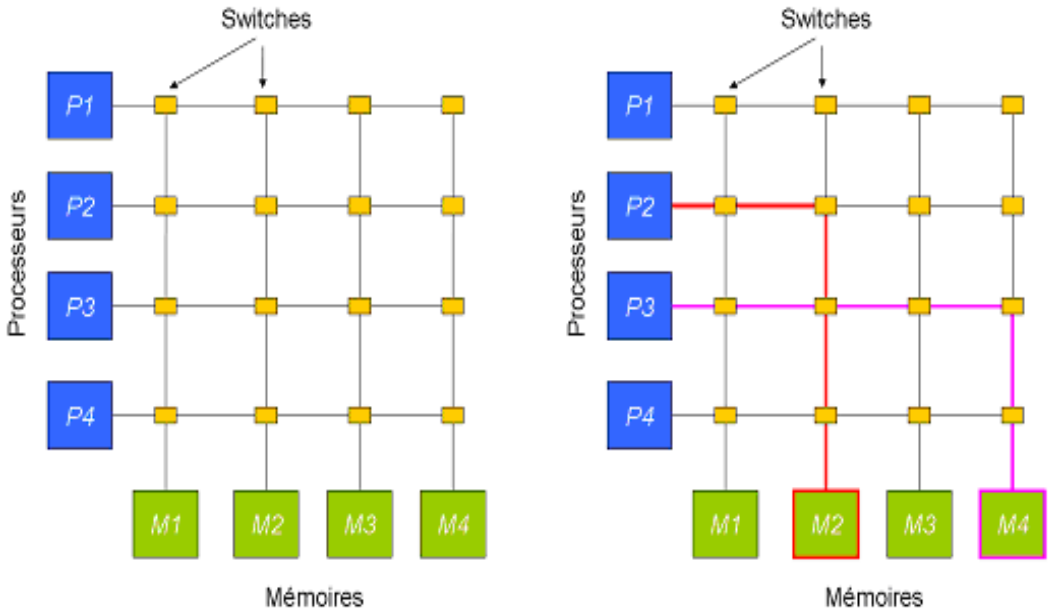


FIG. 4.10 – Réseau Crossbar

FIG. 4.11 – Cross-point

où l'on peut citer comme exemple un réseau 5 étages " 5-Stage Shuffle-Exchange Network " (figure 4.12) proposé en 1971 pour être une architecture efficace pour le calcul parallèle.

4.3.4 Réseaux hybrides

Une des solutions utilisée afin d'améliorer la bande passante des réseaux est d'avoir des réseaux hybrides. Ces réseaux sont de plus en plus utilisés vu qu'ils permettent d'atteindre de meilleurs résultats d'implémentation et d'avoir des réseaux d'interconnexion plus efficaces [90]. Les réseaux hybrides se divisent en trois classes principales à savoir, les " Multiple BackPlane Buses ", réseaux hiérarchiques et enfin les " Cluster-Based Networks ". Une des premières approches pour avoir une meilleure bande passante sur le réseau serait de mettre en place plusieurs bus : c'est le

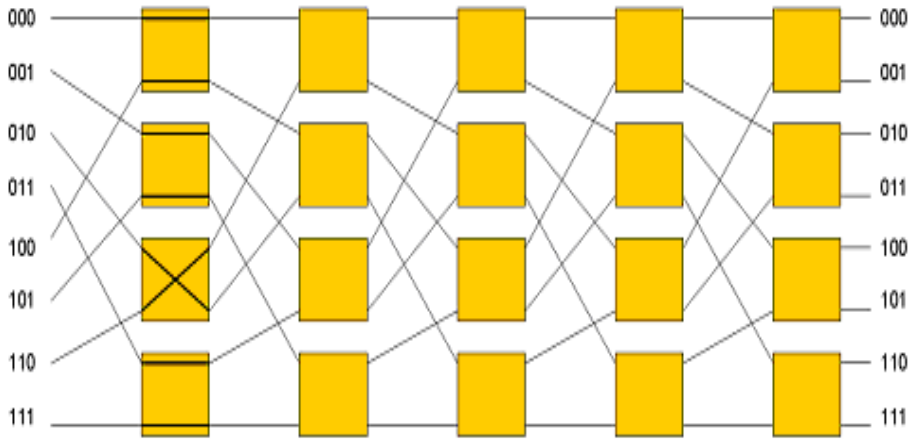


FIG. 4.12 – Shuffle-Exchange Network (5 étages)

cas pour les " multiple BackPlane Module ". Seulement, cette manière de faire n'est utilisée que dans les petits systèmes multiprocesseurs. Cela est dû principalement au câblage que nécessite ce type de réseaux. La deuxième approche permettant d'améliorer la bande passante est d'utiliser des bus hiérarchiques. Les différents niveaux de hiérarchie sont connectés entre eux à travers des routeurs ou des ponts, permettant ainsi le passage de l'information d'un côté du réseau à un autre. L'avantage avec ce type de réseaux réside dans le fait qu'on peut élargir notre interconnexion et prendre en compte un nombre plus grand d'éléments sans pour autant détériorer la bande passante.

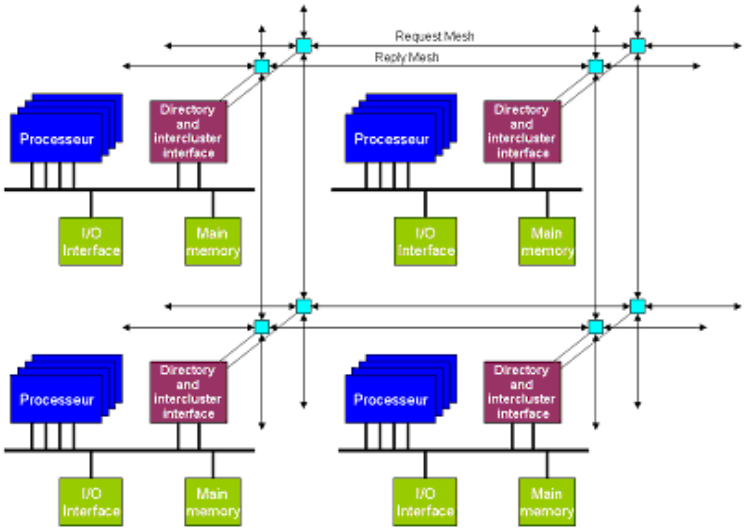


FIG. 4.13 – Schéma bloc du Système DASH (2 x 2)

Enfin un exemple de réseau ayant un aspect hiérarchique est le réseau " Cluster-

Based Network ". Dans ce modèle de réseaux, on retrouve plusieurs types d'interconnexions. On peut associer par exemple un réseau type mesh avec un réseau du type bus unique. Un exemple pour ce type de réseaux d'interconnexion est le " Directory Architecture for Shared-Memory Multiprocessor " (DASH) réalisé à l'Université de Stanford (figure 4.13). Cette architecture, malgré qu'elle soit ancienne est souvent citée dans plusieurs travaux de recherche. Dans cet exemple on distinguera un niveau bas où l'on va trouver plusieurs groupes (Clusters) constitués de 4 processeurs, ces derniers sont connectés entre eux à l'aide d'un bus unique, puis dans un niveau hiérarchique plus élevé ces différents groupes de processeurs seront connectés à l'aide d'un autre type de réseau du type direct (Point-à-Point).

4.4 Les réseaux sur puces

4.4.1 Gestion des réseaux sur puce

L'adaptation du modèle de référence d'interconnexion des systèmes (OSI : Open Systems Interconnection) aux réseaux sur puces a été proposée afin de permettre une abstraction des propriétés électriques, logiques et fonctionnelles du réseau d'interconnexion [91]. Ce standard est constitué de trois couches principales : couche physique, couche d'architecture et de contrôle et couche logicielle. La couche d'architecture et de contrôle indique la topologie et l'organisation physique du réseau d'intercommunication ainsi que les algorithmes utilisés pour la gestion de la communication. Elle est elle même constituée de trois parties appelées couches réseaux : la première est la couche liaison puis vient la couche réseau et enfin la couche transport. Ces trois couches sont brièvement décrites dans les paragraphes suivants.

4.4.1.1 Couche liaison

Cette couche se charge de transférer des données entre deux points appartenant à un même réseau ou bien de réseaux différents à travers la couche physique. En plus de fournir la possibilité de connecter des modules du réseau, cette couche détecte aussi les différents problèmes de transmission de données associés à la couche physique. La couche de liaison assure le bon séquencement des trames ainsi qu'une retransmission en cas d'erreurs. Une autre fonctionnalité de la couche de liaison est de réguler le trafic sur le réseau afin d'éviter les congestions et les blocages réseaux. On remarquera que cette couche se subdivise en deux parties où la première assure un trafic régulé sans congestion (Couche MAC : Medium Access Control) et où la deuxième se charge de la détection d'erreurs (LLC : Logical Link Control).

4.4.1.2 Couche réseau

Dans les systèmes sur puce standard l'ensemble des modules du système se partagent un médium de communication consistant souvent en un bus. Dans ce cas la couche réseau est vide. Cette couche intervient lorsque plusieurs éléments sont connectés à travers différents réseaux. Cette couche assure alors un chemin de communication de bout en bout en passant par les voies de communication des voisins directs. La couche réseau se caractérise à travers ces deux fonctionnalités : le routage et le relayage. Ces deux points constituent les points durs d'une communication multiprocesseur et ont fait l'objet de plusieurs études [92, 93].

4.4.1.3 Couche transport

Cette couche représente celle de plus haute abstraction dans les couches réseaux. Celle-ci se charge d'encapsuler les messages en des paquets lors de la transmission et de les décomposer à la destination. Cette couche se préoccupe aussi des corrections d'erreurs dans les messages, de leur bon ordonnancement et leur perte.

4.4.2 Etat de l'art des réseaux sur puces

Les travaux réalisés et proposés sur l'implémentation des réseaux sur puce ont couvert un large spectre de recherche. La spécification à un haut niveau d'abstraction, le choix de la topologie et l'implémentation physique, constituent les trois problèmes les plus abordés dans ce domaine. Nous trouvons dans ces travaux différents niveaux de spécification. Des travaux ont privilégié la montée en abstraction afin de simplifier et accélérer la conception. SystemC ou SystemC niveau TLM ont été utilisés afin de répondre à ces contraintes. Cependant, ces méthodes n'ont pas encore tout à fait défini une implémentation directe sur circuit. Effectivement, un important travail manuel reste nécessaire afin de réaliser l'implémentation physique. Différentes architectures de réseaux sur puce allant du 2D mesh, Torus, Fat-tree...etc, ont été étudiées. Enfin, des implémentations sur FPGA et ASIC de ces réseaux sont venues confirmer les potentialités de ce paradigme de conception de systèmes sur puce, centré sur la communication. Le tableau 4.2 illustre l'ensemble des réseaux sur puces les plus connus réalisés ces dernières années.

Réseau NoC	Topologie et routage	Taille paquets	Surface	Interface Routeur	Performance	QoS	Implémentation
SPIN-2000 [94]	Fat-tree / Deterministic and adaptive	32-bit data + 4 bits control	$0.24mm^2$ CMOS $0.13\mu m$	VCI	2 Gbits/s per switch	NA	ASIC layout 4.6 mm^2 CMOS 0.13 μm
aSOC-2000 [95]	2D mesh / Determined by application	32 bits	50,000 transistors	NA	NA	Circuit-switching (no wormhole)	ASIC layout CMOS 0.35 μm
Dally-2001 [76]	Folded 2D torus / XY source	256 bits data + 38 bits control	$0.59mm^2$ CMOS $0.1\mu m$	NA	4 Gbits/s per wire	GT - virtual channels	No
Nostrum-2001 [96]	2D mesh / Hot potato	128 bits data + 10 bits control	$0.01mm^2$ CMOS $65nm$	NA	NA	NA	NA
Sgroi-2001 [88]	2D mesh	18 bits data + 2 bits control	NA	OCP	NA	NA	NA
Octagon-2002 [97]	Chordal ring / Distributed and adaptive	Variable data + 3 bits control	NA	NA	40 Gbits/s	Circuit-switching	No
Marescaux-2002 [98]	2D torus / XY blocking, hop-based, deterministic	16 bits data + 3 bits control	611 slices VirtexII	Custom	320Mbits/s per virtual channel at 40 MHz	2 virtual channels (to avoid deadlock)	FPGA VirtexII / VirtexII Pro
Bartic - 2003 [99]	parameterizable links / Deterministic, virtual-cut-through	Variable data + 2 bits control	552 slices + 5 BRAMs VirtexII Pro	Custom	800Mbits/s per channel for 16-bit flits at 50 MHz	Injection rate control, congestion control	FPGA VirtexII Pro
Æthereal-2005 [100, 101, 90]	2D mesh / Source	32 bits	$0.26mm^2$ CMOS $0.12nm$	DTL	80Gbits/s per switch	Circuit-switching	ASIC layout
Eclipse-2002 [102]	2D sparse hierarchical mesh	68 bits	NA	NA	NA	NA	No

Proteo-2002 [103, 104, 105]	Bi-directional ring	Variable control and data sizes	NA	VCI	NA	NA	ASIC layout CMOS 0.18μm
SOCIN-2002 [106]	2D mesh / XY source	n bits data + 4 bits control	420 LCs APEX FPGAs	VCI	1 Gbits/s per switch at 25 MHz	No	No
SoCBus-2002 [107]	2D mesh / XY adaptive	16 bits data + 3 bits control	NA	Custom	2.4 Gbits/s per link	Circuit-switching	No
QNOC-2003 [108]	2D mesh / XY	16 bits data + 10 bits control	0.02mm^2 CMOSnm	Custom	80 Gbits/s per switch for 16-bit flits at 1GHz	GT - virtual channels, (4 different traffic)	No
T-SoC-2003 [46]	Fat-tree / Adaptive	38 bits	NA	Custom/OCP	NA	GT - 4 virtual channels	NA
Xpipes-2002 [109]	Arbitraire / Source static	32, 64 or 128 bits	0.33mm^2 CMOS 100nm	OCP	64 Gbits/s per switch for 32-bit flits at 500MHz	No	No
Hermes-2004 [110, 111]	2D mesh / XY	8 bits data + 2 bits control	555 LUTs 278 slices VirtexII	OCP	500 Mbits/s per switch at 25 MHz	No	FPGA VirtexII
MANGO 2005 [112]	2D mesh	32 bits data + 2 bits control	CMOS 0.12 μm / 0.188 mm^2	OCP	NA	BE / GS	No
Faust 2005 [113]	2D mesh	32 bits data + 4 bits control	CMOS 0.13 μm / 35K Gates	OCP	NA	BE / GS	ASIC

TAB. 4.2 – Etat de l'art des réseaux sur puce

Ce bref état de l'art, prouvant les larges potentialités des réseaux sur puce, n'est pas anodin car il vient renforcer notre choix d'architecture multiprocesseur sur puce. L'architecture réseau sur puce que nous avons réalisée sera présentée dans la section 4.6 de ce chapitre.

4.5 Modèle de programmation

L'importance d'un modèle de programmation pour les architectures multiprocesseurs sur puce est illustrée par les figures 4.14 et 4.15. La figure 4.14 correspond à une situation sans modèle de programmation. Dans ce cas, l'organisation de la communication met en oeuvre toutes les couches successives : API, HAL (couche logicielle d'adaptation au matériel) et OS spécifique jusqu'au réseau d'interconnexion. Le concepteur du logiciel doit connaître le partitionnement sur l'architecture des tâches de l'application dès les premières étapes de la conception de son logiciel.

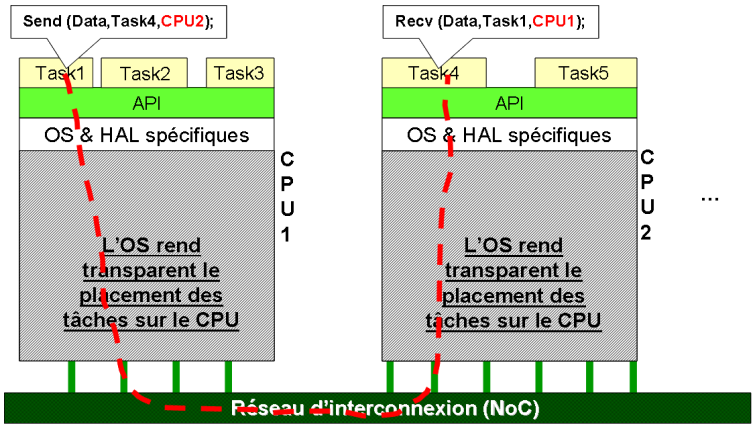


FIG. 4.14 – Communication entre processeurs sans modèle de programmation

La figure 4.15 présente la situation avec un modèle de programmation. Dans ce cas, le détail de l'architecture du multiprocesseur sur puce sous-jacent n'est pas visible aux différentes tâches de l'application. Le partitionnement, le placement, la communication et la synchronisation sont ainsi pris en compte via le modèle de programmation.

Nous donnons, dans ce qui suit, une présentation succincte de modèles de programmation ainsi que le type de programmation utilisé et les raisons de son choix dans cette étude.

4.5.1 Modèle de programmation parallèle

Le modèle de programmation parallèle permet d'entamer la spécification logicielle sans avoir au préalable en main l'architecture matérielle. Il tente de fournir

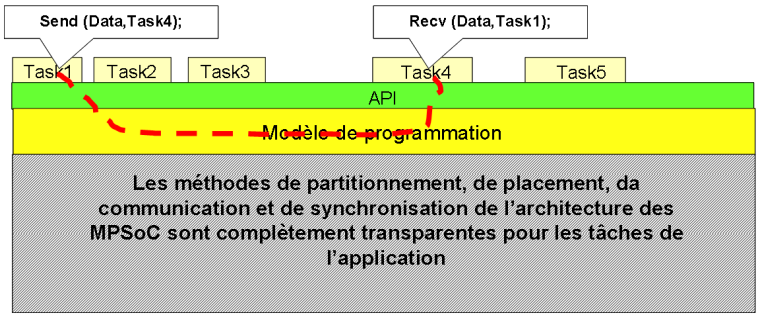


FIG. 4.15 – Communication entre processeurs avec modèle de programmation

au concepteur (programmeur) seulement les détails architecturaux lui permettant une implémentation efficace de l'application [114]. Aussi, le modèle de programmation parallèle ouvre la possibilité d'avoir assez tôt dans le cycle de conception les premières estimations de performance. Le fait que ceci se fasse à un niveau élevé d'abstraction, nous permet de concevoir du code portable complètement indépendant de l'architecture. Il existe différentes architectures de machines parallèles et malgré leur diversité on pourra regrouper la majorité de ces machines en trois catégories :

- Les architectures où la communication se fait par échange de messages
- Les architectures à mémoire partagée
- Accès direct à une mémoire distribuée sur le réseau

Cependant, le type de communication dans un MPSoC n'est pas forcément tributaire de l'architecture mémoire. Il est tout à fait possible de simuler, par exemple, une communication par échange de messages sur une architecture à mémoire partagée.

Afin de passer à la programmation d'architectures MPSoC, les programmeurs ont été contraints de passer vers un modèle de programmation haut niveau permettant de faire abstraction du matériel facilitant ainsi ce mode de programmation et améliorant aussi sa portabilité.

Dans ce chapitre nous n'avons pas proposé une méthodologie de programmation parallèle (voir chapitre 5). Cependant, il est nécessaire de signaler l'importance de celle-ci. Le flot de conception proposé ici peut être entièrement efficace que si la spécification de l'application parallèle en entrée est décrite de manière optimale. Un bon modèle de programmation doit fournir une spécification efficace et rapide de l'application. Il représente aussi le premier point à optimiser lors de la conception d'un système MPSoC. Le compromis entre abstraction et flexibilité constitue un point crucial dans la méthodologie de conception MPSoC.

4.5.1.1 Modèle de programmation parallèle bas niveau

Selon le type de l'architecture des machines parallèles que nous voulons programmer, il nous faut développer un noyau d'exécution afin de prendre en compte leur matériel et leurs caractéristiques. Avec ce modèle de programmation, on doit donc s'adapter par rapport à l'architecture sur laquelle on travaille. Pour les architectures à mémoire partagée, il faut créer tout d'abord les processus puis passer à leur synchronisation en spécialisant l'ordonnancement à l'architecture. Cette synchronisation de processus se fait à l'aide de verrous ou variables de condition, proposés par des standards tels que les "threads Posix" et OpenMP. Au niveau matériel, la mémoire partagée avec des processeurs possédant des caches implique de garantir la cohérence des caches, ce qui est faisable de manière relativement classique avec une interconnexion à bus (espionnage de bus), mais devient complexe avec un réseau d'interconnexion multi-étages. Sur une machine à mémoire distribuée, nous aurons besoin de passer des communications entre les différents processeurs afin d'atteindre la donnée située dans chacune de leur mémoire associée. Il faudrait arriver donc à communiquer entre deux noeuds, cela se fait à l'aide de bibliothèques de communications développées au préalable. Parmi ces bibliothèques on peut citer les implémentations MPI. Enfin, sur le dernier type d'architectures multiprocesseur en l'occurrence les réseaux de multiprocesseurs à mémoire partagée, des noyaux d'exécution ont été développés afin d'arriver à fusionner processus et communication, on pourra citer comme exemple : Nexus, PM2 et Athapascan. Il est à noter que des études comparatives entre différents modèles de programmation ont été réalisées comme dans [115] comparant par exemple MPI à MPI+OpenMP.

Modèle de programmation bas niveau :

- Programmation efficace et optimisée
- Mais, moins flexible et moins portable vers de nouvelles architectures matérielles

4.5.1.2 Modèle de programmation parallèle haut niveau

Ce modèle de programmation parallèle a été mis au point afin de faciliter la programmation et augmenter l'aspect portabilité des applications. Pour atteindre une grande facilité de programmation parallèle, il faudrait que le programmeur puisse faire abstraction de tout ce qui est réseau de communication ainsi que des processeurs. Nous nous intéresserons aussi dans ce modèle de programmation parallèle à l'extraction du parallélisme d'une application car ce dernier peut être soit explicite (apparent) et donc sera décrit directement au moment de la programmation, soit il peut être implicite et dans ce cas-ci le parallélisme sera extrait lors de la compilation ou bien de l'exécution.

Modèle de programmation haut niveau :

- Programmation moins efficace et moins optimisée
- Mais, flexible et portable vers de nouvelles architectures matérielles

4.5.2 Modèle de programmation choisi dans cette étude

Il existe principalement deux types de modèles de programmation parallèle : (1) un modèle de programmation implicite et (2) un modèle de programmation explicite. Dans le premier, plusieurs détails d'implémentation sont extraits par le modèle de programmation afin que le concepteur puisse faire une totale abstraction de l'architecture. Le parallélisme au niveau instructions (ILP : Instruction Level Parallelism) fait partie de ce type de modèle de programmation parallèle où le concepteur se base sur la complexité de l'unité de calcul. Le second modèle de programmation est, quant à lui, explicite dans le sens où plusieurs détails de l'architecture restent visibles au concepteur comme : le nombre de processeurs et la mémoire distribuée à travers le système. Notre travail est basé sur ce type de modèle de programmation parallèle car la communication peut être facilement suivie. Par conséquent, le concepteur pourra avoir une meilleure vue du système et des chemins de communication circulant à travers les différentes unités de calcul.

Lors de l'utilisation d'un modèle de programmation parallèle, deux types architectures peuvent être utilisées : (1) mémoire partagée et (2) échanges de messages. Cette approche correspond typiquement aux mémoires distribuées et MPI est la surcouche logicielle standard par-dessus des langages de programmation comme C/C++ et Fortran. En ce qui concerne notre travail, nous avons utilisé une architecture à échange de messages, non pas parce que celle-ci représente plus naturellement un systèmes multiprocesseur mais c'est la nature des multiprocesseurs sur puce qui l'a dicté. La taille de la puce grandissant de plus en plus rend difficile le fait de partager une mémoire avec plusieurs unités de calcul. La difficulté émane du fait qu'il sera fastidieux de fournir un accès symétrique à tous les éléments de calcul à la mémoire partagée. Première conséquence directe à cela, sera le déséquilibre des délais qui rajoutera une difficulté supplémentaire à la conception. De plus, l'utilisation d'une mémoire partagée rend pratiquement impossible la réalisation de circuits asynchrones GALS (Globally Asynchronous and Locally Synchronous) [116]. La nature géométrique deux dimensions de la puce impose ce type de modèle de programmation.

4.6 Plateforme MPSoPC à base de processeurs Microblaze

La figure 4.16 représente la plateforme multiprocesseur que nous avons implémentée sur FPGA. La taille de la plateforme peut varier de 2 processeurs à un maximum de 9 processeurs, limitée pour des raisons de surface sur FPGA (Xilinx Virtex-II 8000). Les processeurs sont connectés entre eux à l'aide de liens FSL (Fast Simplex Link) qui est une IP Xilinx reprenant le fonctionnement d'une FIFO permettant une communication rapide (2 Cycles en lecture et écriture) entre deux

éléments. Cette architecture permet un lien point-à-point entre deux processeurs adjacents mais n'assure pas de communication avec un autre processeur distant.

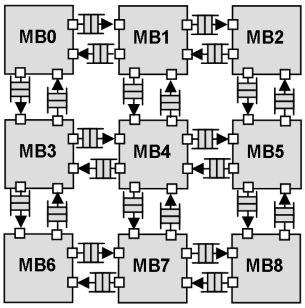


FIG. 4.16 – Architecture multiprocesseur 3x3 à base de Microblaze et de canaux FSL

Le choix du protocole de communication s'est porté sur l'implémentation d'une version logicielle embarquée du protocole MPI [117]. Ceci a été décidé afin d'avoir un aspect portable pour nos applications permettant ainsi de changer plus facilement d'architecture en vue d'une exploration. La figure 4.17 illustre les couches par lesquelles une communication passe entre deux processeurs.

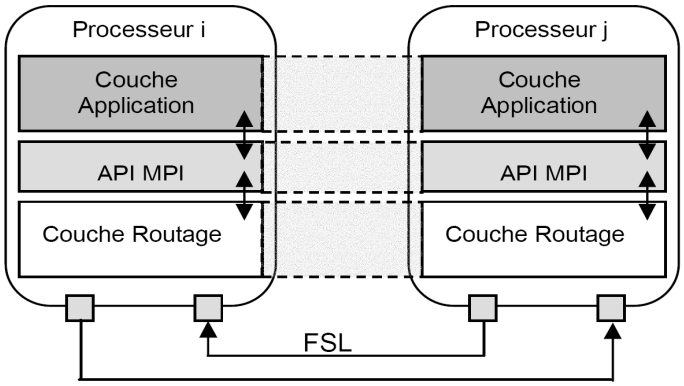


FIG. 4.17 – Couche de communication MPI

Nous avons défini une structure de paquets permettant cette communication. Nous disposons de deux types de paquets où chaque paquet est constitué de 64 bits. La figure 4.18 illustre ces deux types de paquets. Le fait que la largeur du canal FSL soit de 32 bits impose une taille de paquets multiples de 32 bits. Les adresses de source et de destination sont codées sur 8 bits. Nous avons implémenté une routine sur chaque processeur qui se déclenche périodiquement et teste l'état des FSL connectés au processeur. Si un canal FSL contient un paquet, celui-ci est lu immédiatement. Le processeur lit le paquet. Si ce dernier lui est destiné, il est directement rangé dans un tampon de réception (par logiciel). Autrement le processeur le transmet avec tous les paquets suivants vers le processeur voisin.

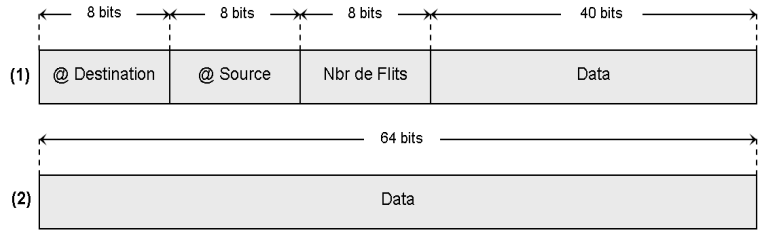


FIG. 4.18 – Paquets MPI

Nous avons opté pour un routage de type Wormhole XY. Cette technique est adaptée aux systèmes sur puce du fait qu'elle ne consomme pas beaucoup de ressources mémoire. Chaque processeur est doté d'une table de routage statique lui permettant de router les paquets à transférer. Cette table est paramétrable et peut être adaptée à la topologie du réseau ainsi qu'à l'algorithme de routage.

Nous présentons ci-dessous en pseudo code l'algorithme de routage inspiré du protocole MPI que nous avons implémenté sur notre plateforme multiprocesseur sur puce programmable. Nous précisons que la routine de routage se déclenche par interruption à chaque fois qu'une donnée est présente dans le tampon de réception. Cette interruption interrompt l'exécution de l'application en cours jusqu'à ce que la tâche de routage soit achevée.

Algorithm 4.6.1: ROUTAGE_X_Y()(1)

comment: Routine de routage

```
for eachFSLchannel
Get(&Packet)
 if Packet! = NULL
  if Packet.Destination == Local_Address
   for 1 to Packet.Count
   Save(Packet, Buffer)
   Get(&Packet)
  else
   for 1 to Packet.Count
   Put(&Packet, Routing_Table[Packet.Destination])
   Get(&Packet)
```

Le système doit être, par la suite, capable de reprendre l'exécution de l'application interrompue. Nous soulignons le fait que l'utilisation d'un μOS aurait pu être entreprise. Cependant, l'utilisation d'un μOS en l'occurrence dans notre cas XMK (Xilinx Micro kernel), exige 16 Ko de mémoire par processeur dont 8 Ko seulement pour l'OS. Notre priorité aussi est de laisser un maximum de ressources mémoire

pour les buffers de communication (FSL : Modules qui rentreront dans l'exploration d'architectures multiprocesseur)

4.6.1 Implémentation d'une version embarquée de MPI

Nous avons décidé de ne garder de la bibliothèque MPI que ce que nous avons jugé utile pour l'implémentation d'applications embarquées. Par ailleurs, il aurait été impossible et inutile d'implémenter l'ensemble des fonctions MPI d'autant plus que cela n'aurait eu aucun sens dans le contexte de ce type d'applications. Le choix des fonctions MPI a été guidé par le fait d'avoir des fonctionnalités minimales pouvant être portables et bien sûr une utilisation de ressources mémoire minimale. Les fonctions choisies sont illustrées dans le tableau suivant :

Mode bloquant	Mode non bloquant	Synchronisation
MPI_Send()	MPI_ISend()	MPI_Barrier()
MPI_Recv()	MPI_IRecv()	

TAB. 4.3 – Primitives MPI utilisées

Nous avons implémenté un mode bloquant et un mode non bloquant pour les primitives d'envois et de réceptions. Ces routines utilisent le même schéma d'envoi et de réception.

Nous avons opté pour trois types de données :
- MPI_INT (4 octets).
- MPI_CHAR (1 octet).
- MPI_DOUBLE (8 octets).

Les diagrammes de communication pour les routines d'envoi et de réception pour les modes bloquants et non bloquants sont illustrés par les figures suivantes 4.19 et 4.20.

La différence entre ces deux modes d'envoi est que la routine bloquante attend que le buffer d'envoi ait au moins une case pour stocker un paquet, tandis que la procédure non bloquante envoie le paquet sans se soucier de l'état du buffer. Cependant une gestion d'erreur fait en sorte de renvoyer la donnée si celle-ci est perdue durant la précédente tentative d'envoi.

Pour assurer la portabilité de cette implémentation MPI sur n'importe quel réseau, en plus de sa portabilité liée à l'utilisation d'ANSI-C, nous avons décrit un algorithme permettant de créer une table de routage s'adaptant à la topologie du réseau. L'algorithme décrivant la table de routage doit nécessairement être capable de définir un lien entre deux éléments distants en empruntant un chemin optimal, c'est-à-dire le plus court et le moins coûteux. Une description formelle de l'algorithme est donnée ci-dessous :

Soit un graphe de communication $C(V, A)$, où $V = \{v_1, v_2, ..., v_n\}$ représente l'ensemble des sommets du graphe, $A = \{a_{11}, a_{12}, ..., a_{ij}\}$ l'ensemble de ses arêtes et

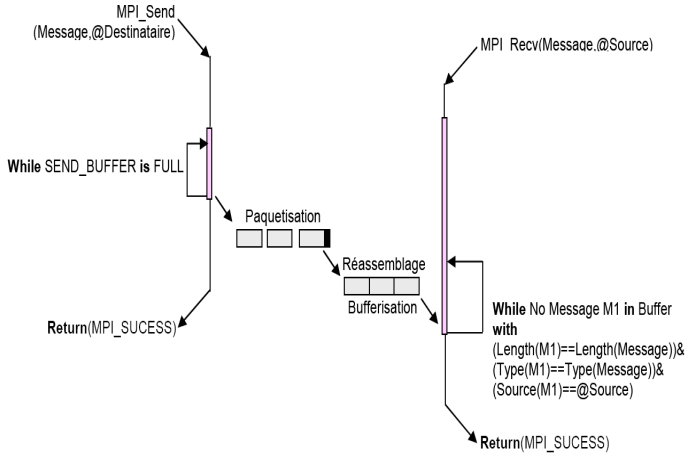


FIG. 4.19 – communication MPI bloquante

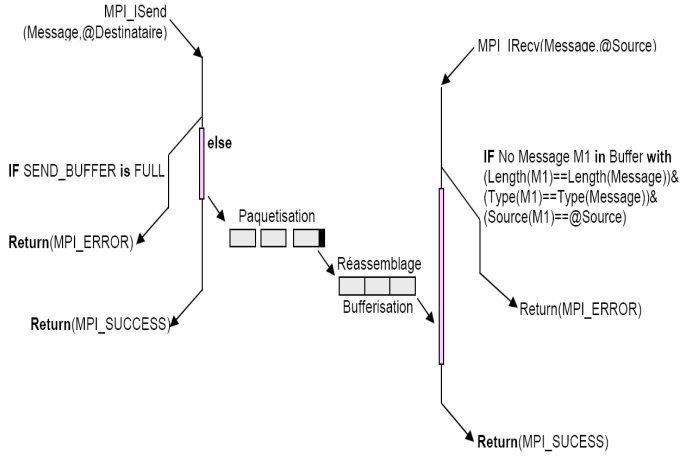


FIG. 4.20 – communication MPI non bloquante

$C = \{c_{ij}, 0 < i, j < n+1\}$ l'ensemble des arcs du graphe. n représente à chaque fois le nombre de noeuds dans le graphe. On définit la fonction de routage R la fonction qui associe à chaque couple de sommets (v_i, v_j) une chaîne c_{ij} :

$$R : A \rightarrow C$$

$$(v_i, v_j) \rightarrow R(v_i, v_j)$$

On définit aussi la quantité de paquets routés par chaque arête du graphe à l'aide d'une fonction "Charge" Ch :

$$Ch : A \rightarrow N$$

$$(v_i, v_j) \rightarrow Ch(v_i, v_j)$$

La valeur liée à la congestion d'un lien est donnée par la fonction $Cong(R) = Max\left\{Ch\left(v_{i),v_j}\right), \left(v_{i),v_j}\right) \in E\right\}$.

Notre algorithme génère les tables de routage avec le plus court chemin sans se soucier de la congestion ni du coût de chaque lien du fait que cela ne nous intéresse pas pour le moment. Par conséquent, nos arêtes sont toutes étiquetées avec un même poids. L'algorithme proposé commence par construire l'arbre de plus courts chemins vers tous les sommets puis en fonction de ce dernier une table de routage est définie.

4.6.1.1 Performance de l'implémentation MPI

Nous présentons ici quelques résultats de performance de routage de la version MPI embarquée implémentée dans ce travail. La figure 4.21 illustre le temps d'exécution de la routine de routage. Cette étude s'est faite sur l'envoi à chaque fois d'un nombre différent de paquets 64 bits. Il est clair que cette routine de routage consomme un temps non négligeable des ressources CPU, ce qui était plus ou moins attendu dans un cas de routage logiciel.

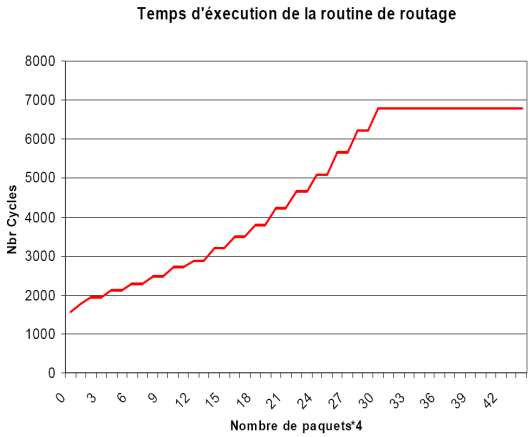


FIG. 4.21 – Performance du routage logiciel : Routine de routage

La figure 4.22 représente l'impact du nombre de sauts dans le réseau sur la performance de routage.

Nous rappelons que nous ne cherchons pas à optimiser les performances de routage mais fournir une plateforme multiprocesseur avec une couche de communication robuste et facilement programmable pour notre application d'exploration d'espace de conception.

Nous avons testé notre implémentation avec un benchmark "Ping Pong". Ce test est l'un des benchmarks les plus utilisés pour l'évaluation des communications. Le benchmark "Ping Pong" permet d'évaluer le temps de routage de l'émission/réception et réception/émission. Les résultats sont illustrés dans la figure 4.23.

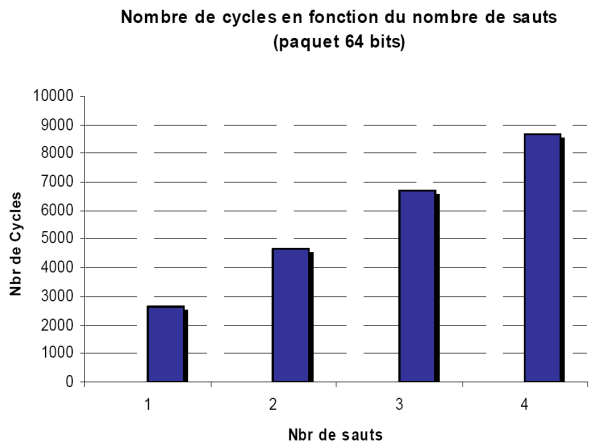


FIG. 4.22 – Impact du Nombre de saut sur la performance de routage

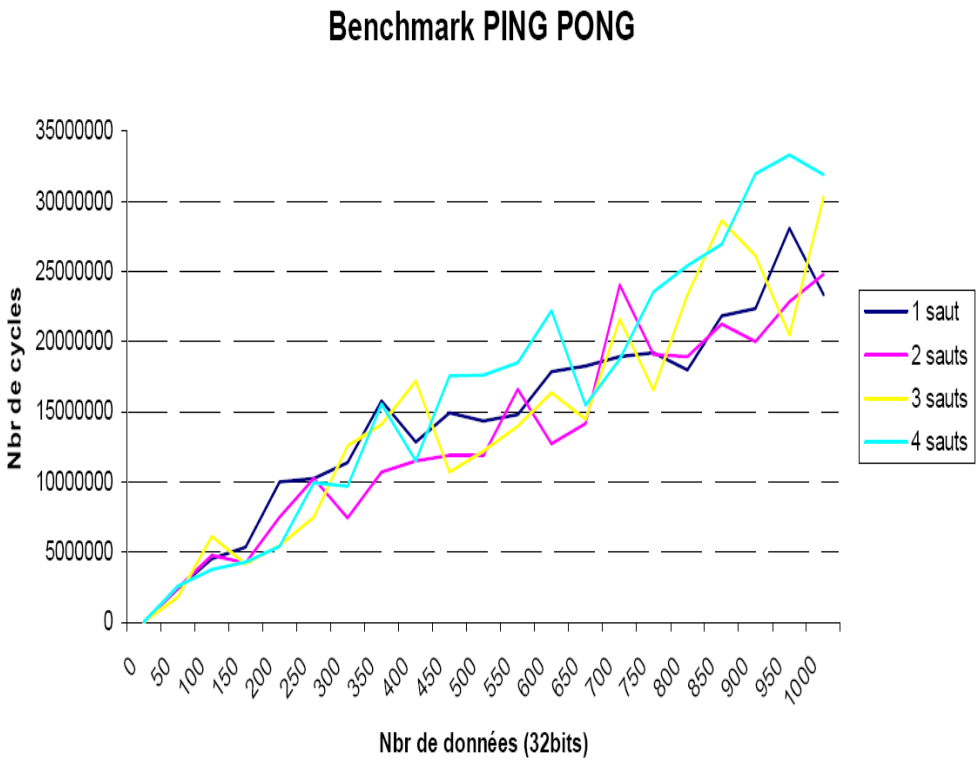


FIG. 4.23 – Benchmark Ping Pong

Nous remarquons sur cette figure que le délai de routage n'est pas tout à fait prévisible et peut dans des cas diminuer alors qu'il devrait augmenter. Cela est dû principalement au routage logiciel qui induit un certain indéterminisme. Même si on sait que la routine de routage se déclenche à des intervalles connus (routage sans interruption) on ne sait pas à quel moment du routage du paquet en cours cela

se fait. Donc selon les cas cela peut plus ou moins augmenter le délai de routage expliquant ainsi l'allure de la courbe.

4.7 Méthodologie d'exploration MPSoPC

Nous proposons dans ce chapitre une méthodologie de conception pour systèmes multiprocesseurs sur puce. Cette méthodologie est basée sur une exploration multiobjectif à base d'algorithmes évolutionnaires. Nous avons intitulé cette méthodologie MOCDEX pour "Multiprocessor On Chip Multiobjective Design Space Exploration with Direct Execution". Le flot général de MOCDEX est présenté dans ce qui suit :

1. Générer aléatoirement une population de configurations MPSoC
2. Pour chaque configuration
   - (a) générer le fichier de spécification de la plateforme matérielle/logicielle
   - (b) générer à l'aide de l'outil de conception et des IPs le modèle matériel/logiciel du MPSoC
   - (c) Synthétiser puis placer et router à l'aide des outils d'aide à la conception électronique
   - (d) Sauvegarder les rapports de placement routage.
   - (e) configurer le circuit FPGA
   - (f) exécuter l'application MPSoC et récupérer les temps d'exécution en nombre de cycles
   - (g) ranger la solution
3. génére une nouvelle population avec l'algorithme évolutionnaire multi-objectif
4. si le front de Pareto n'est pas satisfaisant ou le nombre de générations n'est pas atteint, aller à 3
5. le front de Pareto final est disponible

Les fonctions objectif que nous avons fixées ici concernent le nombre de blocs de RAM, de slices utilisés sur le FPGA et le temps total que prend l'exécution de l'application multiprocesseur sur la plateforme MPSoC. L'algorithme évolutionnaire multiobjectif évalue chaque configuration par rapport à ces fonctions objectif.

Comme le montre la figure 4.24, seule l'exploration de l'espace de configuration (algorithme évolutionnaire multiobjectif) et la synthèse placement routage sont effectuées sur un ordinateur. L'exécution de la plateforme se fait sur une carte FPGA reliée à l'hôte via une connexion PCI afin de communiquer les résultats de temps d'exécution.

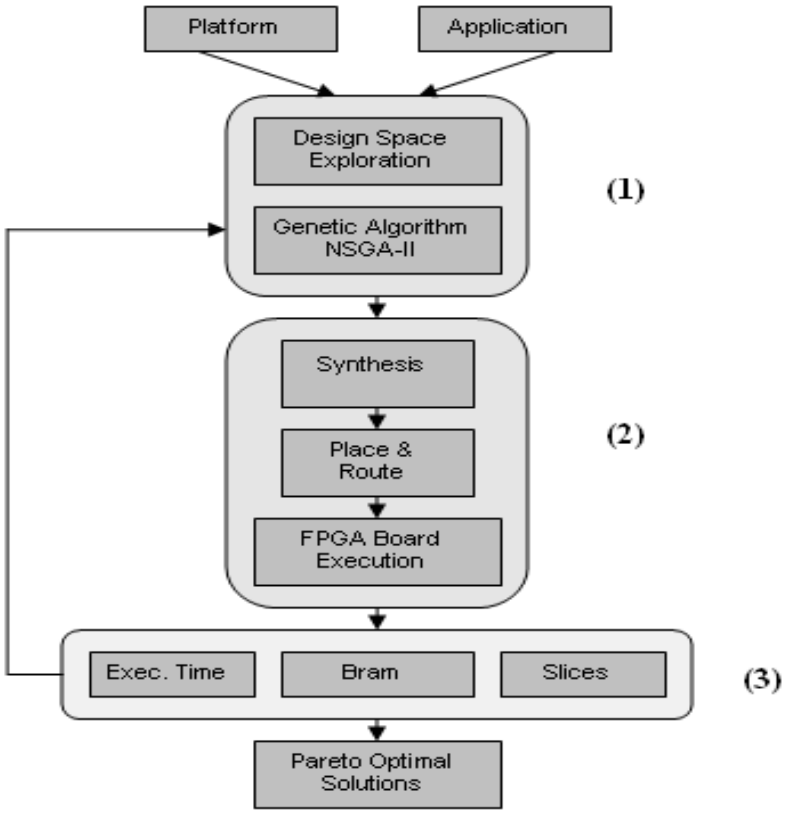


FIG. 4.24 – Flot d'exploration

4.8 Exploration : cas d'études MPSoPC en grille 2D

L'architecture multiprocesseur sur puce programmable utilisée dans l'exploration multiobjectif étudiée ici, est celle présentée précédemment dans ce même chapitre. Nous rappelons que celle-ci est réalisée à base de processeurs logiciels Microblaze communiquant à l'aide de connections FIFO FSL. Une couche logicielle de communication y a été implémentée. Celle-ci utilise des fonctions de communication MPI et reprend un mécanisme de communication en Wormhole. L'architecture est illustrée par la figure 4.25.

Cette plateforme représente un système multiprocesseur 2x2 exécutant trois algorithmes de traitement d'images en pipeline. Nous présentons dans la section 4.8.1 qui suit plus de détail sur la manière dont cette plateforme s'exécute.

4.8.1 Application de traitement d'images

La figure 4.26 illustre l'architecture de filtrage d'images implémentée sur la plateforme multiprocesseur sur puce programmable (Virtex-II 8000). Comme nous pou-

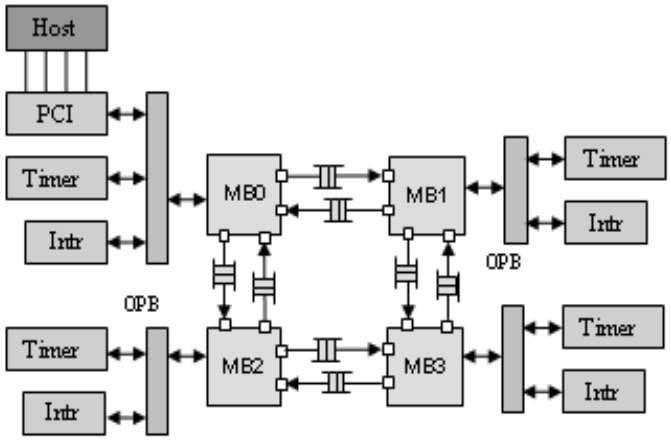


FIG. 4.25 – Architecture multiprocesseur filtrage 2x2

vons le remarquer à travers cette figure, nous avons fait le choix d'exécuter trois algorithmes de traitement d'images de manière pipelinée. Ce type d'exécution optimisera la performance et le taux d'utilisation des mémoires, rendant par conséquent l'application facilement implémentable. Effectivement, en adoptant cette technique, on aura un maximum de trois lignes stockées à la fois dans la mémoire de chaque processeur plutôt qu'une image entière. Les lignes de l'image restantes entreront une par une dans les FIFOs de communication respectives à chaque processeur. Le processeur MB0 (Microblaze 0) reçoit les données de l'image à partir de l'ordinateur hôte à travers le bus PCI.

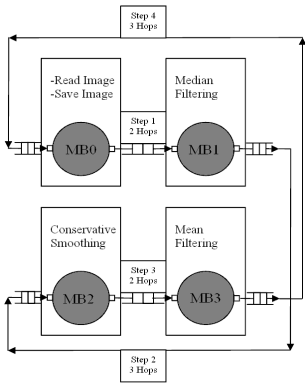


FIG. 4.26 – Plateforme traitement d'images

Une IP a été réalisée afin d'assurer la communication entre la plateforme FPGA et l'hôte. Aussi, des fonctions d'écriture et de lecture vers et depuis le FPGA ont été spécifiées et rajoutées à l'API Alpha-Data afin de faciliter ces tâches. Le processeur MB0 transfère les lignes de l'image, aussi tôt qu'il les reçoit de l'hôte, vers le processeur MB1. Ce dernier effectue un filtrage médian classique dès que les données

de la ligne sont reçues. Ces données traitées sont ensuite directement envoyées au processeur MB2 où elles passent par un filtre conservateur. Enfin, on exécute un dernier traitement de filtrage moyenneur sur le processeur MB3 avant de renvoyer les lignes de l'image complètement traitées vers la hôte. Nous pouvons visiblement mentionner que les trois opérations ont un comportement clairement différent et emploient des opérateurs arithmétiques différents. Ainsi, le temps d'exécution pour chaque algorithme de filtrage diffère et par conséquent implique une occupation inégale des FIFOs de communication. Nous rappelons que le but de notre étude est d'avoir une distribution optimale d'utilisation de mémoires. Par conséquent, l'application utilisée doit être naturellement déséquilibrée pour analyser objectivement le problème. On fait une hypothèse de départ, stipulant que les différents algorithmes sont implémentés de manière optimale au sens de la performance et que la seule amélioration qui reste à faire viendra du bon paramétrage de la taille des FIFOs de communication et des différentes mémoires caches.

4.8.2 Chromosome et espace d'exploration

Le tableau 4.4 illustre l'espace d'exploration fixé pour cette recherche. Cet espace concerne les 4 processeurs de la plateforme MPSoC. Notre objectif ici est de trouver un compromis optimal d'utilisation des blocs de RAM du FPGA partagés entre les FIFOs de communication et les caches d'instructions et de données de l'ensemble des processeurs.

Procs	FSL1 en sortie	FSL2 en entrée	D-Cache	I-Cache
MB0	16..2048	16..2048	512..4096	512..4096
MB1	16..2048	16..2048	512..4096	512..4096
MB2	16..2048	16..2048	512..4096	512..4096
MB3	16..2048	16..2048	512..4096	512..4096

TAB. 4.4 – Espace d'exploration

Cela implique un compromis entre les ressources utilisées et la performance.

Le nombre possible de configurations différentes est donné par le produit du nombre de configurations pour chaque paramètre architectural configurable. Chaque mémoire cache peut avoir jusqu'à 4 tailles différentes et chaque FIFO jusqu'à 8 tailles différentes. Tout l'espace de conception représente $(4 \times 4 \times 8 \times 8)^4 = 2^{40}$ configurations. Si chaque évaluation de configuration exigeait 1 seconde, le temps total d'évaluation serait de 34865 années. Dans ce cas-ci, une technique d'évaluation exhaustive est irréalisable. Une méthodologie d'exploration basée sur un algorithme évolutionnaire peut efficacement gérer ce très large espace d'exploration tandis que la simulation est clairement surpassée par l'exécution directe sur de gros FPGA.

4.8.3 Description du flot d'exploration

Le flot de conception automatique proposé décrit par la figure 4.24 peut être appliqué avec les outils de conception SOPC de Xilinx et de l'environnement Alpha-Data. Le flot se compose principalement de 3 parties : (1) moteur d'exploration de l'espace d'architectures, (2) conception physique (Synthèse, Placement Routage), et enfin (3) plateforme FPGA sur port PCI. La partie relative à l'exploration de l'espace d'architectures commande la totalité du flot et est exécutée sur un ordinateur. Le DSE (Design Space Exploration) spécifie les paramètres architecturaux des configurations multiprocesseurs à évaluer puis traduit ces paramètres en fichier de spécification pour l'outil de conception Xilinx.

1. MOCDEX (Plateforme FPGA Xilinx)
2. Générer aléatoirement une population de configurations MPSoC (Variation sur les caches et les FSL) avec NSGA-II
3. Pour chaque configuration
   - (a) générer les fichiers de spécification de la plateforme matérielle/logicielle (mhs, mss, pao, mss, mld, mdd)
   - (b) générer à l'aide de l'outil de conception XPS de Xilinx et des IPs le modèle matériel/logiciel du MPSoC
   - (c) Synthétiser puis placer et router à l'aide d'outil Xilinx ISE 6.3
   - (d) Sauvegarder les rapports de placement routage générés par Xilinx ISE 6.3.
   - (e) configurer le circuit FPGA avec l'API ADM-XRC d'Alpha-Data
   - (f) exécuter l'application MPSoC et récupérer les temps d'exécution en nombre de cycles
   - (g) ranger la solution
4. générer une nouvelle population avec l'algorithme avec NSGA-II
5. si le front de Pareto n'est pas satisfaisant ou le nombre de générations n'est pas atteint, aller à 3
6. le front de Pareto final est disponible

Pour la réalisation du flot, nous nous sommes inspirés du flot d'exploration de plateforme monoprocesseur proposé dans le chapitre 3. L'outil XPS de Xilinx à été lancé à partir d'un code C où nous avons repris toutes les commandes lignes afin de les exécuter de manière automatique. Le code C est chargé de lire des fichiers de spécification afin de lancer la partie du flot relative à la conception physique en synthétisant et en plaçant routant la configuration multiprocesseur sur le circuit FPGA. Les résultats de ressources consommées sont lus à partir des fichiers placement routage générés par Xilinx ISE 6.3. Le bitstream relatif à chaque configuration est chargé sur la carte FPGA pour exécution et récupération des résultats de performance (nombre de cycles d'exécution). Le nombre de cycle d'exécution est obtenu

grâce à un timer connecté au bus OPB du Microblaze (MB0). Ce dernier est déclenché au lancement de l'exécution puis arrêté une fois celle-ci arrive à sa fin et compte le nombre de coups d'horloge. Ces résultats d'exécution sont communiqués à l'ordinateur hôte à travers le bus PCI. L'ensemble des résultats (Blocs RAM, Slices et temps d'exécution) est injecté comme entrées pour l'algorithme d'exploration pour la prochaine génération exécutée.

4.9 Résultats d'exploration MPSoPC

Nous avons d'abord lancé deux explorations. La première exploration a été lancée sur 10 générations et 22 individus par population (242 implémentation en comptant la génération d'initialisation). La figure 4.27 représente l'ensemble de Pareto rang 1 obtenu avec cette exploration.

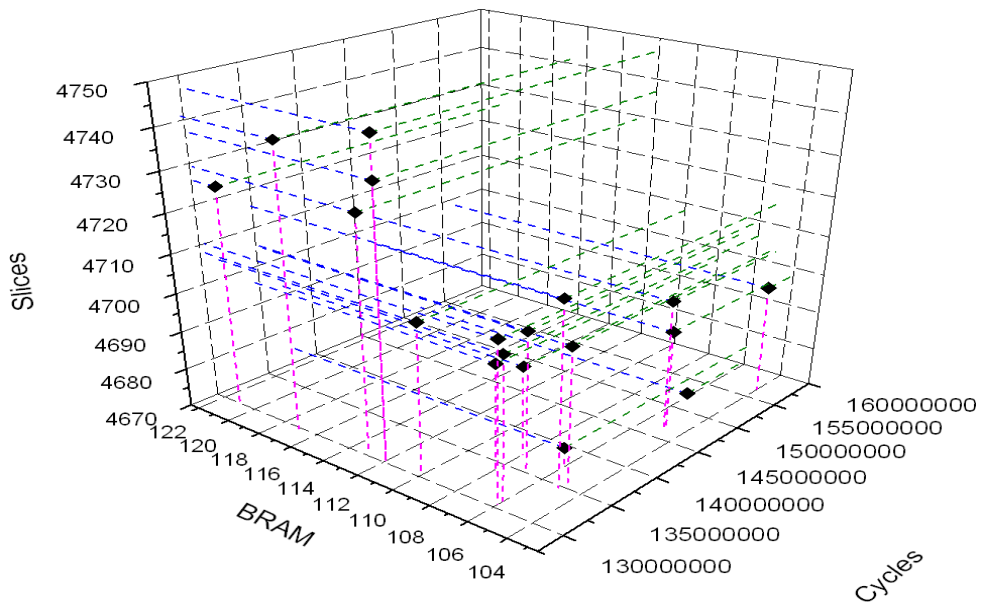


FIG. 4.27 – Exploration génétique nbre population=22, nombre génération=10

Nous avons par la suite lancé une deuxième exploration en augmentant le nombre de génération à 14 et le nombre d'individus à 30. Ceci a été fait dans le souci d'évaluer l'intégrité du flot et sa capacité de convergence. La figure 4.28 illustre les résultats relatifs à cette deuxième exploration.

A partir de ces résultats, nous remarquons une meilleure convergence de l'algorithme comme cela était attendu. Il est tout à fait clair que durant les deux exécutions de flots précédentes, l'utilisation des slices sur FPGA ne change pas beaucoup car nous avons choisi d'utiliser des mémoires pour l'implémentation des FIFO de communication, et le fait d'augmenter leur taille engendre bien sûr une

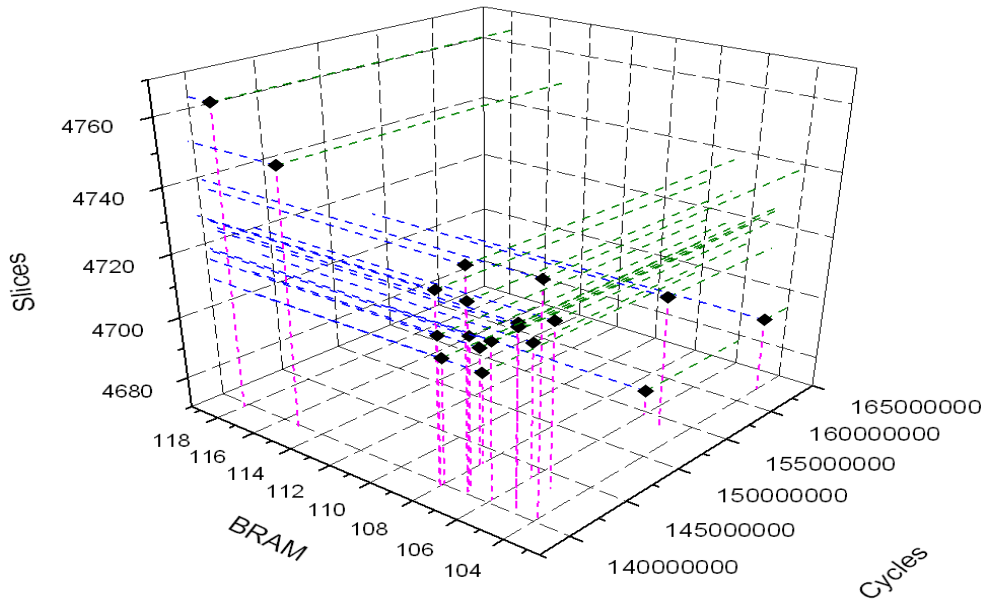


FIG. 4.28 – Exploration génétique nbre population=30, nombre génération=14

augmentation d'utilisation de slices, sauf que cette utilisation reste minime. Cela aurait évidemment été complètement le contraire si nous avions fait le choix de passer par de la mémoire sur slices. Vu ce résultat nous avons décidé de continuer l'exécution d'autres flots en changeant le nombre de générations et d'individus mais dans ce cas en portant l'attention plus sur les ressources en mémoire utilisées et les performances. Pour cette deuxième partie d'exécution nous avons fixé le nombre d'individus à 30 et varié le nombre de générations à 30 puis 60.

Les résultats pour chacune des exécutions sont respectivement illustrés par les figures 4.29 et 4.30.

Ces figures montrent d'après leur forme une bonne convergence de l'algorithme NSGA-II permettant ainsi une bonne interprétation des résultats. On remarquera sur ces deux figures que nous avons obtenu des résultats différents de performance avec plus ou moins la même occupation en mémoire. Cela implique que l'augmentation des ressources mémoire n'implique pas systématiquement de meilleurs résultats. Cependant, de meilleures performances sont obtenues en équilibrant l'utilisation de la mémoire et en essayant de la fournir plus simplement aux ressources qui l'exigent.

Les figures 4.31 et 4.32 représentent respectivement la distribution sur le dernier flot de Pareto de la performance et de l'utilisation des ressources mémoire. Les importants changements dans l'utilisation des BRAMS impliquent du côté de la performance, des changements plus ou moins minimes.

Cela démontre clairement l'importance de la bonne distribution de mémoires embarquée.

Des exemples numériques relatifs aux configurations du dernier flot de Pareto

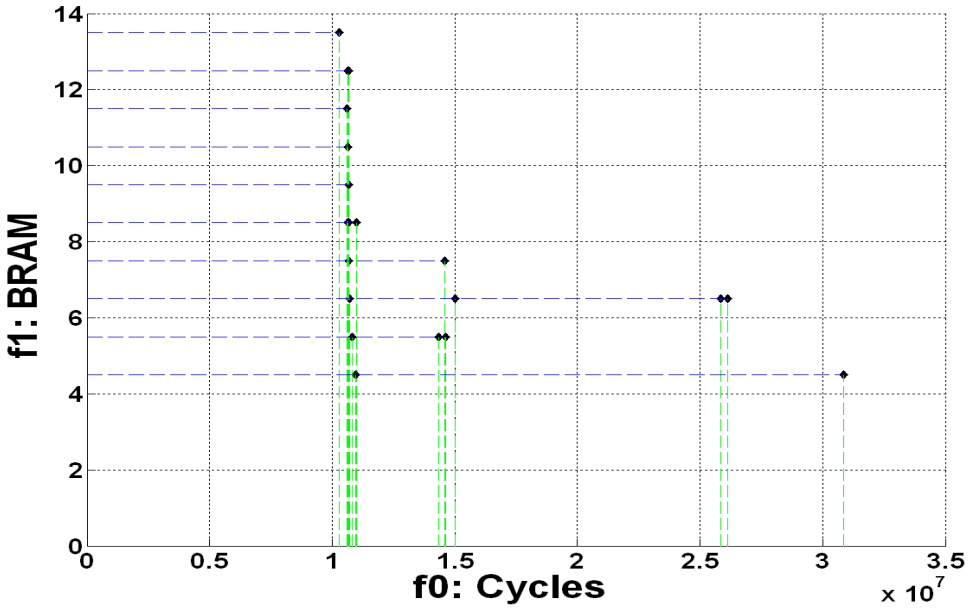


FIG. 4.29 – Exploration génétique nbre population=30, nombre génération=30

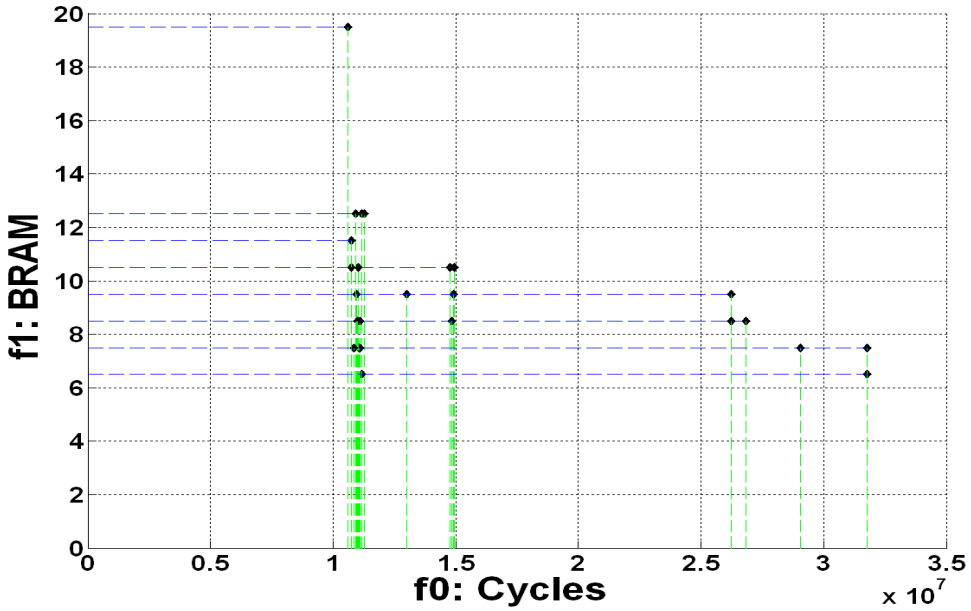


FIG. 4.30 – Exploration génétique nbre population=30, nombre génération=60

sont représentés dans les tableau 4.5 et 4.6. Ces deux individus représentent respectivement un taux d'utilisation de 69,64 % et 64,88 % de la mémoire totale disponible sur FPGA.

Une réduction du taux d'utilisation de la BRAM de 6,8 % est réalisée avec une amélioration très faible de la performance de 0,009 % sur la deuxième configuration.

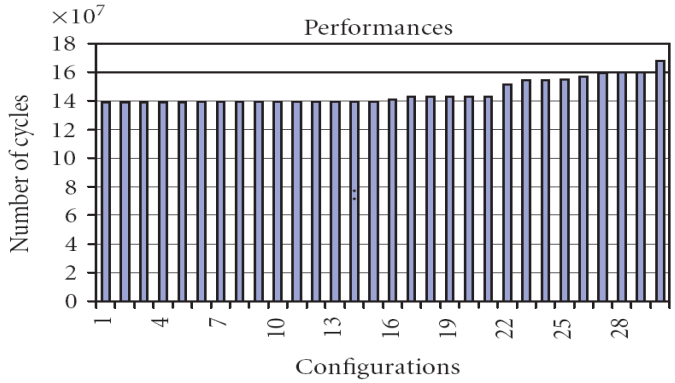


FIG. 4.31 – Distribution de la performance sur le front de Pareto

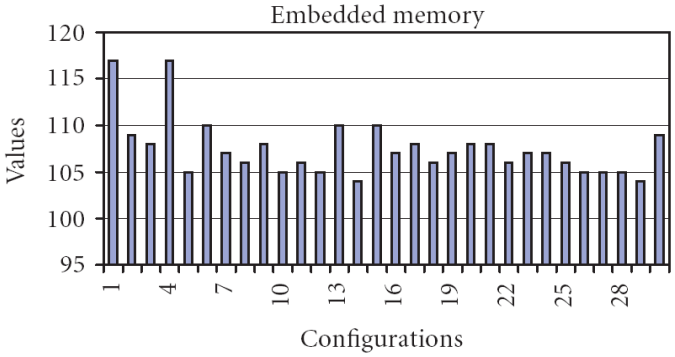


FIG. 4.32 – Distribution de l'utilisation des Blocs de RAM (BRAM) sur le front de Pareto

Procs	FSL1Out	FSL2Out	D-Cache	I-Cache
MB0	2048	2048	1024	4096
MB1	512	512	1024	1024
MB2	2048	512	2048	2048
MB3	1024	1024	4096	4096

TAB. 4.5 – Pareto cycles : 138,844,064 BRAM :109

Procs	FSL1Out	FSL2Out	D-Cache	I-Cache
MB0	2048	128	2048	2048
MB1	256	32	2048	512
MB2	512	16	4069	512
MB3	1024	32	512	2048

TAB. 4.6 – Pareto cycles : 138,974,816 BRAM :117

4.9.1 Analyse statistique de l'espace exploré

Nous avons analysé en détail la complexité de l'espace d'exploration et nous avons illustré la distribution de configurations pour les slices, les BRAM et enfin la performance en nombre de cycles respectivement par la figure 4.33, 4.34 et 4.35.

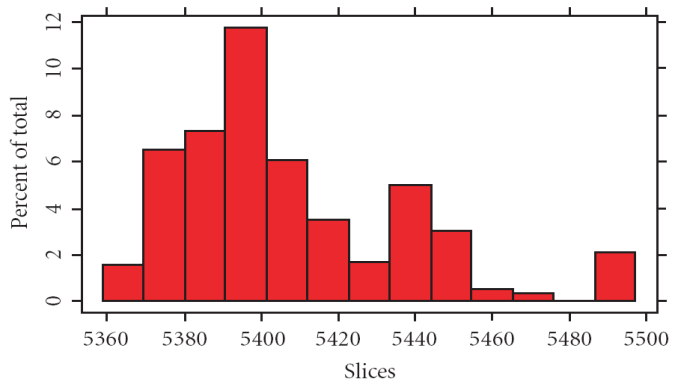


FIG. 4.33 – Espace exploré : Histogramme Slices

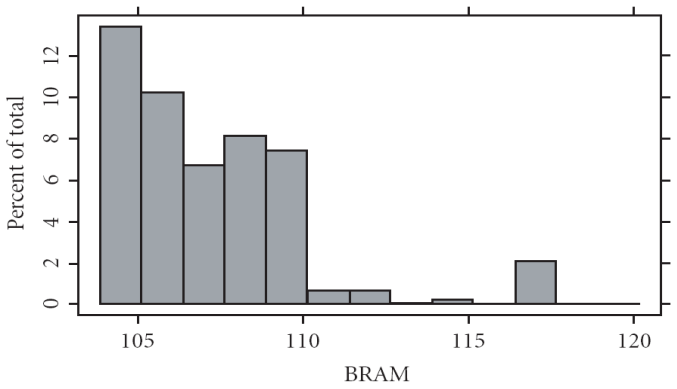


FIG. 4.34 – Espace exploré : Histogramme BRAM

Nous remarquons clairement à partir de ces histogrammes de slices, de BRAM et de performance que leur distribution dans l'espace d'exploration est différente et bien étalée sur tout l'intervalle, démontrant ainsi que l'espace d'exploration ne s'est pas confiné en un sous espace mais au contraire qu'un ensemble large et varié de configurations multiprocesseurs a été exploré.

L'espace exploré est donné par la figure 4.36. La figure 4.36 démontre la complexité de la conception et le besoin d'adapter cette complexité avec des techniques mathématiques d'optimisation appropriées.

Les résultats réalisés précédemment ont exigé l'évaluation de performances de 3120 configurations différentes multiprocesseurs sur puce. Ces évaluations sont "cycle accurate" vu qu'elles ont été obtenues après exécution réelle sur une puce FPGA

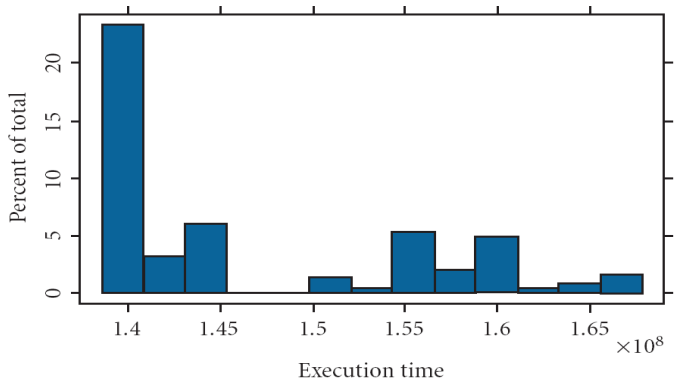


FIG. 4.35 – Espace exploré : Histogramme temps d'exécution

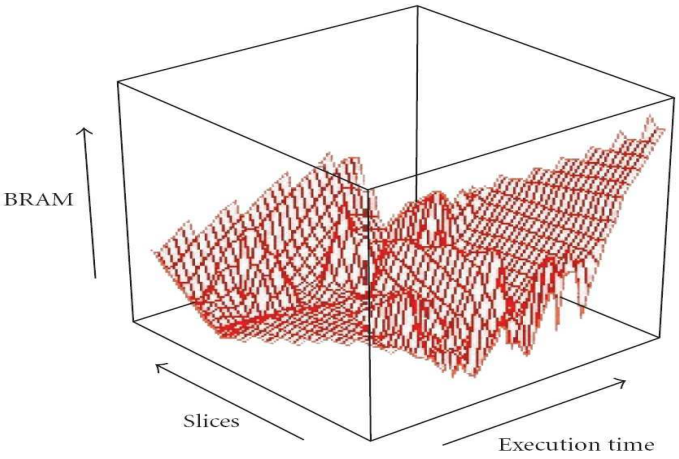


FIG. 4.36 – Espace exploré

à large échelle d'intégration. Contrairement aux multiprocesseurs traditionnels, les multiprocesseurs sur puce impliquent en plus de leur architecture, des contraintes de placement routage pouvant affecter les performances du système. Effectivement, le fait de travailler dans un espace confiné précieux (espace sur puce) donne de l'importance au placement des modules et au routage des communications. Il est par conséquent impossible d'entreprendre une évaluation de tels systèmes sans prise en compte de l'aspect placement routage. Cette évaluation peut être effectuée selon deux procédés différents : (1) soit à l'aide d'une simulation post placement routage, (2) en émulation avec une exécution directe. On peut facilement estimer qu'on aura un gain de temps incomparable en passant par l'émulation que par la simulation, mais nous avons voulu quantifier cela pour mieux conforter notre idée et mieux estimer les performances des nouveaux outils de simulations. Pour cela nous avons simulé notre plateforme multiprocesseur avec l'outil Modelsim6.0a [118] qui est une référence dans le domaine. Modelsim englobe la simulation de différents langages

de description matérielle tels que VHDL, Verilog et SystemC. Il peut exécuter des simulations en post synthèse et post placement routage en lui spécifiant les délais du circuit utilisé. Le tableau ci-dessous (Timings) illustre l'important gain de temps réalisé en utilisant l'émulation au lieu de la simulation sur ce type de plateformes (Plateformes multiprocesseurs).

Timings

Algorithme évolutionnaire multiobjectif (ms)	Indi. Gene.	190
	Obj Functions Eval.	293
	Selection	0.116
	Crossover	0.033
	Mutation	1.118
Synthèse (sec)	Synth.	523.503
	P and R	655.174
	P/R & Bitgen	797.856
Evaluation	Exploration 60 × 30	
	Sim.64x64	2250 jours
	Exec. Directe 256x256	1.39 heures

TAB. 4.7 – Timings

Il faudrait en passant par la simulation 2250 jours sur un ordinateur doté d'un Pentium 4 et de 4 gigas de mémoire centrale, contre 1,39 heures en passant par une exécution directe sur FPGA. Afin d'atteindre les mêmes performances en simulation qu'en émulation il faudrait un ensemble de plus de 25000 ordinateurs. Même avec cela, dans le cas de la simulation, on n'aurait qu'une estimation de l'impact du placement routage sur la performance contrairement à l'émulation. Le flot proposé ici est très compétitif par rapport aux approches SystemC [119, 120]. Des observations similaires ont été tirées dans des études de conception de processeurs embarqués [121, 122].

4.10 Conclusion

Les tendances d'implémentation des systèmes sur puce sont dictées par les avancées technologiques réalisées dans le domaine de la physique du solide (Ce qui est réalisable avec un coût acceptable) et aussi par la nature des applications qu'on découvre ou qui s'améliorent au cours du temps. L'idée phare actuelle est d'implémenter des systèmes multiprocesseurs sur puce. Effectivement, le parallélisme inhérent de la plupart des applications actuelles et les larges capacités des circuits simplifient et encouragent ce genre d'implémentation. C'est pourquoi, nous nous sommes intéressés à l'implémentation d'architectures multiprocesseurs sur puce. Cependant, la possibilité de cette implémentation ne garantit pas une exploitation efficace de ses

larges potentialités. Ceci explique le fait que nous ayons considéré dans ce chapitre une méthodologie permettant d'implémenter de manière efficace ce genre d'architecture, une efficacité exprimant le fait d'avoir introduit la notion d'optimalité au sens de Pareto. Nous avons, pour cela, proposé l'outil MOCDEX permettant de réquilibrer automatiquement l'utilisation de la ressource mémoire, qui est à budget constant au sein de la puce, afin d'améliorer la performance et l'utilisation des ressources. Ce chapitre dessine une relation directe avec son précédent, où nous avons démontré les potentialités en qualité de résultats et en gain de temps de conception que procure cette méthodologie. Ainsi, son adaptation à l'architecture multiprocesseur sur puce, où la taille du circuit implique un nombre de paramètres et des délais plus importants, a montré sa totale efficacité en répondant clairement à nos attentes. Ce que nous proposons peut évidemment être amélioré. Nous avons juste démontré les améliorations en conception que nous pouvons atteindre. Cette méthodologie peut très bien être utilisée dans la recherche du meilleur schéma de communication et des meilleurs éléments de calcul pour une application donnée. Ceci en utilisant des bibliothèques de modules de communication et d'éléments de calcul (Différentes IPs) où l'architecture finale sera un système multiprocesseur hétérogène. Ce qui d'ailleurs, offrira une marge de manoeuvre plus importante permettant d'adapter encore mieux l'architecture cible à l'application de départ. La consommation d'énergie constitue un point important à introduire dans cette proposition de méthodologie afin la compléter (Considérée comme perspective). Aussi, la détermination au préalable (avant le lancement de l'exploration) d'un intervalle d'exploration plus précis constitue une amélioration significative des résultats. Ceci facilitera la tâche d'exploration qui aura comme conséquence directe un gain en temps d'exploration donc en cycle de conception. Une étude statique ou l'observabilité de l'exécution d'un système pourraient nous aider à mieux définir les bornes de l'espace d'exploration. Aussi, la montée en abstraction pour une meilleure spécification de l'architecture par rapport à la nature de l'application à implémenter permet de tirer le maximum de l'exploration. Ces points seront exposés et étudiés dans le chapitre qui suit.

Chapitre 5

Synthèse de NoC

La méthodologie de conception de systèmes multiprocesseurs proposée dans le chapitre précédent consiste à exécuter une exploration multi-objectif (ressources, performance) à base d'algorithmes évolutionnaires sur l'architecture à implémenter. Cette méthodologie s'est avérée très efficace car elle procure dans des temps de conception très satisfaisants (vu que l'évaluation des configurations est basée sur une exécution sur puces programmables) un ensemble de solutions optimales au sens de Pareto. L'exploration a permis d'améliorer la performance et de rééquilibrer l'utilisation des ressources mémoires entre les mémoires caches et les FIFOs de communication. Cependant, cette méthodologie peut gagner en efficacité si l'exploration architecturale est associée à l'exploration logicielle afin de pouvoir réellement adapter l'application à l'architecture et obtenir ainsi une synthèse de systèmes multiprocesseurs sur puce. Ce point sera étudié dans ce chapitre où nous monterons en abstraction afin de pouvoir étudier la possibilité d'une exploration logicielle de la communication pour les systèmes multiprocesseurs sur puce. Le langage de programmation parallèle Occam sera présenté dans les sections suivantes.

La taille des puces programmables (FPGA) actuelles permet l'implémentation sur une même puce de plusieurs IPs logicielles de processeurs. Nous avons, dans ce chapitre, implémenté une réelle architecture de réseau sur puce basée sur des routeurs matériels. De plus, nous proposons une architecture de monitoring non intrusive offrant la possibilité d'avoir, en temps réel, le comportement précis de la communication de l'application implémentée. Par la suite, nous étudierons la possibilité d'injecter les résultats liés au monitoring comme entrée au programme parallèle Occam afin d'améliorer l'implémentation de l'application par un meilleur découpage du parallélisme.

5.1 Travaux effectués dans la synthèse de NoC

L'apparition récente des systèmes multiprocesseurs sur puce comme étant des candidats potentiels forts pour adresser des contraintes d'exécution, d'énergie et

d'espace pour les applications embarquées, nous pousse à se poser la question suivante : comment concevons-nous des multiprocesseurs sur puce efficaces pour une application donnée ? Les outils de conception automatisés n'adressent pas cette question et les techniques parallèles traditionnelles d'architectures d'ordinateur n'ont été exposées ni à la diversité énorme apportée par les méthodologies de conception basées sur la réutilisation d'IP (conception modulaire), ni aux contraintes agressives des systèmes embarqués. Ce problème conduit à diverses propositions [123] essayant de rendre possible la synthèse des systèmes multiprocesseurs sur puce à partir des spécifications abstraites de l'application à implémenter [91, 124, 125]. Dans [126], les auteurs présentent un flot de conception capable de générer une architecture multiprocesseur spécifique à l'application cible. Dans ce flot, les paramètres architecturaux sont d'abord extraits à partir des spécifications à haut niveau puis ils sont utilisés pour instancier des composants architecturaux tels que des processeurs, des mémoires et des réseaux de communication. Dans leur proposition ils ne se basent pas sur une spécification parallèle de leur problème. Dans [127], les auteurs proposent une méthodologie où une architecture est implémentée sur une plateforme multiprocesseur à base de processeurs extensibles. Ils utilisent un algorithme itératif et des outils du marché afin d'adapter leur plateforme au comportement de l'application cible. Cependant, le flot architectural et celui de l'implémentation sont découplés. Dans [128], les auteurs proposent un flot d'exploration automatisé pour les systèmes multiprocesseurs sur FPGA. Ils utilisent comme entrée du système le graphe d'application décrivant les tâches et les liens de communication. Le résultat de ce flot est une configuration des différents processeurs, du médium de communication et un mapping des tâches de l'application sur la microarchitecture. Dans [91], les auteurs présentent une méthodologie automatique de synthèse de réseaux sur puce (NoC) en se basant sur une bibliothèque de composants réseaux. L'entrée de leur flot de conception est une spécification haut niveau de l'application à implémenter. Le but est de générer une architecture réseaux adaptée aux contraintes d'implémentation de l'application et de la conception. La description de leur application se résume en un graphe de tâches et de liens de communication. Ce graphe est obtenu à l'aide d'un outil de codesign (matériel/logiciel) après avoir spécifié manuellement le comportement sous ANSI-C. Une fois ce graphe obtenu, on passe à un mapping de l'application sur un nombre déterminé de topologies. Le premier problème avec cette méthodologie réside dans le fait que le schéma de communication idéal ne représente pas inévitablement une topologie standard du style Mesh, tore, Fat tree ...etc. Le deuxième problème provient du fait que les auteurs font une totale abstraction de l'exécution physique sur leur architecture car les vérifications sont faites au niveau RTL SystemC. Ce que nous proposons dans ce chapitre est une méthodologie permettant de synthétiser le schéma de communication d'une quelconque application parallèle, puis de permettre une vérification de l'architecture directement par exécution [129]. Pour ce travail, nous avons voulu utiliser un langage de programmation parallèle. Occam a été choisi comme outil pour notre méthodologie. OCCAM est

un langage de programmation prouvable. Cette caractéristique permet de réduire le travail de vérification lors de la conception.

5.2 Synthèse de NoC à base de langage parallèle : OCCAM

5.2.1 Le langage de programmation OCCAM

Occam est un langage de programmation parallèle basé sur le paradigme de communication séquentielle des processus [130] CSP. CSP est une théorie mathématique permettant de spécifier et de vérifier les interactions entre modules concurrents. Développé par T. Hoare, CSP a une sémantique qui simplifie considérablement la conception systèmes parallèles. L'objectif principal du langage Occam, qui s'inspire directement de CSP, est d'obtenir un code de programmation parallèle simple et facile à comprendre. La Figure 5.1, illustre la structure d'un programme Occam avec ses processus et canaux de communication. Les programmes Occam se composent de processus concurrents où la communication est réalisée de manière synchrone en passage de messages. Les programmes Occam peuvent fonctionner sur un vrai système multiprocesseur ou sur un processeur simple où le parallélisme est simulé à l'aide de threads.

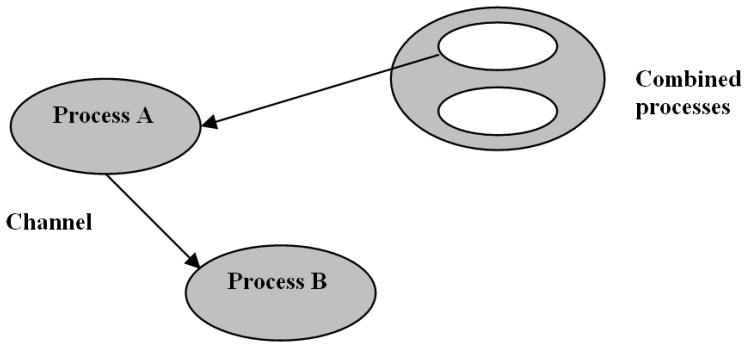


FIG. 5.1 – La structure d'Occam

La simultanéité et la communication sont les concepts principaux d'Occam [131]. La sémantique d'Occam simplifie la tâche lors de la vérification de programmes. Effectivement, Occam est un langage prouvable mathématiquement garantissant une communication sans congestion, deadlock,...etc. La syntaxe concise et formelle du langage Occam en fait un bon candidat pour la spécification et la conception d'applications systèmes embarqués.

Les programmes Occam sont établis en combinant des processus simples entre eux. Des constructions sont établies en combinant des processus primitifs où chaque construction est présentée par un mot-clé.

Une construction est elle-même un processus, et peut être employée comme composant d'une autre construction. Des programmes séquentiels conventionnels peuvent être exprimés avec des variables et des tâches, combinées en constructions séquentielles. Des constructions telles que IF et WHILE sont également fournies. Des programmes concurrents peuvent être exprimés avec des canaux et des entrées/sorties. Chaque canal Occam fournit un chemin de communication entre deux processus concurrents. La communication est synchronisée et n'aura lieu que quand les processus d'entrée et de sortie sont prêts. Les données à produire sont alors copiées du processus producteur au processus consommateur. Le choix de communication synchronisée empêche la perte des données.

Mis à part le fait que le langage Occam permette d'exprimer plus facilement des fonctionnements parallèles et qu'il soit, par ailleurs, prouvable mathématiquement, nous cherchons dans notre cas, et donc, dans le cas de la conception de systèmes multiprocesseurs sur puce, un langage parallèle permettant les transformations tout en assurant leur intégrité. Le fait d'avoir la possibilité de changer facilement d'architecture parallèle, soit dans le nombre de processeurs utilisés ou bien dans la manière dont ces derniers sont connectés, permet d'entreprendre une exploration au niveau logiciel afin d'approcher la meilleure architecture pour l'application à implémenter. Il se trouve qu'Occam ouvre la possibilité à faire ceci, et les premiers transputers devaient être capables d'assumer cette flexibilité. Effectivement, le principal objectif de conception avec Occam et les transputers était de fournir les mêmes techniques de programmation concurrentes pour un transputer simple que pour un réseau de transputers. Ces transputers permettent un haut degré de concurrence grâce à un modèle de programmation décentralisé et une communication entre processus avec passage de messages. En faisant l'analogie avec un processeur utilisé dans notre système multiprocesseur en particulier ou un quelconque autre système multiprocesseur en général, les transputers utilisés avant avec Occam représentaient pratiquement la même architecture. A savoir, ces derniers sont dotés de mémoires, de processeurs et d'un nombre de communications point-à-point afin de se connecter à d'autres éléments de calcul.

La facilité de transformation avec Occam permet de compiler et d'exécuter un programme prévu pour un système multiprocesseur sur un système monoprocesseur, ou bien tout simplement sur un ordinateur consacré à la conception. Une fois le comportement logique fixé, on peut entreprendre l'implémentation de l'application parallèle sur un réseau multiprocesseur sur puce de degré n allant de 1 à N (N étant le nombre maximum de processeurs pouvant être implémentés en même temps sur la puce). La décision se fera par rapport aux besoins directs du concepteur. Il sera question de faire un compromis entre ressources et performance, et le degré de liberté introduit par Occam aidera à le faire de manière efficace car introduit assez tôt dans le flot de conception.

Occam est constitué principalement de six types de processus élémentaires que

l'on peut associer au moyen de différents constructeurs afin de former des programmes.

5.2.1.1 Processus élémentaires

Afin que Occam puisse garder un aspect de programmation simple, le nombre de processus élémentaires a été intentionnellement limité à six. Ces derniers peuvent être par la suite associés afin de réaliser des processus plus complexes.

- Affectation (:=)
  variable := expression
  Ce processus d'affectation transfert la valeur d'une expression après évaluation à une variable locale du processus.
- Emission d'un message (!)
  canal ! expression
  Après avoir évalué correctement l'expression, une sortie reste en attente de transmission jusqu'à ce qu'une entrée utilisant le même canal soit prête à recevoir.
- Réception d'un message (?)
  canal ? variable
  Une entrée reste en attente jusqu'à ce qu'une sortie utilisant le même canal ait envoyé la donnée. A ce moment la donnée reçue est affectée directement à une variable locale du processus receveur.
- Processus d'attente (AFTER)
  TIME ? AFTER expression
  Ce processus commence par évaluer l'expression puis termine une fois que la valeur lue sur le canal TIME est supérieure à l'expression
- Processus d'arrêt (STOP)
  SEQ keyboard ? char
  STOP
  screen ! char
  Dans cet exemple, on lit la donnée à partir du canal " keyboard " puis on exécute le processus " STOP " qui va commencer et ne jamais se terminer, ce qui empêchera le déroulement de la séquence jusqu'à la fin (l'Affichage sur l'écran ne se fera pas). Ce processus permet essentiellement d'avoir une mauvaise terminaison des programmes, sachant qu'une bonne terminaison implique l'arrêt de tous les processus actifs sur la fin de leur code.
- Processus nul (SKIP)
  SEQ keyboard ? char
  SKIP
  screen ! char
  Ce processus se lance, n'exécute aucune action et se termine. Il permet de modéliser une bonne terminaison et de décrire des comportements vides.

5.2.1.2 Les constructeurs dans Occam

On évoquera dans cette partie quatre principaux constructeurs dans Occam SEQ, PAR, ALT et enfin WHILE. Ces derniers sont utilisés afin de construire de nouveaux processus.

- Le constructeur séquentiel SEQ
  SEQ
  Proc1
  Proc2
  .
  .
  Le processus séquentiel assure le fait que les processus composants s'exécutent l'un après l'autre. La construction termine son exécution lorsque le dernier processus termine son exécution. Dans le cas où il n'y a pas de composant, le processus séquentiel se comporte de la même manière qu'un SKIP.
- Le constructeur Parallèle PAR
  PAR
  Proc1
  Proc2
  .
  .
  Ce constructeur impose à ses composants de s'exécuter de manière parallèle. Tous les processus composants commencent leur exécution en même temps. Le processus parallèle termine son exécution lorsque le dernier composant parallèle a achevé sa tâche. Vide, le processus parallèle est équivalent au processus SKIP.
- Le constructeur alternatif ALT
  ALT
  Proc1
  Proc2
  .
  .
  Occam a la possibilité d'exécuter de manière non-déterministe un ensemble de processus. Ceci est rendu possible grâce au constructeur ALT qui permet de décrire un ensemble de processus pouvant être exécutés de manière asynchrone. Le processus ALT se met en attente jusqu'à ce qu'un des processus composants soit prêt à être exécuté. Le processus ALT se comporte comme un processus STOP dans le cas où il n'a pas de composants. On retrouve dans le langage Occam l'implémentation de la commande gardée proposée par W. Dijsktra. La garde représente une expression booléenne directement associée à un processus. Une fois que le processus reçoit son entrée, la garde est calculée. Le système ne pourra lancer l'exécution du processus que si l'expression

booléenne est vraie. Cette façon de faire donne la possibilité au programmeur de contrôler de l'extérieur le lancement des processus (dans le cas où le calcul de la garde est fait par le système et non pas par le processus).
ALT
Garde1
Proc1
Garde2
Proc2
.
.

– Le constructeur répétitif WHILE
WHILE expression
Proc
Ce processus répète l'exécution du processus composant tant que l'évaluation de l'expression retourne une valeur vraie.

5.2.2 Le flot de synthèse à base d'OCCAM

Le travail présenté ici a fait l'objet d'une publication. MOCSOC pour "Multiprocessor on Chip Synthesis from OCCAM" [132].

5.2.2.1 Interprétation du code Occam en C

La première étape explorée consiste à définir la manière dont s'effectue le portage du code parallèle Occam sur une architecture multiprocesseur sur puce. Nous proposons ici une traduction du code Occam vers un code C fonctionnellement équivalent où les routines de communication sont ramenées à des appels de primitives MPI. Les appels aux routines MPI sont traduits par des liens de communication physique entre les différents processeurs de l'architecture MPSoC. Nous avons, pour cela, développé un analyseur lexical et syntaxique du code OCCAM. Les outils Lex & Yacc (Lex : A Lexical Analyzer Generator, Yacc : Yet Another Compiler-Compiler) [133] ont été utilisés. Lex est un analyseur lexical permettant de retrouver des expressions régulières dans un texte et exécuter pour chacune des expressions retrouvées une commande décrite préalablement en C. Lex permet de fournir une suite de "tokens" (les éléments terminaux du langage) à partir d'une suite de caractères fournis en entrée. Yacc permet de saisir une grammaire complexe d'un langage et de l'analyser : c'est un analyseur syntaxique (Parser) utilisé en couple avec Lex. A l'aide de la fonction yylex(), Yacc lit les tokens renvoyés par Lex. Concernant notre code Occam, un arbre syntaxique est construit lors de l'analyse de ce dernier puis traduit en code ANSI-C à chaque fois que des constructions telles que les affectations, les boucles, les conditions...etc sont reconnues.

La figure 5.2 illustre le flot de traduction du code parallèle Occam en code ANSI-C.

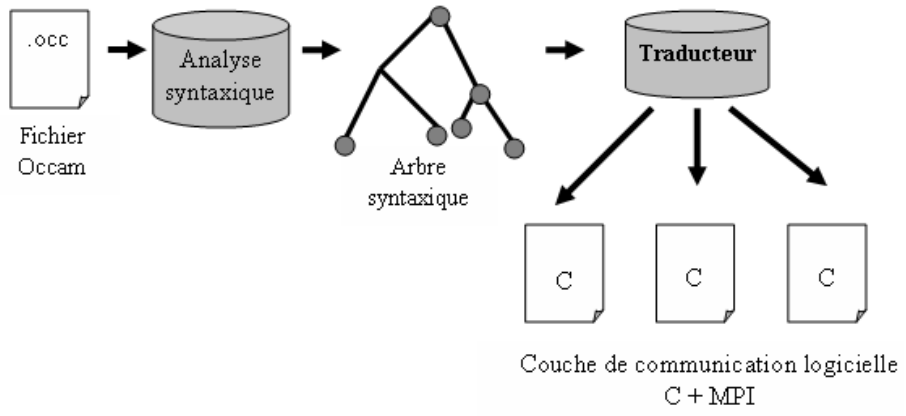


FIG. 5.2 – Flot de génération de code C pour multiprocesseur à Partir d'Occam

Les processus parallèles sont détectés à chaque fois qu'une commande PAR est reconnue dans le code source Occam. Ces processus parallèles sont par la suite directement traduits en C et affectés à un processeur particulier. Nous projetons comme future amélioration, la possibilité de pouvoir mettre les processus concurrents soit sur un processeur à part ou bien en tant que threads, se disputant, ainsi, une même unité de calcul. Cela nous a été difficile à réaliser du fait du manque de ressources en mémoire sur le composant cible (FPGA Virtex-II 8000) sur lequel nous avons implémenté notre application. Le manque de ressources rendait impossible d'implémenter un OS prenant en compte les exécutions de threads. Nous tenons à signaler que ceci est parfaitement réalisable et ce que nous présentons ici ne fait qu'illustrer la possibilité de le faire. Nous utilisons ainsi dans notre cas d'étude un seul niveau de parallélisme (PLACED PAR). Nous donnons dans ce qui va suivre quelques exemples de traduction de code Occam en C. La traduction peut être triviale comme dans le cas d'une construction conditionnelle (5.3).

```
IF
  a < b
    min := a
  a >= b
    min := b
```

```
if (a < b)
{
  min = a;
}
if (a >= b)
{
min = b;
}
```

FIG. 5.3 – Traduction OCCAM-C d'une construction conditionnelle

Effectivement, la construction conditionnelle en Occam est pratiquement équivalente en code C. Pour ce genre de constructions, il suffit juste de réadapter les

caractères afin de les ramener à une syntaxe C.

Le second exemple (Figure 5.4) est plus compliqué car il fait appel à des processus de communication (envoi et réception). Ces processus de communication côté Occam sont directement associés à des appels de routines de communication MPI MPI_Send() et MPI_Recv(). Le traducteur commence dans ce cas par déterminer la taille de la variable Buffer qu'il appliquera à l'argument Count des deux fonctions envoi/réception de MPI.

```
CHAN in, out:
[2]INT buffer:
WHILE buffer <> eof
  SEQ
    in ? buffer
    out ! buffer
```

```
while (buffer!=eof)
{
MPI_Recv(&buffer,2,MPI_INT,MPI_ARG_1,MPI_COMM_WORLD,NULL);
MPI_Send(&buffer,2,MPI_INT,MPI_ARG_2,MPI_COMM_WORLD);
}
```

FIG. 5.4 – Traduction OCCAM-C Communication Entrée/Sortie

Concernant les arguments MPI_ARG_1 et MPI_ARG_2 correspondant à la source et destination, ils sont déduits en fin d'analyse, cela en examinant pour chaque canal de communication déclaré les processeurs émetteurs et récepteurs.

Le troisième et dernier exemple (Figure 5.5), illustre comment des processus parallèles de l'application sont assignés à différents processeurs de notre plateforme système multiprocesseur sur puce. La construction PLACED PAR est utilisée pour spécifier le fait qu'un processeur doit s'exécuter en parallèle.

```
PLACED PAR
  PROCESSOR 1
    SEQ
      in?val
      val1:=val1+1
      out!val1
PLACED PAR
  PROCESSOR 2
    SEQ
      in!val2
      out?val2
```

```
//processus1
void main()
{
  MPI_Recv(&val1,…);
  val1:=val1+1;
  MPI_Send(&val1,…);
}
```

```
//processus2
void main()
{
  MPI_Send(&val2,…);
  MPI_Recv(&val2,…);
}
```

FIG. 5.5 – Traduction OCCAM-C PLACED PAR

Quand le traducteur détecte ceci, il génère un fichier code C qui sera dédié à un processeur particulier.

En plus du fait que le traducteur fournisse une parfaite traduction du code Occam parallèle en code C plus routines MPI, il nous permet aussi d'avoir un graphe récapitulant l'ensemble des processus de l'application s'exécutant en parallèle avec leurs différents canaux de communication associés. Ceci permet de détecter visuellement des états d'interblocage dans le cas, par exemple, où il y un processeur en attente de lecture alors qu'aucun ne lui transmet de données. Sachant qu'Occam fait des accès lecture et écriture bloquants, cette situation constitue réellement un cas typique de

blocage. Ce même graphe d'interconnexion nous permettra par la suite de déduire le placement (Figure 5.6) de l'application parallèle sur une plateforme système sur puce reprogrammable. Aussi nous aurons besoin, pour bien paramétrer les canaux de communication et pour un bon rééquilibrage de charges, d'une connaissance du taux de trafic circulant sur chacun des buffers de communication. Nous proposons dans ce chapitre une technique évitant la simulation procurant ainsi des résultats rapides et réels. Elle sera illustrée et discutée dans les paragraphes qui vont suivre.

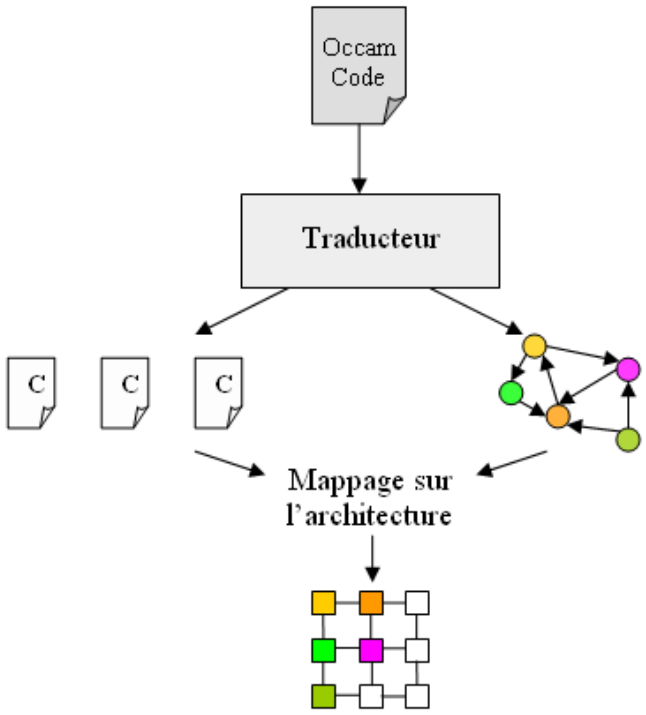


FIG. 5.6 – Mapping des processus Occam sur une plateforme MPSoPC

La figure 5.7 illustre toutes les étapes du flot de synthèse proposé. Le traducteur prend en entrée la spécification parallèle décrite en Occam et produit en sortie : un code (Ansi-C + routines MPI) pour chaque processeur de la plateforme MPSoC, les graphe de communication et de manière optionnelle le code pour simulation (Les routine MPI décrites dans le chapitre 4 sont totalement portables).

5.2.2.2 Cas d'étude sur une application de réseau de neurones

Avant d'aborder une synthèse d'architectures parallèles sur puce, nous avons d'abord testé le bon fonctionnement du traducteur Occam vers C en vérifiant le bon déroulement de toutes les étapes. Le but ici est donc de prendre une application parallèle suffisamment significative, la décrire en Occam et la traduire automatiquement en C plus les routines nécessaires en MPI pour la communication. L'application

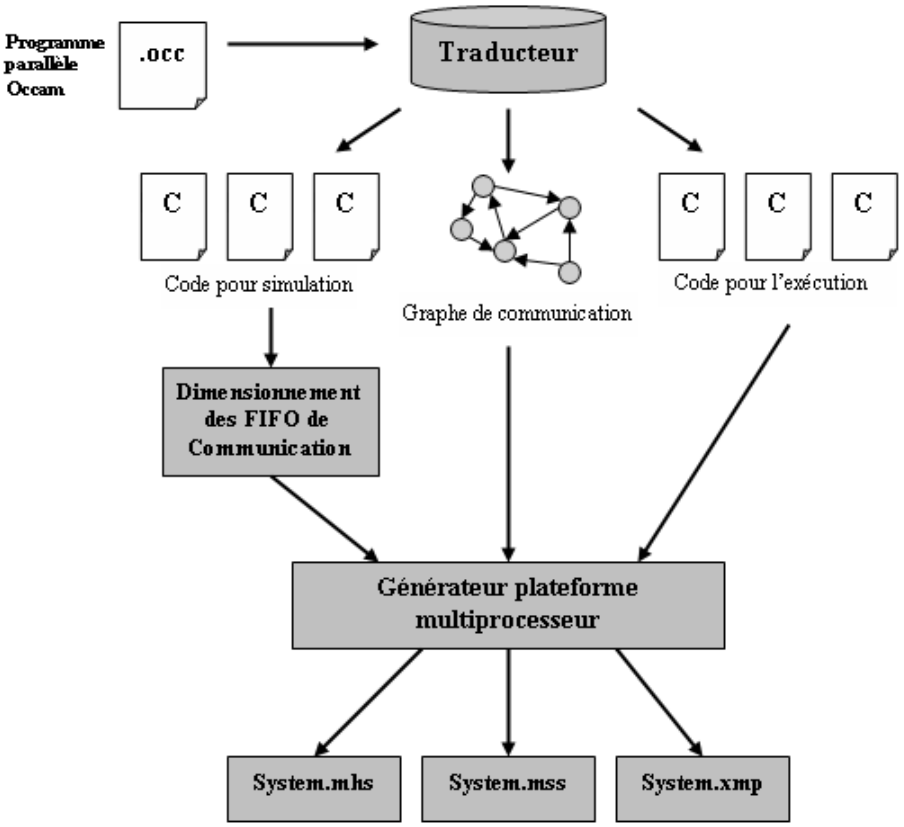


FIG. 5.7 – Flot de synthèse de systèmes multiprocesseurs sur puce

choisie est le SOM (Self Organizing Map) qui est une carte auto-organisatrice. Le SOM est un réseau de neurones basé sur un apprentissage non supervisé. Cet algorithme tente de cartographier les classes de données en entrée et cela en réduisant leur espace multidimensionnel d'origine en une représentation en deux dimensions. Cette représentation bidimensionnelle permet une interface de visualisation simplifiée où l'organisation des données se fait grâce à leur degré de similarité. Plus des données sont similaires, plus elles sont rapprochées dans leur représentation dans l'espace 2 D et vice versa. Chaque neurone du réseau doit à la fin représenter un sous-ensemble de données de l'entrée.

Nous avons choisi d'implémenter une version simplifiée des cartes auto-organistarices. Deux fonctions sont nécessaires pour l'exécution de l'algorithme : le calcul de la distance entre vecteurs Dist, et la fonction d'apprentissage appliquée aux neurones. Les étapes de l'algorithme sont comme suit :

- Initialiser aléatoirement la valeur des poids des différents neurones de la grille
- Répéter la phase d'apprentissage jusqu'à l'obtention d'un résultat satisfaisant :
  - Présenter la même entrée " val " à tous les neurones de la grille
  - Chaque neurone i calcule la distance entre son poids et la valeur d'entrée

Dist(Poids(i),val)
- Le neurone gagnant est le neurone dont la distance est minimale.
- Modifier le poids du neurone gagnant avec la fonction App :(A)
- Modifier le poids des voisins du neurone gagnant avec la fonction App :(B)

La fonction Dist() calcule la distance Euclidienne entre le poids du neurone et la valeur d'entrée :

$$Dist\left(Poids, Val\right) = \sqrt{\sum_{dim}^{j=1} \left(Poids\left[j\right] - Val\left[j\right]\right)^2}$$

La fonction App() met à jour la valeur des poids des neurones en fonction du nombre d'itérations. Celle-ci est appliquée avec différentes valeurs selon le type de neurones à évaluer (neurones gagnants ou neurones voisins) :

Pour le neurone gagnant :

$$Poids\left[j\right]\left(t+1\right) = Poids\left[j\right]\left(t+1\right) + \alpha\left(t\right) \cdot \left\{val\left[j\right]\left(t\right) - Poids\left[j\right]\left(t\right)\right\}$$

Pour les neurones voisins :

$$Poids\left[j\right]\left(t+1\right) = Poids\left[j\right]\left(t+1\right) + \beta\left(t\right) \cdot \left\{val\left[j\right]\left(t\right) - Poids\left[j\right]\left(t\right)\right\}$$

L'algorithme parallèle des cartes auto organisatrices implémenté répond à une architecture maître-esclave telle que représentée par la Figure 5.8. L'exécution de l'algorithme se fait en deux étapes. Durant la première, le processus maître envoie à tous ses esclaves un vecteur d'entrée.

Une fois ce vecteur reçu par les processus esclaves, ces derniers se chargeront de calculer la distance entre ce dernier et leur vecteur associé (Initialement fixé de manière aléatoire) puis enverront le résultat au processus maître. Le processus maître désigne alors le neurone gagnant.

Cette expérience a été réalisée sur une machine hôte. Les résultats sont représentés de manière graphique par la figure 5.10. On remarquera le bon fonctionnement de cette implémentation à travers le fait que l'algorithme a pu converger et classer les couleurs en entrées selon leur similarité.

5.2.2.3 Application de la synthèse

Nous avons dans les sections précédentes mis en exergue les avantages du langage OCCAM comme étant un candidat potentiel pour la synthèse multiprocesseur sur puce. Nous avons démontré aussi la faisabilité de la traduction du code OCCAM en code ANSI-C plus routines MPI. Ce que nous présentons maintenant dans cette section est une application réelle de cette synthèse sur une puce reprogrammble du type FPGA. L'application implémentée est la même que celle utilisée lors de la démonstration de la faisabilité de la transformation OCCAM-C en l'occurrence le

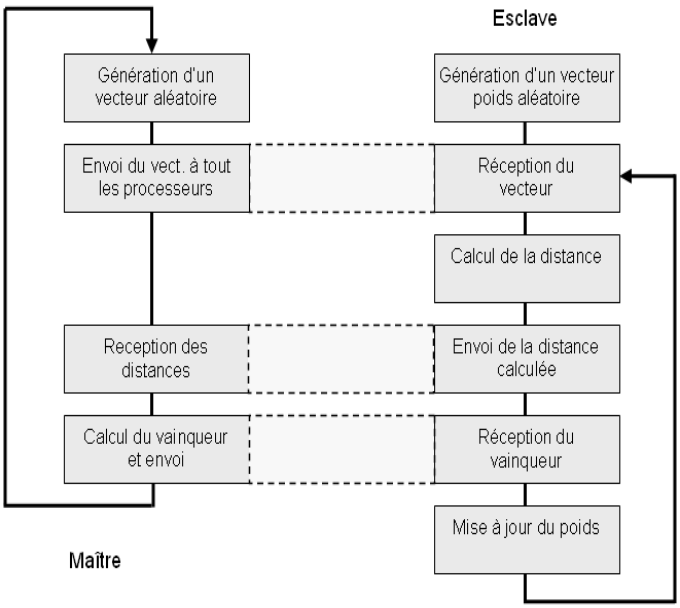


FIG. 5.8 – Algorithme parallèle du SOM

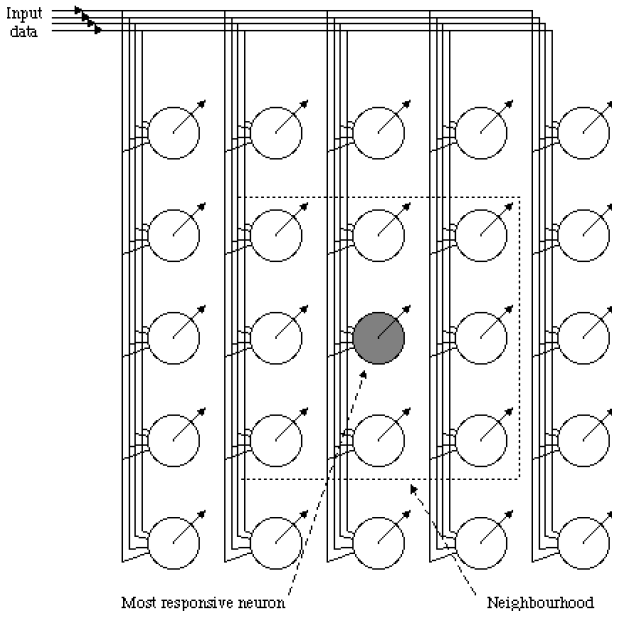


FIG. 5.9 – Graphe de communication d'Occam

réseau de neurone des cartes auto organisatrices (SOM). Notre démarche de synthèse s'appuie sur une bibliothèque de composants matériels.

Nous avons appliqué notre synthèse en se basant sur les outils et les composants matériels que fournit Xilinx sachant que cela peut être naturellement généralisé à d'autres outils. Il est nécessaire de signaler le niveau de granularité auquel nous

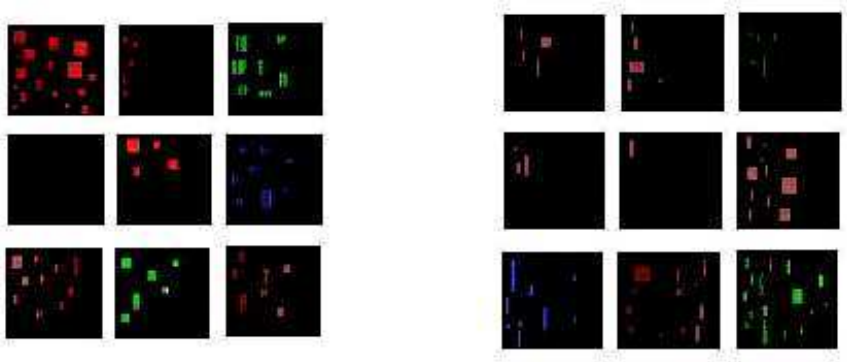


FIG. 5.10 – Résultats de l'implémentation du SOM

travaillons. Ce que nous cherchons dans un premier temps est de démontrer la faisabilité de la synthèse telle que présentée. Nous travaillerons sur l'instanciation de composants de base pour la réalisation d'un système multiprocesseur (processeurs et médias de communication) sans se soucier de la dimension technique relative à chaque composant. Ce qui doit être assuré dans un premier temps (ce qui sera présenté ici) est le passage d'une spécification OCCAM à une implémentation réelle sur FPGA.

Le graphe de processus obtenu avec OCCAM est représenté par la figure 5.11. C'est une topologie en étoile avec le neurone maître au centre. Nous avons utilisé pour son implémentation les mêmes éléments utilisés dans le chapitre 4, processeurs Microblaze, liens de communication FSL et la couche logicielle de communication MPI. Nous avons aussi repris le même script pour générer de façon automatique l'architecture multiprocesseur à partir des outils Xilinx.

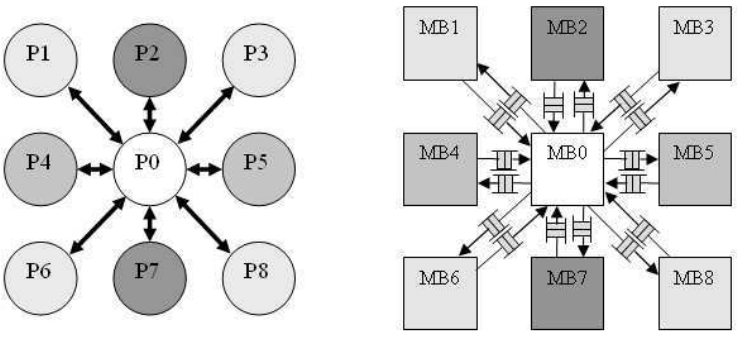


FIG. 5.11 – Architecture

La figure 5.12 illustre l'architecture obtenue avec l'outil XPS de Xilinx après implémentation. Celle-ci a été par la suite implémentée sur un FPGA Xilinx (Virtex-II 8000).

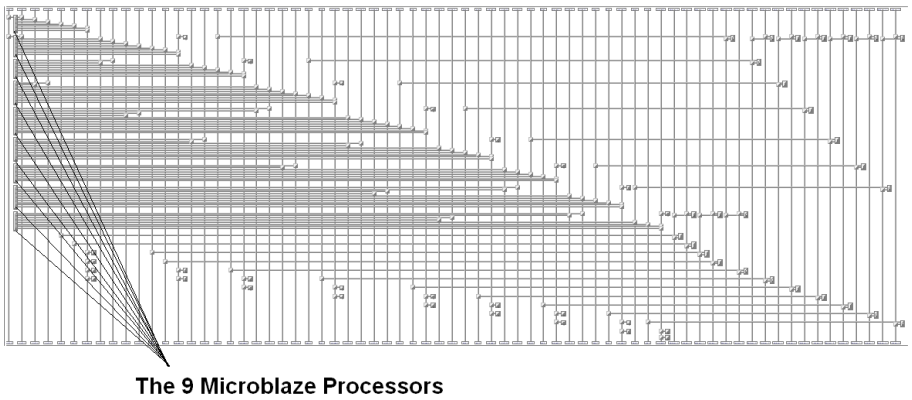


FIG. 5.12 – Architecture SOM générée avec Xilinx XPS

L'élément de base pour cette architecture est représenté par un processeur Microblaze doté d'un bus OPB et d'un timer pour l'exécution des routines de routage MPI.

A travers ce que nous avons présenté, nous avons démontré la possibilité de pouvoir synthétiser à partir du langage OCCAM une application parallèle jusqu'à son implémentation réelle sur une puce reprogrammble. Reste que cette synthèse peut être plus efficace s'il existe un moyen de connaître comment paramétrer rapidement les différentes IPs de communication. Ce que nous avons proposé dans le chapitre précédent permet de réaliser une exploration évolutionnaire multi-objectif. Seulement, afin de rendre plus efficace cette exploration, il est très intéressant de pouvoir définir un intervalle réel où l'espace de recherche sera réduit afin de gagner en efficacité lors de la phase d'exploration. Comme exprimé dans le chapitre 4, la taille des architectures actuelles rendent impossible la simulation RTL d'un système monoprocesseur encore moins d'un système multiprocesseur dans les temps qui nous intéressent. On peut tout de même monter en abstraction et simuler à un niveau transactionnel (SystemC TLM), mais cela se fait bien sûr en payant le coup de la précision des timings et de l'estimation des ressources utilisées. Nous avons pensé que la taille des puces et le nombre de ressources disponibles nous permettrons d'enfouir sur puce (en plus de notre système multiprocesseur) des éléments susceptibles (Mouchards) de nous remonter assez rapidement (de manière instantanée car en même temps que l'exécution) les informations sur le comportement des IPs instanciées dans notre plateforme. Nous souhaiterions, après avoir vérifié la faisabilité de ce concept de monitoring, pouvoir l'intégrer à notre flot de synthèse à base d'OCCAM afin de synthétiser en même temps que le schéma de communication, les ressources (Processeurs, liens de communication...etc) utilisées dans les systèmes multiprocesseurs. Ce travail est présenté dans les sections suivantes.

5.3 Monitoring des réseaux sur puce

Les réseaux sur puces, comme discutés précédemment, permettent d'avoir un meilleur (Au sens de la modularité et de l'extensibilité de la bande passante) schéma de communication comparés aux bus. Cependant, une bonne exploitation des réseaux nécessite une excellente compréhension de la nature du trafic y existant ainsi que du comportement des différents éléments constituants le réseau (Routeurs, Fifo, liens etc...). Connaître précisément le comportement de ces modules de communication permet une réalisation optimale de l'architecture de communication sur puce. Une fois que nous obtenons le comportement et l'utilisation des composants de communication tels que les routeurs et les liens, nous pouvons décider de la méthode de cheminement la plus appropriée, des profondeurs optimales des buffers de communication ainsi que de la topologie de communication. Ce que nous présentons, dans cette partie du travail, est une évaluation des performances instantanée des réseaux sur puces de n'importe quelle application donnée qui tient compte des vraies contraintes d'exécution et d'implémentation physique. Elle tire profit de la programmabilité simple et facile et de la grande capacité d'intégration des FPGA, nous aide à fournir une méthodologie plus rapide d'évaluation des performances en évitant le problème du temps prohibitif de simulation. Dans cette situation, enfouir des IPs spécifiques reprogrammables et non intrusives dans l'architecture système, semble être une bonne solution pour le monitoring des réseaux sur puces afin de fournir au concepteur des résultats significatifs et instantanés, lui permettant d'améliorer l'exécution de son application.

5.3.1 Travaux réalisés

Nous présentons ici différents travaux de recherche réalisés et proposés dans le but d'investiguer et de comprendre le comportement d'un trafic dans une architecture réseau sur puce. Ceci permettra de positionner notre travail par rapport aux besoins actuels dans le domaine des NoC. La complexité croissante des systèmes sur puces permet l'intégration d'un nombre important d'IPs sur une même puce offrant une puissance de calcul significative mais où le temps de communication constitue un paramètre crucial à optimiser. L'effort de recherche dans l'évaluation des schémas de communication des MPSoCs et plus particulièrement des réseaux sur puces a été largement abordé afin de garantir des exécutions optimales [134, 135, 136]. Dans [137], les auteurs présentent un travail réalisé sur la génération d'un trafic et l'évaluation de performance d'un réseau sur puce basé sur une topologie MESH. Le trafic généré est paramétrable et est sensé reprendre le comportement de n'importe quelle application réseau. Cela a pour but de tester rapidement l'architecture de communication d'une plateforme sous de réelles conditions de trafic. Les mesures qui ont été faites concernaient l'utilisation moyenne de la bande passante entre les liens de communication, le temps moyen pour la transmission d'un flit et l'utilisation

des ressources du réseau. Cette méthode est très utile pour étudier le comportement du réseau sur puce en fournissant des résultats de bande passante et de latence. Cependant, toutes ces différentes métriques ont été extraites à partir de la simulation, et cela consomme en moyenne 35 minutes pour l'évaluation d'un scénario de 1000 paquets, ce qui est insignifiant par rapport à un réel trafic de communication sur puce. Dans [138], les auteurs ont proposé une méthodologie pour comparer des architectures de NOC en termes de latence, dissipation d'énergie, et d'espace occupé sur puce. Cinq topologies de réseaux sur puces ont été comparées : SPIN, CLICHE, torus, l'Octogone et enfin BFT (Butterfly fat-tree). Un MPSoC a été simulé avec différentes topologies de réseaux sur puces. Dans [139], une méthodologie pour simuler des plateformes réseaux sur puces est présentée. A partir de la simulation les auteurs analysent le comportement du réseau en examinant l'influence de la tailles des buffers d'entrées/sorties des différents routeurs et la charge de communication sur l'ensemble du réseau. De ces deux paramètres, des résultats de simulation concernant les taux de pertes et les délais sont fournis. Pour résoudre le problème du temps excessif de simulation, un travail important de recherche sur le monitoring de système a été entrepris. La majeure partie de ce travail a visé de deboggage, le test, et l'évaluation de performances des systèmes complexes en temps réel avec des contraintes agressives de communication. On peut regrouper l'ensemble de ces travaux en trois catégories de monitoring. La première manière est logicielle et est basée sur l'insertion de codes incluant des fonctions spécifiques de monitoring se déclenchant à chaque fois que les conditions de déclenchement d'une phase de monitoring sont remplies. Le monitoring basé sur le matériel est la deuxième alternative. Dans celle-ci, des IPs moniteurs sont incluses au sein de la plateforme système afin de recueillir les informations pendant l'exécution. Cette façon assure une surveillance rapide et non intrusive. Basé sur ceci, dans [140], l'auteur présente un système de monitoring non intrusif basé sur une IP matérielle externe reconfigurable. Les événements concernant la communication sur un bus sont collectés durant l'exécution système. La troisième, et dernière façon d'effectuer un monitoring comprend la combinaison du matériel et du logiciel permettant un monitoring plus flexible. Dans [141], les auteurs ont présenté une approche pour le monitoring de réseaux sur puces basée sur des générateurs de trafic. La plateforme d'émulation a été mise en application sur un FPGA. Des trafics réalistes de communication sur puce sont produits pour explorer le comportement du réseau de routeurs. Des informations générales sur le cheminement de paquets telle que la latence moyenne et les résultats moyens de congestion ont été présentées. Dans ce travail, il y a seulement un générateur de trafic (TG) qui injecte des paquets au réseau considérant le NoC comme étant une boîte noire. Nous pensons que l'émulation devrait être plus réaliste et significative si chaque commutateur sur le réseau obtenait son propre générateur et récepteur de trafic rendant l'application plus flexible. Cela nous permettra de détecter quelle partie dans la conception devrait être modifiée pour améliorer l'exécution du NoC. Dans notre travail, nous avons conçu un cadre de monitoring où un processeur incor-

poré dans l'application et ses moniteurs matériels associés se chargent de collecter des informations d'exécution. L'utilisation d'un processeur nous donne la possibilité de modifier différentes conditions de déclenchement de monitoring à tout moment sans avoir à re-synthétiser ou re-mapper la plateforme. Une contrainte additionnelle dans notre travail consiste à réaliser des moniteurs non intrusifs qui sachent espionner l'application sans altérer son comportement. Ceci permet l'exploration de vraies applications de NoC où le but est de déterminer comment chaque composant de l'architecture de communication se comporte et de définir avec précision l'espace d'exploration dans un temps égal à une exécution réelle.

5.3.2 Cas d'étude du monitoring

5.3.2.1 Description de la plateforme réseau sur puce

Notre plateforme FPGA multiprocesseur se compose de quatre processeurs Microblaze dotés d'unités de cache d'instructions et de données. Ces processeurs communiquent entre eux à travers un réseau d'interconnexion à commutation de paquets. Chaque Microblaze est relié, comme représenté sur la figure 5.13, à un bus OPB. Nous avons connecté sur ce dernier un timer et un contrôleur d'interruptions pour l'exécution des threads. Le Microblaze MB0 est relié au bus OPB qui est lui même relié au bus PCI de la machine hôte (Connexion FPGA machine hôte). Ceci nous permet d'envoyer et de recevoir des données de la machine hôte au système multiprocesseur sur FPGA.

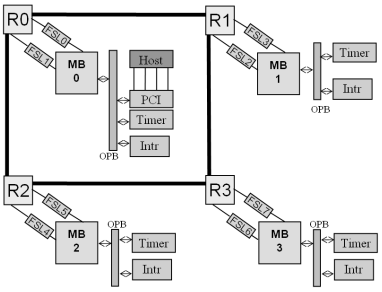


FIG. 5.13 – Plateforme Mesh 2x2

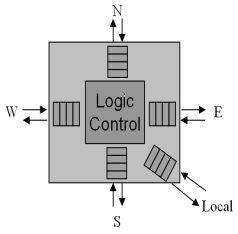


FIG. 5.14 – Routeur

Les processeurs accèdent au port local de leur routeur associé à travers un lien FSL, IP Xilinx déjà expliquée et utilisée dans les architecture présentées précédemment. Le routeur utilisé a été inspiré du projet Hermes [111]. Nous avons repris ce routeur en le modifiant et l'adaptant à notre architecture. Une interface (NI : Network Interface) a été développée afin de le connecter au lien FSL. Ce routeur est composé d'une logique de routage supervisant cinq ports bi-directionnels : Local, Sud, Nord, Est et Ouest. Chacune de ces entrées/sorties sont composées de buffers de profondeur paramétrable permettant de stocker temporairement les paquets.

Chaque port représente un vecteur de données de 8 bits. Sur le port Local, les données sont présentées par paquets multiples de 32. Ceci est principalement dû à la largeur des données FSL. L'interface réseau que nous avons développée consiste à grouper les vecteurs de 8 bits avant de les envoyer au processeur via le lien FSL. Dans l'autre sens (Processeur vers réseau), les vecteurs de 32 bits provenant des FSL sont décomposés en flits de 8 bits chacun avant d'être injectés dans le réseau. Les paquets sont constitués d'une entête comprenant l'adresse de destination et la source puis le nombre de flit de 8 bits envoyés avec ce paquet. les 16 bits restant dans l'entête ont été utilisés pour stocker une valeur de compteur nécessaire pour le monitoring de tâches (time stamping). Pour une utilisation optimale l'adresse source et de destination ont été codées sur 4 bits chacune. Permettant ainsi d'adresser un maximum de 16 processeurs. Il est à noter que cette capacité d'adressage peut facilement être modifiée (augmentée ou réduite). L'algorithme de routage choisi est un algorithme Wormhole déterministe en XY. Le flit de l'entête réserve les Entrées/Sorties d'un routeur afin que le reste du paquet puisse suivre le même cheminement. Le dernier flit du paquet se chargera de libérer les ressources du routeur. Un contrôleur d'interruptions a été utilisé afin de détecter la présence de paquets dans les FSL. Le processeur s'abonne à cette interruption et à chaque fois que celle-ci se déclenche, il exécute une routine afin de lire les données se trouvant dans le canal FSL.

5.3.2.2 Plateforme du monitoring

Le cadre du système de monitoring proposé est illustré dans la figure 5.15. Le but est de permettre au concepteur de recueillir des informations en temps réel sur son application au moyen d'éléments (IPs) intégrés au sein du système. Le problème est de ne pas altérer le comportement de notre application et ce, en utilisant des IPs non intrusives pour le monitoring. Vu la taille des FPGA (Virtex II 8000), nous avons opté pour l'utilisation d'un processeur Microblaze dédié au système de monitoring ce qui facilite considérablement la gestion des données récoltées. Le fait d'utiliser un processeur nous permet aussi de réadapter ou tout simplement modifier notre système de monitoring en redéfinissant le code logiciel du processeur. Nous pouvons ainsi modifier à volonté à quel instant on aimerait déclencher le monitoring, ou bien quel IP on aimerait espionner. Dans ce travail de monitoring, nous nous sommes intéressé à deux informations. La première représente le taux d'occupation des FIFO FSL (Utilisées ici comme interface entre le processeur et le port local du routeur). La deuxième est le temps que met un paquet pour atteindre sa destination (temps de routage).

Pour avoir la première information (taux d'occupation des FIFO FSL), nous avons décrit tout simplement un processus qui lit constamment la valeur du taux d'occupation du canal FSL et attend que cette valeur change. Si c'est le cas, cette valeur est associée au temps auquel elle s'est produite et envoyée directement au processeur chargé de gérer ces informations. La deuxième information (temps de

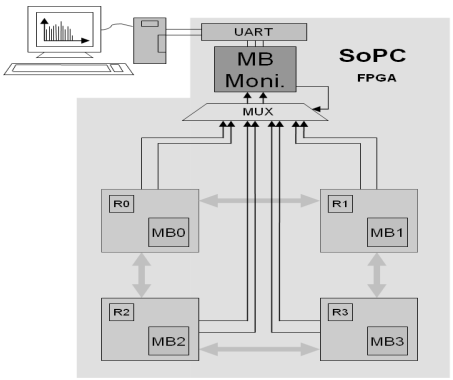


FIG. 5.15 – Plateforme du monitoring

routage d'un paquet), est obtenue grâce au 16 bits vacants dans l'entête. Ces derniers vont nous servir à écrire le temps auquel le début du routage a commencé, puis ce temps sera soustrait au temps global lorsque le paquet atteint sa destination. Ces deux informations sont envoyées au processeur espion à travers des liens FSL. Ces canaux FSL ont une profondeur maximale de 8192 cases. Une fois les données reçues par le processeur, elles peuvent être stockées dans une mémoire ou bien directement transférées vers la machine hôte à travers une liaison UART. La liaison série UART n'est pas très performante mais on l'a choisie car elle est simple à mettre en oeuvre. On peut très bien utiliser un autre moyen de transfert(USB, Ethernet...etc).

5.3.2.3 Decription de l'IP moniteur

La figure 5.16 illustre le schéma fonctionnel de l'IP utilisée pour le monitoring. Elle se compose principalement de registres, de comparateurs, de compteurs (pour les cycles) et de multiplexeurs.

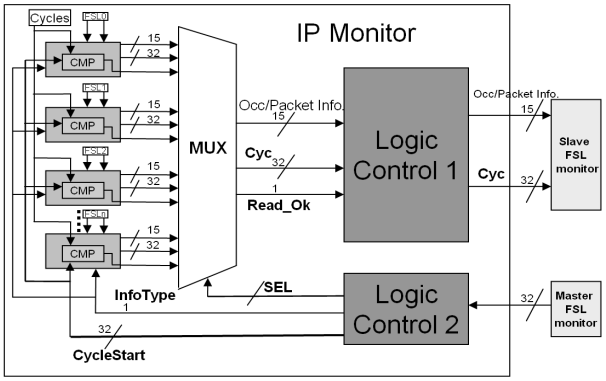


FIG. 5.16 – IP Moniteur

Le processeur chargé du monitoring envoie trois informations au multiplexeur à

travers un canal FSL. Ces informations représentent : l'IP choisie pour être espionnée (le signal SEL), l'information que nous voulons avoir (le signal InfoType) et l'instant de démarrage du monitoring (le signal CycleStart). Le signal SEL représente l'entrée sélection du multiplexeur. La sortie de ce dernier peut être le taux d'occupation de la FIFO ou bien l'information concernant le temps de routage des paquets. Le signal CYC est utilisé pour la variable temps (Time stamp) et Read_Ok informe la machine d'état qu'une nouvelle donnée est arrivée. Nous avons résumé dans le tableau 5.1 les ressources consommées sur le FPGA. Ce que nous présentons est un ratio entre les ressources utilisées par le système multiprocesseur et le système pour le monitoring.

	Min		Max	
	Used M	U.M/U.App.	Used M	U.M/U.App.
Slices	1,480	0.19	1,986	0.26
LUTs	1,672	0.12	2,628	0.19
FFs	1,406	0.18	1,990	0.24
Block RAMs	6	0.05	25	0.23

TAB. 5.1 – Ressources consommées

Ceci nous permet d'estimer la place que ce dernier occupe (U.M : les ressources utilisées par le système de monitoring et U.App : ressources utilisées par l'application multiprocesseur). Il est à signaler que les éléments du système pour monitoring sont : le processeur, l'IP pour le monitoring, FSL, contrôleur de RAM et l'UART. Le canal FSL utilisé pour envoyer les informations relatives au monitoring peuvent avoir de 2 à 8192 cases. On remarquera à travers le tableau 5.1 que même pour une utilisation maximale de ressource pour le système de monitoring, le taux d'occupation reste acceptable par rapport à l'application principale. Le seul problème viendrait des ressources en mémoire mais peut être facilement contourné en utilisant des mémoires externes.

La figure 5.17 représente l'utilisation de ressources par l'IP de moniteur comparée à l'utilisation de ressources pour l'ensemble de l'application. La comparaison a été faite avec différentes tailles de réseaux sur puce. D'après ces résultats, nous remarquons que les ressources utilisées par l'IP moniteur ne compromet pas l'implémentation de l'ensemble de l'application (Monitoring + NoC). Ceci est vérifié même pour des NoC d'assez grande taille (16 processeurs).

5.3.2.4 Résultats

Nous avons implémenté sur la plateforme NOC présentée précédemment, l'application de traitement d'images déjà utilisée dans le chapitre 4. Nous rappelons que celle-ci est constituée de trois filtres où chaque filtre est exécuté sur un processeur qui transfert, une fois son calcul achevé, à son processeur voisin qui effectuera un

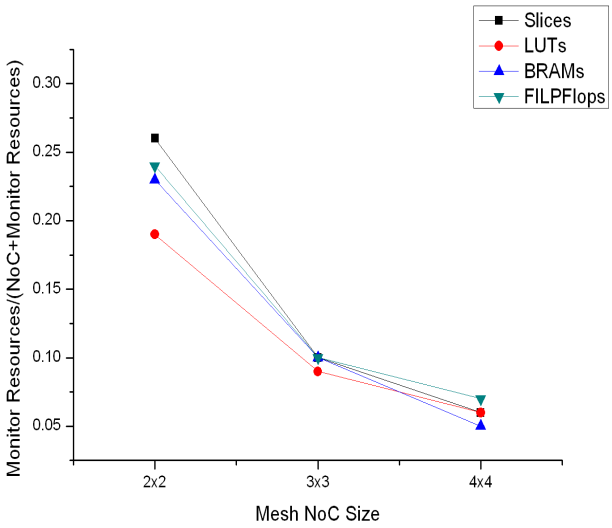


FIG. 5.17 – Ressources utilisées par l'IP moniteur par rapport à différentes plateformes NoC

autre type de filtrage. Pour notre application de monitoring, nous avons exécuté cette application de deux manières différentes.

Les Figure 5.18 et 5.19 illustrent respectivement une première façon d'implémentation non pipelinée et une deuxième façon pipelinée. Dans le premier cas, la communication et l'exécution sont réalisées de manière différée. Dans ce cas, toutes les interruptions de lecture ont été désactivées sauf pour le processeur MB0.

Cela explique la différence de temps de communication entre MB0-MB3 et le reste, démontrant ainsi le coût associé au déclenchement de l'interruption.

L'implémentation a été améliorée dans le second cas où l'exécution sur processeur recouvre la communication. Cette façon de programmer nous a fait obtenir une accélération de $2,14$. Les figures 5.20 et 5.21 donnent une représentation exhaustive de l'occupation des différents canaux FSL.

Dans ce travail, nous avons fixé une taille maximale de FSL. Ainsi, nous pouvons voir la profondeur maximale qui est atteinte pour chaque IP FSL. Le schéma 5.20 représente le taux d'occupation FSL pour la version non piplinée. Nous pouvons avoir une vue globale du comportement de la plateforme en réalisant une analyse de performance.

Les figures 5.22 et 5.23 représentent respectivement la valeur moyenne et le coefficient de variance. Ces résultats peuvent guider le concepteur à choisir la profondeur la plus appropriée pour chaque FSL ou alors fixer un intervalle pour son espace d'exploration.

Le tableau 5.2 donne les valeurs de temps maximale et minimale pour la trans-

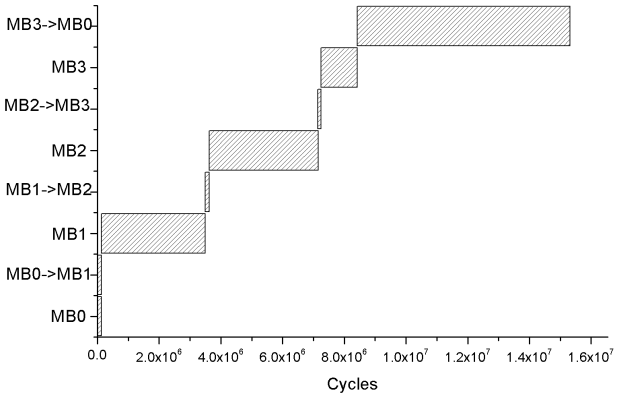


FIG. 5.18 – Sans Pipeline

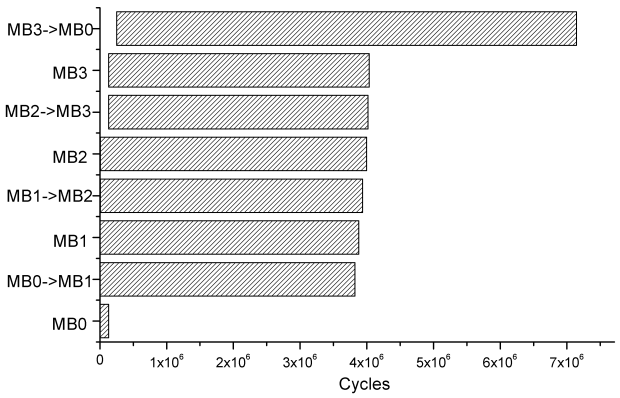
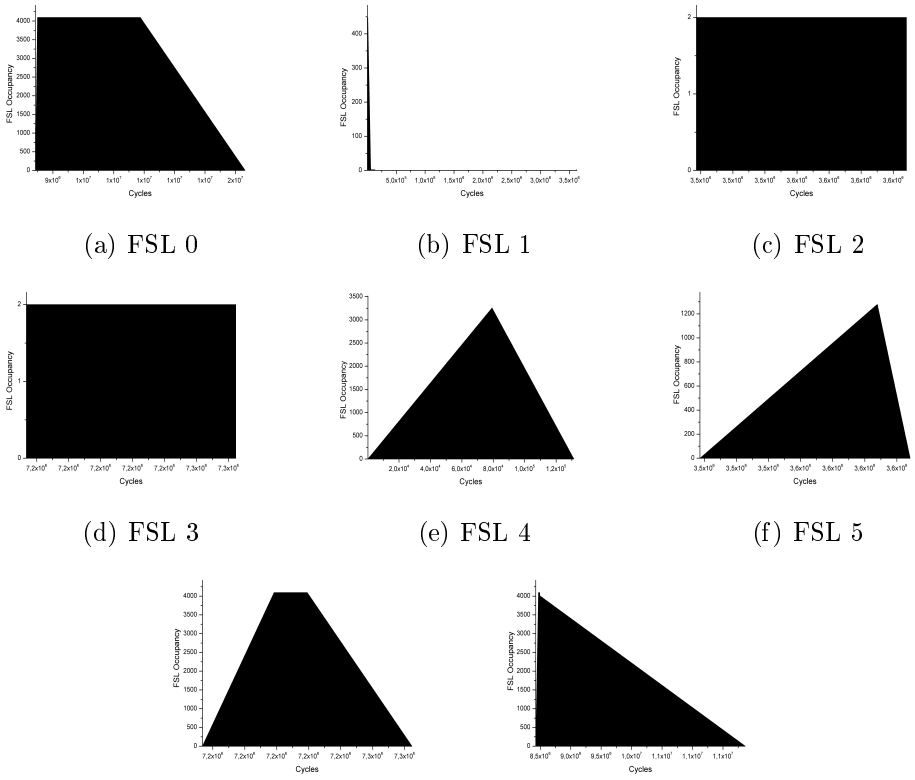


FIG. 5.19 – Pipeline

mission de paquets. Les paquets qui sont envoyés du processeur MB1 à MB2 et MB3 à MB0 ont un temps de transmission plus important que ceux transmis de MB0 à MB1 et à MB2 à MB3. C'est dû au fait qu'ils traversent un routeur additionnel pour être transmis. Les FSL esclaves par rapport aux processeurs sont FSL0, FSL 1, FSL 2 et FSL3. Le taux d'occupation de ces FSL varie de 2 à 448 entre la première version et la deuxième version (non pipelinée et pipelinée) sauf pour le canal FSL0 où sa valeur moyenne d'occupation est de 3053. Nous avons expliqué précédemment cela par le fait qu'il soit lié à une interruption pour lire son canal FSL associé.

Dans la version non pipelinée, les processeurs vont chercher les pixels à traiter de leurs FSL respectifs, puis ne font que traiter ces données et les stockent en mémoire avant de les renvoyer à l'élément de calcul suivant. Ceci explique le fait que les FSL



(a) FSL 0 (b) FSL 1 (c) FSL 2

(d) FSL 3 (e) FSL 4 (f) FSL 5

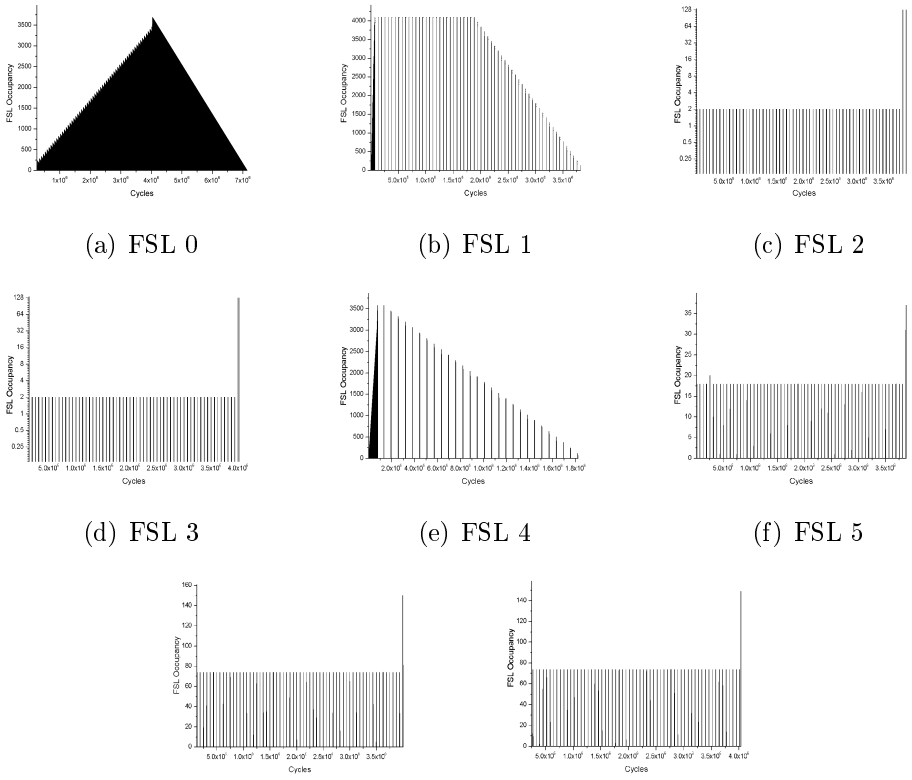
(g) FSL 6 (h) FSL 7

FIG. 5.20 – Traces FSL : implémentation non pipelinée

Source	Desti.	Min (Cyc)	Max (Cyc)	BW(Mbit/s)
MB0	MB1	44	71	13.08
MB1	MB2	54	90	12.91
MB2	MB3	44	71	13.12
MB3	MB0	54	86	7.60

TAB. 5.2 – Temps de routage

dans cette version ne se remplissent pas. Le meilleur exemple est donné par le FSL1. On remarque que ce dernier atteint au maximum une occupation de 448 alors qu'il passe à 4095 dans la version pipelinée où le processeur alterne entre traitement de données et communication. Au premier abord, on penserait que les FSL2 et FSL3 devraient avoir le même comportement que le FSL1 (dans la version pipelinée). Cependant, l'expérience montre que ces deux FSL ne se remplissent pas. Ceci est dû à la cadence avec laquelle le processeur en amont envoie les pixels est faible. Ce dernier envoie un pixel d'image traitée à la fois puis revient à sa tâche de traitement



(a) FSL 0 (b) FSL 1 (c) FSL 2

(d) FSL 3 (e) FSL 4 (f) FSL 5

(g) FSL 6 (h) FSL 7

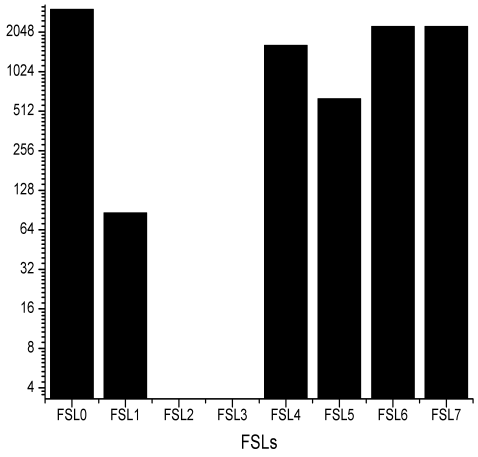
FIG. 5.21 – Traces FSL : implémentation pipelinée

d'image, excepté pour la dernière ligne qui est envoyée sans être traitée. Ceci est illustré dans la figure 5.23 où la valeur de la variance pour les FSL2 et FSL3 passe de 0,8 à 4. Le même phénomène se produit pour les FSL5, FSL6 et FSL7.

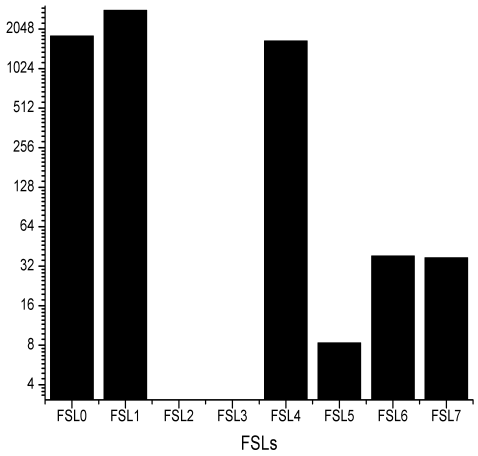
Les traces des figures 5.20 et 5.21 montrent que la version pipelinée, en plus du fait qu'elle nous fait gagner un "speed up" de 2, 14, permet une consommation de ressource en mémoire nettement moindre. Effectivement, la valeur moyenne de l'occupation FSL passe de 2158 à 1494 entre la version non pipelinée et pipelinée.

5.3.3 Flot de synthèse NoC : Occam & Monitoring

Le monitoring de réseaux sur puce peut être employé avantageusement dans différents contextes. Dans ce cas-ci, une première application qu'on peut dégager du monitoring est pour les programmeurs parallèles qui utiliseront ce système afin de paramétrer leurs applications en temps réel. Ce système pourra être utilisé dans le cas d'une conception basée sur plateforme [142] (platform based design) où le réseau sur puce a été conçu pour un ensemble d'applications cibles. La seconde application



(a) 1st Prog. Ver.

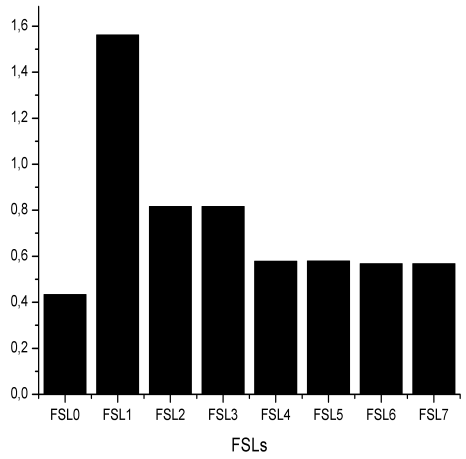


(b) 2nd Prog. Ver.

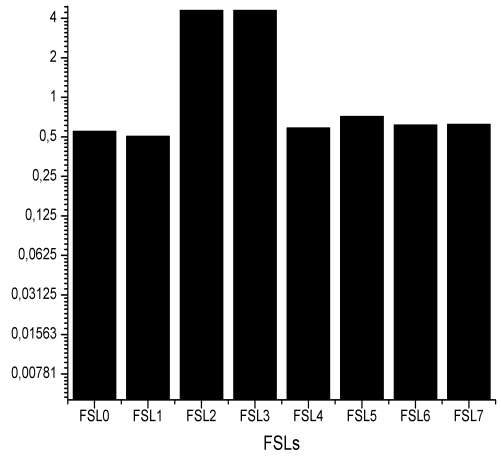
FIG. 5.22 – Valeur moyenne de l'occupation FSL

se rapproche plus du contexte des systèmes embarqués.

Les systèmes sur puce représentent des plateformes paramétrables où une exploration est nécessaire afin de spécifier les différents éléments de calculs et autres modules de communication. Le travail présenté précédemment (chapitre 4) [143] sur l'exploration multi-objectif de l'espace de conception d'un système multiprocesseur sur puce n'offrait pas une application permettant de spécifier l'amplitude de paramètres du réseau sur puce. Nous fixions ceci de manière arbitraire rendant plus compliquée l'exploration multi-objectif. Effectivement, l'algorithme se dirigera dans



(a) 1st Prog. Ver.



(b) 2nd Prog. Ver.

FIG. 5.23 – Coefficient de variance de l'occupation FSL

l'espace de conception vers des paramètres qui n'amélioreront pas l'implémentation, mettant ainsi plus de temps à converger. Le monitoring de systèmes sur puce proposé permettra d'apporter une solution au problème de définition de l'amplitude des paramètres. En effectuant au préalable (avant une exploration) un monitoring, nous pourrons connaître, de manière instantanée, l'amplitude des paramètres de chaque composant du système.

Ainsi, l'algorithme d'exploration n'aura dans son espace de conception que des valeurs réellement atteintes par les éléments le constituant. Cette méthode représente

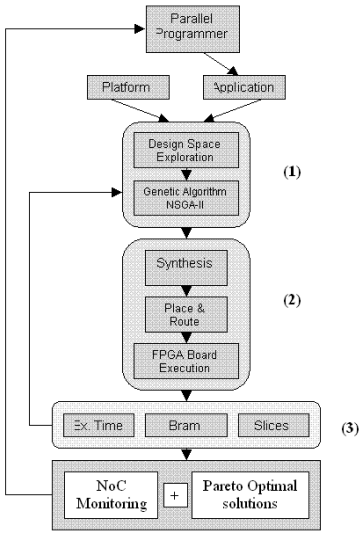


FIG. 5.24 – Programmation parallèle avec synthèse MPSoC (Espace/Exécution) avec monitoring de NOC

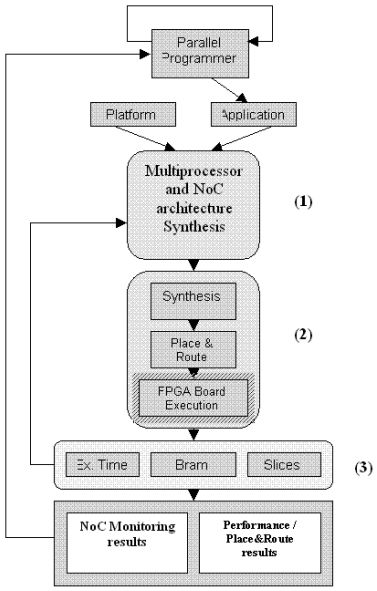


FIG. 5.25 – Programmation parallèle avec exploration de l'espace de conception (Espace/Exécution) et de NOC

le seul moyen de remonter de façon instantanée ce genre de résultats. Le monitoring peut être intégrée au flot de synthèse d'application parallèle à base d'OCCAM suivie d'une exploration multiobjectif, telle présentée dans le chapitre 4, réalisant ainsi un flot de conception complet répondant efficacement à la conception de systèmes multiprocesseurs sur puce (Figure 5.25).

Les résultats récoltés par le monitoring peuvent être utilisés pour la reconfiguration dynamique des FPGAs. On peut remarquer sur les figures 5.20 et 5.21 des occupations FSL sous formes triangulaires avec une variation assez importante. Cela veut dire que le besoin en profondeur pour le canal FSL varie considérablement et que le maximum n'est atteint que pendant un temps très petit. Aussi, on peut remarquer dans le taux d'utilisation des canaux FSL, qu'un lien FSL peut être bien rempli alors qu'un autre est vide ou moyennement rempli (FSL0 et FSL4 dans figure 5.21). Sachant l'importance des ressources mémoires dans un système sur puce, il serait nécessaire de trouver un moyen afin de les exploiter intelligemment.

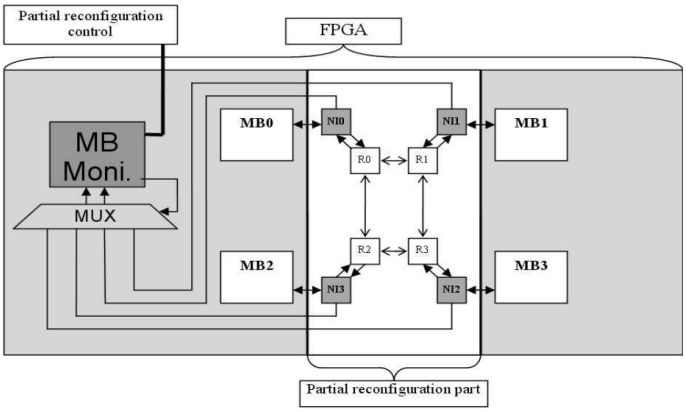


FIG. 5.26 – Reconfiguration partielle dynamique sur un NoC

La reconfiguration dynamique partielle des FPGA associée à l'architecture de monitoring que nous avons proposée (Figure 5.26) peut pallier à ce type de problèmes. Celle-ci est possible sous certaines contraintes rigoureuses de cheminement de signaux et de placement sur le FPGA (dans notre cas il s'agit d'un FPGA Xilinx). Avec les circuits FPGA Xilinx, la reconfiguration dynamique partielle impose l'utilisation de slices sur la longueur du FPGA. Par conséquent, la place utilisée pour la reconfiguration dynamique partielle doit être exploitée de manière optimale [144, 145]. Enfin du fait que la synthèse de NoC se fait sur circuit FPGA une extension intéressante est décrite dans [146] qui exploite les spécificités des structures d'interconnexion des FPGA pour un mapping optimal.

5.4 Conclusion

Les réseaux sur puce permettent d'assurer une communication performante entre les différents éléments de calcul d'un système multiprocesseur sur puce. Cependant, il ne peuvent être efficacement exploités que s'ils répondent au mieux aux exigences de l'application à implémenter. Le but est de synthétiser le schéma de communication d'une application parallèle à partir de sa spécification. Cette synthèse de communication a été réalisée ici grâce au langage de programmation parallèle Occam. Ce

langage permet une description simplifiée et naturelle des applications parallèles. De plus, le fait qu'il soit prouvable mathématiquement réduit considérablement les tâches de vérification dans le cycle de conception. Un autre avantage avec Occam réside dans ses transformations. Le fait que celles-ci se fassent facilement tout en garantissant une communication sans erreurs (deadlock, starvation...etc), nous permet de l'utiliser aisément pour l'exploration logicielle du schéma de communication. Ainsi, Occam permet d'exécuter le code d'une application parallèle soit en tant que threads (sur un unique élément de calcul) ou bien sur un réel système multiprocesseur. En somme, Occam nous a ouvert la possibilité de synthétiser à partir du code d'une application parallèle son schéma de communication. Cependant, il ne nous permet pas de connaître le comportement des modules de communication pour chaque topologie générée. Nous avons pensé que la taille des puces dans les systèmes embarqués actuels permettait d'introduire, en plus du système multiprocesseur, une structure de monitoring afin de recueillir des informations concernant le comportement des modules de communication et de l'état du réseau. Cela nous procure des informations instantanées en évitant le temps prohibitif de la simulation. Après avoir vérifié la faisabilité de la synthèse à partir d'Occam et du monitoring sur puce, nous avons proposé d'intégrer ce système dans le flot d'exploration multi-objectif et le flot de synthèse Occam (MOCDEX [132] et MOCSOC[143]) afin d'extraire une architecture de communication répondant naturellement à la spécification de l'application parallèle, et de définir concrètement l'intervalle des paramètres de l'espace de conception, permettant ainsi à l'algorithme d'exploration d'effectuer une recherche efficace. Ce flot de conception pour multiprocesseur sur puce est entièrement automatique et basé sur des résultats réels (Évitant le temps prohibitif de simulation et les estimations) partant d'une spécification parallèle haut niveau jusqu'à une implémentation réelle sur circuit. Nous avons aussi dans ce chapitre ouvert la porte à des améliorations futures concernant l'utilisation de threads à partir d'Occam (raffinant la granularité de la synthèse MPSoC) et la reconfiguration dynamique partielle qui consiste à optimiser dynamiquement l'utilisation des ressources sur FPGA selon leur utilisation.

Chapitre 6

Conclusion

L'augmentation continue de la capacité d'intégration d'une part, la complexité croissante des applications embarquées d'autre part, ont conduit aux systèmes sur puce (SoC) puis aux systèmes multiprocesseurs sur puce (MPSoC). Une spécificité de la conception de ces systèmes est de devoir aborder conjointement la conception de la partie matérielle et la conception de la partie logicielle et plus particulièrement le partitionnement et l'articulation entre ces deux parties. Les outils de conception doivent être capables de travailler à tous les niveaux d'abstraction afin d'exploiter efficacement les ressources sur puce mises à disposition tout en conservant des temps de conception raisonnables pour respecter la contrainte de temps de mise sur le marché.

Dans cette thèse, nous avons présenté une méthodologie de conception pour les systèmes multiprocesseurs sur FPGA, dont l'originalité porte sur trois aspects :

- L'utilisation de circuits logiques reconfigurables (FPGA) permet d'éviter des temps prohibitifs de simulation ou de co-simulation matériel logiciel par une exécution directe de l'application sur différentes instances de l'architecture MPSoC.
- Une exploration intelligente de l'espace de conception à l'aide d'algorithmes de recherche évolutionnaires multi objectifs permet de contourner l'impossibilité d'une recherche exhaustive de l'espace d'exploration architecturale tout en obtenant des résultats proches des solutions optimales.
- L'utilisation d'un langage de haut niveau (OCCAM) permet de faire rapidement la synthèse d'architectures multiprocesseur, notamment du réseau d'interconnexion, à partir de la description de l'application. Combinée avec le monitoring sur puce, cette possibilité permet d'adapter plus facilement l'architecture à l'application.

Dans un premier temps, nous avons validé l'exploration évolutionnaire multi objectifs sur un système monoprocesseur sur trois benchmarks relativement simples : deux filtres de traitement d'images (le filtre conservatif et le filtre médian) et sur le codeur entropique de JPEG2000). Les objectifs à optimiser étaient : le nombre

de " slices " du FPGA, la taille des mémoires, le temps d'exécution. Nous avons constaté l'importance de considérer l'ensemble des paramètres sensibles d'une plateforme système sur puce comme étant un tout indissociable lors d'une exploration architecturale. Aussi, la recherche exhaustive a montré ses limites et ses faiblesses face à des systèmes de plus en plus gros et riches. L'algorithme NSGA-II utilisé a bien su s'adapter à la diversité de solutions qui existent dans l'espace d'exploration des systèmes sur puce actuels. Les résultats d'exploration au sens de Pareto avec NSGA-II ont démontré, en utilisant la notion d'élitisme, la qualité d'une telle démarche par rapport à une exploration exhaustive suivie d'une classification de l'ensemble des solutions au sens de Pareto. L'autre valeur ajoutée réside dans le fait d'avoir utilisé la capacité d'intégration des gros FPGA afin de les utiliser en tant que plateforme d'émulation. La reconfigurabilité simple et rapide des FPGAs permet de programmer rapidement l'architecture du système sur puce, puis de mesurer les temps d'exécution. Cette méthode permet d'éviter le temps prohibitif de simulation pour effectuer l'évaluation des différentes configurations de l'espace de recherche.

Un système monoprocesseur est constitué d'un processeur et d'un certain nombre d'IPs accélératrices, qui sont des unités de calcul indépendantes communiquant avec le processeur. Le passage aux systèmes multiprocesseurs est donc l'étape naturelle suivante, que nous avons effectuée sur une grille 2D de quatre processeurs sur l'enchaînement de trois filtres de traitement d'images (filtre médian, filtre moyenne et filtre conservatif). Les résultats obtenus ont prouvé leur efficacité en terme de qualité des solutions et de temps d'exploration. Cependant, les résultats que produit cette méthode sont complètement dépendants de l'architecture initiale considérée. Si l'architecture multiprocesseur fournie à l'entrée du flot n'est pas optimisée pour l'application, les résultats obtenus seront sous optimaux. Pour remédier à cela, nous avons proposé une synthèse d'applications parallèles à partir d'un langage parallèle haut niveau. Le langage de programmation Occam a été choisi pour sa simplicité, sa prouvabilité mathématique et sa facilité dans ses transformations. Nous avons dans cette thèse pu démontrer la faisabilité d'une telle démarche en présentant des résultats d'expériences pratiques sur un cas d'étude. Cependant, en joignant les résultats d'exploration multi-objectif et de la synthèse de réseaux sur puce avec Occam, nous avons constaté qu'il serait important de pouvoir fixer de manière efficace (précision et rapidité) l'espace des paramètres des différents modules de l'architecture multiprocesseur. Par conséquent, nous avons utilisé les ressources matérielles disponibles dans les gros FPGA permettant d'y intégrer, en plus du système multiprocesseur, une structure de monitoring remontant instantanément les résultats comportementaux des IPs formant la plateforme multiprocesseur et du trafic entre les différents composants. Ce monitoring permet de collecter rapidement des informations utilisables pour fixer avec précision l'espace de recherche pour une exploration : elles peuvent être réinjectées comme entrée pour la spécification parallèle ou comme données pour la reconfiguration dynamique sur FPGA de certains éléments déterminants de l'architecture. Comparée à la simulation, cette façon de faire procure des résultats

d'implémentation réels dans un temps égal à une exécution. Cette approche, utilisant OCCAM et le monitoring, a été testée et validée avec 9 processeurs MicroBlaze sur un réseau de neurones de type SOM (Self Organizing Map).

Nous avons, grâce à cette méthodologie de conception, obtenu des résultats prometteurs et satisfaisants. Compte tenu du temps disponible, nous nous sommes limités à des applications (traitement d'images, réseau de neurones) de taille limitée. Il fallait dans un premier temps valider complètement les différents points fondamentaux de la méthodologie. De plus, avoir des applications de taille limitée permettait d'utiliser uniquement des blocs de RAM FPGA pour l'implémentation du code exécutable relatif à chaque élément de calcul de notre plateforme multiprocesseur sans avoir à utiliser des mémoires externes au FPGA.

La méthodologie de conception que nous avons présentée va voir son intérêt croître avec les progrès continus des performances des FPGA. Ces derniers proposent plus de ressources en éléments logiques reconfigurables (104.882 pour le Virtex-II contre 330.000 pour le Virtex-5), en mémoires, en processeurs enfouis et en performance (65 nm à 550 Mhz pour le Virtex-5 de Xilinx). Le champ d'applications des FPGA va donc continuer à s'étendre. La dernière génération de FPGA Xilinx permet un accès plus simple à la reconfiguration dynamique partielle et d'avoir aussi un mécanisme de monitoring, via la chaîne JTAG [5], de température et d'alimentation. Ce système constitué d'éléments analogiques est enfoui au sein du FPGA, permettant dans les flots de conception futurs d'intégrer la consommation d'énergie. Le flot que nous proposons ici n'est pas destiné exclusivement à un produit FPGA. L'architecture obtenue peut être soit laissée sur FPGA, soit migrée vers des circuits spécifiques ASIC. En effet, les fabricants FPGA proposent de migrer de manière transparente et directe d'une implémentation FPGA à un circuit " full custom " ASIC, permettant, dans un cas nominal, un gain de performance de 100% et une réduction de consommation de 50% [50].

Les résultats prometteurs obtenus dans l'exploration évolutionnaire multi-objectif, l'émulation sur FPGA versus simulation, la synthèse d'applications parallèles, le monitoring en temps réel d'applications systèmes sur puce et la reconfiguration dynamique, sont une première étape. Des travaux supplémentaires sont nécessaires pour évaluer la méthodologie sur des applications de taille plus conséquente et sur d'autres classes d'applications, pour tester ses limites éventuelles.

Liste des publications[1]

Publications en conférences

1. Imed Aouadi, Riad Benmouhoub, Omar Hammami, "Exploring JPEG-2000 Entropy Coder Implementations on Xilinx Virtex-II Pro Platform",12th European Signal Processing Conference September 6-10, 2004 Vienna, Austria
2. Riad Benmouhoub, Imed Aouadi, Omar Hammami, "System on programmable chip platform based design of JPEG-2000 entropy coder", The 12th Workshop on Synthesis And System Integration of Mixed Information technologies, 18-19 Oct 2004, Kanazawa, Japan
3. Riad Benmouhoub, Omar Hammami, "System-Level Design Methodology with Direct Execution for Multiprocessors on SoPC" IEEE Seventh International Symposium on Quality Electronic Engineering Design (ISQED 2006), Pages 781-786, 27-29 March 2006, San Jose, California
4. Riad Benmouhoub, Omar Hammami, "MoCSoC : Multiprocessor on Chip Synthesis from OCCAM", Pages 246-252, The 13th Workshop on Synthesis and System Integration of Mixed Information technologies, 3-4 Apr 2006, Nagoya, Japan.
5. Riad Benmouhoub, Omar Hammami, "Networks on Chip Real Time Monitoring Feedback for Multiprocessors Systems on Chip Parallel Programmers", the 4th international IEEE-NEWCAS 2006, Pages 141-144, 18-21 June, Quebec, Canada
6. Riad Benmouhoub, Omar Hammami, "NoC Monitoring Hardware Support for Fast NoC Design Space Exploration and Potential NoC Partial Dynamic Reconfiguration" IEEE Symposium On Industrial Embedded Systems IES, 18-20 Oct 2006

Publication dans des Journaux

1. Riad Benmouhoub, Omar Hammami, MOCDEX :Multiprocessor on Chip Multiobjective Design Space Exploration with Direct Execution, Volume 2006, Eurasip journal on Embedded Systems, Article ID 54074, 14 pages, Hindawi

[1]Les articles de conférences 4,6 et l'article de journal sont présentés dans l'annexe 1

Index

Bibliographie

[1] Aimen Bouchhima. Modélisation du logiciel embarqué à différents niveaux d'abstraction en vue de la validation et de la synthèse des systèmes monopuces. Available on : `http://www.TIMA.com/`, Mai 2003.

[2] D. C. Lee, S. J. Harper, P. M. Athanas, and S. F. Midkiff. A stream-based reconfigurable router prototype. In *Proceedings of the IEEE Symposium on Field Programmable Gate Arrays for Custom Computing Machines (FCCM)*, pages 581–585, 1999.

[3] M. Horowitz and W. Dally. How scaling will change processor architecture. In *Digest of Technical Papers. ISSCC. 2004 IEEE International Solid-State Circuits Conference*, volume 1, pages 174 – 190, 2004.

[4] mathstar. mathstar fpoa. Available on : `http://www.mathstar.com/`.

[5] Xilinx. Available on : `http://www.xilinx.com/`.

[6] A. Fin, F. Fummi, M. Martignano, and M. Signoretto. SystemC : A homogenous environment to test embedded systems. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign (CODES-01)*, pages 17–22. ACMPress, April 2001.

[7] Object Management Group. Omg unified modeling language specification version 1.3.

[8] Celoxica. Handel-c. Available on : http ://www.celoxica.com/.

[9] R.K. Gupta, C.N. Coelho, and G.D. Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proc. of the DAC*, 1992.

[10] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. Hardware-software co-design of embedded systems : The polis approach. In *Kluwer Academic Press*, 1997.

[11] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. In *Design and Test of Computers, IEEE*, volume 10, pages 64–75, Dec 1993.

[12] Robert K. Brayton, Alberto Sangiovanni-Vincentelli, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, and et al. Vis : A system for verification and synthesis. In *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, 1996.

[13] A. Kalavade and E.A. Lee. A global criticality/local phase driven algorithm for theconstrained hardware/software partitioning problem. In *Proceedings of the Third International Workshop on Hardware/Software Codesign*, pages 42–48, Sep 1994.

[14] Asawaree Kalavade and P.A.Subrahmanyam. Hardware/software partitioning for multi-function systems. In *International Conference on Computer Aided Design, Proceedings of the IEEE/ACM international conference on Computer-aided design*, pages 516 – 521, 1997.

[15] K.S. Chatha and R. Vemuri. Magellan : multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs. In *International Conference on Hardware Software Codesign*, pages 42–47, 2001.

[16] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha. Cosyn : hardware-software co-synthesis of embedded systems. In *Annual ACM IEEE Design Automation Conference*, pages 703–708, 1997.

[17] Theodore W. Manikas and James T. Cain. Genetic algorithms and simulated annealing algorithm. In *Annual ACM IEEE Design Automation Conference*, pages 96–101, May 1996.

[18] William E. Hart. A theoretical comparison of evolutionary algorithms and simulated annealing. In *proceedings of the Fifth Annual Conf. on Evolutionary Programming*, 1996.

[19] Glover F. Future paths for integer programming and links to artificial intelligence. In *Computers and Operation research*, pages 533–549, 1986.

[20] M. Laguna F. Glover. Tabu search. In *Kluwer Academic Publishers*, July 1997.

[21] D. de Werra A. Hertz, E. Taillard. A tutorial on tabu search. In *Proc. of Giornate di Lavoro AIRO'95 (Enterprise Systems : Management of Technological and Organizational Changes)*, pages 13–24, Italy, 1995.

[22] F. Glover. Tabu search, part i. In *ORSA Journal on Computing 2*, 1989.

[23] F. Glover. Tabu search, part ii. In *ORSA Journal on Computing 2*, 1990.

[24] M. Ehrgott and X. Gandibleux. Approximative solution methods for multiobjective combinatorial optimization. In *ORSA Journal on Computing 2*, volume 12, pages 1–88, June 2004.

[25] J. H. Holland. Outline for a logical theory of adaptative systems. In *Journal of the association of computing machinery*, 1962.

[26] K. E. De Jong. An analysis of the behavior of a class of genetic adaptive systems. In *University of Michigan Press, Ann Arbor*, 1975.

[27] D. E Goldberg. Genetic algorithms in search, optimization and machine learning. In *Addison Wesley*, 1989.

[28] James. E. Baker. Reduicing bias and inefficiency in the selection algorithm. In *In J. J. Grefenstette, editor proceeding of the 2nd international conference on genetic algorithms*, pages 14–21, 1987.

[29] B. Sareni. Méthodes d'optimisation multimodales associées à la modélisation numérique en électromagnétisme. In *thèse de doctorat école centrale de Lyon, France*, 1999.

[30] E. Zitzler and L. Thiele. An evolutionary algorithm for multiobjective optimization : the strength pareto approach. In *TIK Report*, 1998.

[31] David W. Corne, Joshua D. Knowles, and Martin J. Oates. The pareto envelope-based selection algorithm for multiobjective optimization. In *In Proceedings of the Sixth International Conference on Parallel Problem Solving from Nature (PPSN VI)*, pages 839–848, 2000.

[32] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization : Nsga-ii. In *In Proceedings of the Parallel Problem Solving from Nature VI (PPSN-VI)*, 2000.

[33] N. Srivinas and K. Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. In *technical report, department of mechanical engineering, institute of technology India*, 1993.

[34] Tony Givargis, Frank Vahid, and Jörg Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-chip. In *IEEE Transactions on VLSI Systems*, volume 10, pages 416–422, April-Aug 2002.

[35] T.Givargis and F.Vahid. Platune : A tuning framework for system-on-a-chip platforms. In *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, volume 21, pages 1317–1327, Nov 2002.

[36] Frank Vahid, Tony Givargis, and Jörg Henkel. Evaluating power consumption of parameterized cache and bus architecture in system-on-a-chip designs. In IEEE, editor, *Transactions on Very Large Scale Integration (VLSI) Systems*, volume 9, 2001.

[37] G.Ascia, V.Catania, and M.Palesi. A ga-based design space exploration framework for parameterized system-on-a-chip platforms. In *IEEE Transaction on Evolutionary Computation*, volume 8, pages 329–346, Aug 2004.

[38] K. Ghali. Méthodologie de conception système à base de plateformes reconfigurables et programmables. In *Thèse Orsay*, 2005.

[39] E.Zitzler, K.Deb, and L.Thiele. Multiobjective evolutionnary algorithms : A comparative case study and the strength pareto approach. In *IEEE Trans. On Evolutionary Computation*, pages 257–271, 1999.

[40] M. Ehrgott and X. Gandibleux. Approximative solution methods for multiobjective combinatorial optimization. In *SBCCI '05 : Proceedings of the 18th annual symposium on Integrated circuits and system design*, volume 12, pages 1–88, June 2004.

[41] S. Choi and C. Wu. Partitioning and allocation of objects in heterogeneous distributed environments using the niched pareto genetic algorithm. In *In Proceedings of 1998 Asia Pacific Software Engineering Conference APSEC*, page 322Ũ329, Dec 1998.

[42] K. Ben Chehida, P. Guitton-Ouhamou, S. Raimbault, and M. Auguin. A multiobjective hardware-software partitioner for dynamically recongurable system design. *WSEAS Transactions on Systems*, pages 741–745, 2003.

[43] K. Ben Chehida, S. Raimbault, and M. Auguin. Partitionnement logiciel matériel ciblant une architecture reconfigurable dynamiquement. *Technique et Science Informatiques, Architecture des ordinateurs*, pages 737–757, 2003.

[44] J.P Diguet, G. Gogniat, J.L Philippe, Y. Le Moullec, S. Bilavarn, C. Gamrat, K. Ben Chehida, M. Auguin, X. Fornari, and P. Kajfasz. Epicure : A partitioning and co-design framework for reconfigurable computing. *Microprocessors and Microsystems, Special Issue on FPGA's*, 30(6) :367–387, September 2006.

[45] Robert P. Dick and Niraj K. Jha. Mogac : A multiobjective genetic algorithm for hardware-software co-synthesis of hierarchical heterogeneous distributed embedded systems. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 920–935, October 1998.

[46] C. Grecu, P. Pande, A. Ivanov, and R. Saleh. A scalable communication-centric soc interconnect architecture. In *IEEE International Symposium on Quality Electronic Design*, pages 343– 348, 2004.

[47] N. Thepayasuwan, S. Kallakuri, A. Doboli, and S. Doboli. Communication subsystem synthesis and analysis tool using bus architecture generation and stochastic arbitration policies. In *IEEE International Symposium on Circuits and Systems*, volume 2, pages 1044– 1047, May 2005.

[48] Dongwan Shin, Samar Abdi, and Daniel D. Gajski. Automatic generation of bus functional models from transaction level models. In *ASP-DAC '04 : Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 756–758, 2004.

[49] M. Ariyamparambath, D. Bussaglia, B. Reinkemeier, T. Kogel, and T. Kempf. A highly efficient modeling style for heterogeneous bus architectures. In *International Symposium on System-on-Chip*, pages 83–87, Nov 2003.

[50] Altera. Available on : `http://www.altera.com/`.

[51] Xilinx. Xilinx microblaze soft core processor. Available on : `http://www.xilinx.com/ise/embedded/mb\_ref\_guide.pdf`.

[52] Xtensa. tensilica. Available on : `http://www.tensilica.com/`.

[53] PowerPC 405. Ibm. Available on : `http://www.ibm.com/chips/techlib/techlib.nsf/products/PowerPC\_405\_Embedded\_Cores/`.

[54] Xilinx. Embedded developement kit. Available on : http ://www.xilinx.com.

[55] D.Taubman and M.W.Marcellin. Jpeg2000 - image compression fundamentals, standards and practice. In *Kluwer Academic Publishers*, 2001.

[56] R. Benmouhoub, I. Aouadi, and O. Hammami. System on programmable chip plateform based design of jpeg-2000 entropy coder. In *Workshop on synthesis and system integration of mixed information technologies (SASIMI'04)*, pages 103–106, Oct 2004.

[57] I. Aouadi, R. Benmouhoub, and O. Hammami. System on a programmable chip oriented jpeg-2000 entropy implementation for multimedia embedded systems. In *The International Conference on Consumer Electronics (ICCE)*, Jan 2005.

[58] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies. Image coding using the wavelet transform. In *IEEE Transactions on Image Processing*, volume 2, pages 205–220, Apr 1992.

[59] M. J. Gormish, D. Lee, and M. W. Marcellin. Jpeg 2000 : Overview, architecture and applications. In *Proc. IEEE Int. Conf. Image Processing ICIP*, pages 29–32, Sept 2000.

[60] D. Santa Cruz and T. Ebrahimi. An analytical study of the jpeg2000 functionalities. In *Proc. IEEE Int. Conf. Image Processing ICIP*, volume 2, pages 49–52, Sept 2000.

[61] ALPHA DATA. Adm-xrc-ii pci mezzanine card user guide. In *IEEE Transactions on Image Processing, Version 1.5.*

[62] K.Ghali, O.Hammami, and I.Hermann. Multiobjective design of embedded processors on fpga platforms. In IEEE, editor, *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops (ICDCSWŠ04)*, pages 871–875, 2004.

[63] D. Kulkarni, W.A. Najjar, R. Rinker, and F.J. Kurdahi. Compile-time area estimation for lut-based fpgas. *ACM Trans. Des. Autom. Electron. Syst.*, 11(1) :104–122, 2006.

[64] P. Paulin. Emerging challenges for mpsoc platforms. *MPSoC2006 (6th international forum on Application-Specific Multiprocessor SoC).*

[65] A.A. Jerraya, A. Bouchhima, and F. Pétrot. Programming models and hw-sw interfaces abstraction for multi-processor soc. In *DAC '06 : Proceedings of the 43rd annual conference on Design automation*, pages 280–285, 2006.

[66] G. Martin. Overview of the mpsoc design challenge. In *43rd ACM/IEEE Design Automation Conference*, pages 274–279, July 2006.

[67] L. Xue, O. Ozturk, F. Li, M. Kandemir, and I. Kolcu. Dynamic partitioning of processing and memory resources in embedded mpsoc architectures. In *IEEE Design, Automation and Test in Europe DATE*, volume 1, pages 261–272, Mar 2006.

[68] M.D. Nava, P. Blouet, P. Teninge, M. Coppola, T. Ben-Ismail, S. Picchiottino, and R. Wilson. An open platform for developing multiprocessor socs. In *IEEE Computer*, volume 38, pages 60–67, Mar 2005.

[69] A. Grasset, F. Rousseau, and A.A. Jerraya. Network interface generation for mpsoc : From communication service requirements to rtl implementation. In *Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping (RSP'04)*, pages 66–69, 2004.

[70] W. O. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, and A. A. Jerraya. Colif : a multilevel design representation for application-specific multiprocessor system-on-chip design. In *IEEE Proceedings of the 12th International Workshop on Rapid System Prototyping*, page 110, 2001.

[71] M.P. Bonaciu. Plateforme flexiblepour l'exploitation d'algorithmeset d'architectures en vue de la réalisation d'application vidéo haute définition sur des architectures multiprocesseurs monopuce. Available on : `http://www.TIMA.com/`, May 2006.

[72] D. LYONNARD. An approach for the systematic gathering of interface items toward the generation of multiprocessor architectures. Available on : `http://www.TIMA.com/`, Apr 2003.

[73] I. PETKOV. Design of multiprocessor system on chip : Link between simulation and realization. Available on : `http://www.TIMA.com/`, Jan 2006.

[74] Y. Sheynin, E. Suvorova, and F. Shutenko. Complexity and low power issues for on-chip interconnections in mpsoc system level design. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, volume 00, page 6 pp, Mar 2006.

[75] R. Ho, K. Mai, and M. Horowitz. The future of wires. In *PROCEEDINGS OF THE IEEE*, volume 89, pages 490–504, Apr 2001.

[76] W. Dally and B. Towles. Route packets not wires : On-chip inttercnnection networks. In *Design Automation Conference*, pages 684–689, June 2001.

[77] A.A.Jerraya and W.Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, June 2004.

[78] P. Wielage and K. Goossens. Networks on silicon : Blessing or nightmare ? In *IEEE Proceedings of the Euromicro Symposium on Digital Systems Design*, page 196, 2002.

[79] J. Xu and W. Wolf. Networks and applications : Are application-specific networks worth the trouble ? *Workshop on Future Interconnection and Network on Chip*, March 2006.

[80] Giovanni De Micheli and Luca Benini. Networks on chips. In *Morgan Kaufman*, 2006.

[81] Jorg Henkel, Wayne Wolf, and Srimat Chakradhar. On-chip networks : A scalable, communication-centric embedded system design paradigm. In *Proceedings of the 17th International Conference on VLSI Design*, pages 845–851, Nov 2004.

[82] L. Benini and D. Bertozzi. Network-on-chip architectures and design methods. In *IEE Proceedings on Computers and Digital Techniques*, volume 152, pages 261–272, Mar 2005.

[83] Lionel M. Ni. Issues in designing truly scalable interconnection networks. In *Workshop on Parallel Processing*, pages 74–83, Aug 1996.

[84] ARM. Amba 2.0 specification. Available on : `http://www.arm.com/products/solutions/AMBAOverview.html/`.

[85] X. Ningyi, L. Xianglun, L. Renfei, and Z. Zucheng. A systemc-based noc simulation framework supporting heterogeneous communicators. In *IEEE 6th International Conference On ASIC*, volume 2, pages 1032–1035, Oct 2005.

[86] P. Wielage and K. Goossens. Networks on silicon : Blessing or nightmare ? In *Proceedings of the Euromicro Symposium on Digital Systems Design*, pages 196–200, 2002.

[87] P. Aldworth. System-on-a-chip bus architecture for embedded applications. In *IEEE International Conference on Computer Design*, pages 297–298, 1999.

[88] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *IEEE Proceedings of the Design Automation Conference*, pages 667–672, 2001.

[89] T.Bjerregaard and S.Mahadevan. A survey of research and practices of network-on-chip. In *ACM Computing Surveys (CSUR) archive*, volume 38, 2006.

[90] E. Rijpkema et al. Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *DATE conference*, pages 350–355, Mar 2003.

[91] Davide Bertozzi, Antoine Jalabert, Srinivasan Murali, Rutuparna Tamhankar, Stergios Stergiou, Luca Benini, and Giovanni De Micheli. Noc synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Transaction on Parallel and Distributed Systems*, Vol. 16 :113–129, Feb 2005.

[92] D. Bertsekas and R. Gallager. Data networks. In *Prentice Hall*, 1991.

[93] J. Walrand and P. Varaiya. High-performance communication networks. In *Morgan Kaufman*, 2000.

[94] A.Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C.A. Zeferino. Spin : A scalable, packet switched, on-chip micro-network. In *IEEE Computer Society Proceedings of the conference on Design, Automation and Test in Europe*, page 70Ű73, 2003.

[95] A. Laffely, J.Liang, R. Tesseir, and W. Burleson. Adaptive system on a chip (asoc) : a backbone for power-aware signal processing cores. In *IEEE Proceedings of the International Conference on Image Processing*, volume 3, page 105Ű108, Sept 2003.

[96] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch. The nostrum backbone-a communication protocol stack for networks on chip. In *IEEE 17th International Conference on VLSI Design, 2004. Proceedings*, page 693Ű696, 2004.

[97] F. Karim, A. Nguyen, and S. Dey. An interconnect architecture for networking systems on chips. In *Micro, IEEE*, volume 22, pages 36–45, 2002.

[98] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. Interconnection networks enable fine-grain dynamic multi-tasking on fpgas. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 795–805, 2002.

[99] T. A. Bartic, J.Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Highly scalable network on chip for reconfigurable systems. In *IEEE proceedings International Symposium onSystem-on-Chip*, pages 79–82, Nov 2003.

[100] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip : concepts, architectures, and implementations. In *IEEE proceedings International Symposium onSystem-on-Chip*, volume 22, pages 414–421, Oct 2005.

[101] E. Rijpkema, K. Goossens, and P. AWielage. Router architecture for networks on silicon. In *2nd Workshop on Embedded Systems (PROGRESS)*, pages 181–188, Nov 2001.

[102] M.A Forsell. Scalable high-performance computing solution for networks on chips. In *IEEE Micro*, volume 22, pages 46–55, Sep 2002.

[103] I. Saastamoinen, M. Alho, and J. Nurmi. Buffer implementation for proteo networks-on-chip. In *IEEE International Symposium on Circuits and Systems*, volume 2, pages 113–116, May 2003.

[104] I. Saastamoinen, M. Alho, J.Pirttimäki, and J. Nurmi. Proteo interconnect ips for networks-on-chip. In *IP Based SoC Design*, Oct 2002.

[105] D. Sigüenza-Tortosa and J. Nurmi. Proteo : A new approach to network-on-chip. In *IASTED International Conference on Communication Systems and Networks*, Sep 2002.

[106] C. Zeferino and A. Susin. Socin : A parametric and scalable network-on-chip. In *16th Symposium on Integrated Circuits and Systems Design*, pages 169–174, Sep 2003.

[107] D. Wiklund and D. Liu. Socbus : Switched network on chip for hard real time systems. In *International Parallel and Distributed Processing Symposium*, Apr 2003.

[108] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. Qnoc : Qos architecture and design process for network on chip. In *Euromicro : The Journal of Systems Architecture, Special Issue on Networks on Chip*, volume 50, pages 105 – 128, Apr 2004.

[109] M.Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. xpipes : a latency insensitive parameterized network-on-chip architecture for multi-processor socs. In *ICCD '03 : Proceedings of the 21st International Conference on Computer Design*, pages 536–539, Washington, DC, USA, 2003. IEEE Computer Society.

[110] Fernando Moraes, Ney Calazans, Aline Mello, Leandro M&#246 ;ller, and Luciano Ost. Hermes : an infrastructure for low area overhead packet-switching networks on chip. *Integr. VLSI J.*, 38(1) :69–93, 2004.

[111] A. Mello, L. Tedesco, N. Calazans, and F. Moraes. Virtual channels in networks on chip : implementation and evaluation on hermes noc. In *SBCCI '05 : Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 178–183. ACM Press, 2005.

[112] T. Bjerregaard, S. Mahadevan, R.G. Olsen, and J. Sparso. An ocp compliant network adapter for gals-based soc design using the mango network-on-chip. In *International Symposium on System-on-Chip*, pages 171–174, Nov 2005.

[113] Y. Durand, C. Bernard, and D. Lattard. Faust : On-chip distributed soc architecture for a 4g baseband modem chipset. In *Proceedings Design and Reuse IP-SOC*, pages 51–55, Dec 2005.

[114] N. Shah, W. Plishker, and K. Keutzer. Np-click : A programming model for the intel ixp1200. In *2nd Workshop on Network Processors (NP-2) at the 9th International Symposium on High Performance Computer Architecture (HPCA-9) Anaheim CA*, February 2003.

[115] F. Cappello and D. Etiemble. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. *Supercomputing SC2000*, November 2000.

[116] S.K. Shukla and Mi. Theobald. Special issue on formal methods for globally asynchronous and locally synchronous (gals) systems. *Form. Methods Syst. Des.*, 28(2) :91–92, 2006.

[117] Message Passing Interface Forum. Mpi. Available on : `http://www.mpi-forum.org/`.

[118] ModelSim. Modelsim 6.0a se. Available on : `http://www.model.com/`.

[119] F.Ghenassia. *Transaction-Level Modeling with SystemC TLM Concepts and Applications for Embedded Systems*. Springer, 2005.

[120] F.Fummi, S.Martini, G.Perbellini, and M.Poncino. Iss-systemc integration for the co-simulation of multi-processor soc. In *DATE*, 2004.

[121] K.Ghali and O.Hammami. Embedded processor characteristics specification through multiobjective evolutionary algorithms. In *IEEE International Symposium on Industrial Electronics*, volume 2, pages 907–912, June 2003.

[122] K. Ghali and O. Hammami. Embedded processors optimization with hardware in the loop. In *in Proceedings of the IEEE International Symposium on Industrial Electronics*, page 561Ű564, May 2004.

[123] S. Fei, S. Ravi, A. Raghunathan, and N.K. Jha. Application-specific heterogeneous multiprocessor synthesis using extensible processors. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 25, pages 1589–1602, Sept 2006.

[124] Neal K. Bambha and Shuvra S. Bhattacharyya. Joint application mapping/interconnect synthesis techniques for embedded chip-scale multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 16(2), Feb 2005.

[125] S. Pasricha and N. Dutt. Cosmeca : Application specific co-synthesis of memory and communication architectures for mpsoc. In *Proceedings Design, Automation and Test in Europe*, volume 1, pages 1–6, Mar 2006.

[126] D.Lyonnard, S.Yoo, A.Baghdadi, and A.A.Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *DAC*, pages 518–523, 2001.

[127] F. Sun, N.K.Jha, S.Ravi, and A.Raghunathan. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *In 18th International Conference on VLSI Design*, pages 551–556, 2005.

[128] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer. An automated exploration framework for fpga-based soft multiprocessor systems. In *International Conference on Hardware/Software Codesign and System Synthesis(CODES-05)*, September 2005.

[129] Roger M.A. Peel and Barry M. Cook. Occam on field-programmable gate arrays - fast prototyping of parallel embedded systems. In *the Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2000.

[130] May David and Richard Taylor. Occam : An overview. In *Microprocessors and Microsystems*, volume 8, pages 73–79, March 1984.

[131] Muhammed Al-Mulhem and Shahid Ali. Formal semantics of visual occam. In *Computer Languages*, volume 24, pages 99–113, 1998.

[132] R.Benmouhoub and O.Hammami. Mocsoc : Multiprocessor on chip synthesis from occam. In *The 13th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, Apr 3-4 2006.

[133] Stephen C. Johnson. The lex yacc page. Available on : `http://dinosaur.compilertools.net/`.

[134] C.Ciordas, K. Goossens, A. Radulescu, and T. Basten. Noc monitoring : impact on the design flow. In *IEEE International Symposium on Circuits and Systems*, May 2006.

[135] Calin Ciordas, Twan Basten, Andrei Radulescu, Kees Goossens, and Jef Van Meerbergen. An event-based monitoring service for networks on chip. *ACM Trans. Des. Autom. Electron. Syst.*, 10(4) :702–723, 2005.

[136] C. Ciordas, A. Hansson, K. Goossens, and T. Basten. A monitoring-aware network-on-chip design flow. In *IEEE 9th EUROMICRO Conference on Digital System Design : Architectures, Methods and Tools*, pages 97–106, Aug 2006.

[137] Leonel Tedesco, Aline Melloand Diego Garibotti, Ney Calazans, and Fernando Moraes. Traffic generation and performance evaluation for mesh-based nocs. In *18th ACM Symposium on Integrated Circuits and Systems Design*, pages 184–189, 2005.

[138] P.P. Pande, C. Grecu, M.Jones, A. Ivanov, and R. Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. In *IEEE Transactions on computers*, volume 54, pages 1025 – 1040, Aug 2005.

[139] A. Hegedus, G.M. Maggio, and L.Kocarev. A ns-2 simulator utilizing chaotic maps for network-on-chip traffic analysis. In *IEEE International Symposium on Circuits and Systems, ISCAS.*, volume 4, pages 3375–3378, May 2005.

[140] Mohammed El Shobaki. On-chip monitoring of single- and multiprocessor hardware real-time operating systems. In *Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, March 2002.

[141] N. Genko, D. Atienza, G. De Micheli, L. Benini, J.M. Mendias, R. Hermida, and F. Catthoor. A novel approach for network on chip emulation. In *Circuits and Systems, 2005. ISCAS 2005. IEEE*, volume 3, pages 2365–2368, May 2005.

[142] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. L. Sangiovanni-Vincentelli. System level design : Orthogonalization of concerns and platform–based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, December 2000.

[143] R.Benmouhoub and O.Hammami. Mocdex : Multiprocessor on chip multiobjective design space exploration with direct execution. In *The 13th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, Apr 3-4 2006.

[144] Xilinx. Ug012 : " virtex-ii pro and virtex-ii pro x fpga user guide " (v4.0) 23 march 2005. Available on : `http://www.xilinx.com/bvdocs/userguides/ug012.pdf`.

[145] Xilinx. Xapp 290 (v1.2) : Two flows for partial reconfiguration : Module based or difference based. 09/09/2004. Available on : `http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf`.

[146] H.Mrabet, Z. Marrakchi, P. Souillot, and H. Mehrez. Performance improvement of fpga using novel multilevel hierarchical interconnection structure. *ICCAD*, pages 675–679, 2006.

Annexe 1

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2006, Article ID 54074, Pages 1–14
DOI 10.1155/ES/2006/54074

MOCDEX: Multiprocessor on Chip Multiobjective Design Space Exploration with Direct Execution

Riad Ben Mouhoub and Omar Hammami

UEI, ENSTA 32, Boulevard Victor, 75739 Paris, France

Received 15 December 2005; Revised 5 May 2006; Accepted 2 June 2006

Fully integrated system level design space exploration methodologies are essential to guarantee efficiency of future large scale system on programmable chip. Each design step in the design flow from system architecture to place and route represents an optimization problem. So far, different tools (computer architecture, design automation) are used to address each problem separately with at best estimation techniques from one level to another. This approach ignores the various and very diverse vertical relations between distinct levels parameters and provides at best local optimization solutions at each step. Due to the large scale of SoC, system level design methodologies need to tackle the system design process as a global optimization problem by fully integrating physical design in the design space exploration. We propose MOCDEX, a multiobjective design space exploration methodology, for multiprocessor on chip which closes the gap between these associated tools in a fully integrated approach and with hardware in the loop. A case study of a 4-way multiprocessor demonstrates the validity of our approach.

Copyright © 2006 R. B. Mouhoub and O. Hammami. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

System on chip are increasingly becoming complex to design, test, and fabricate. SoC design methodologies make intensive use of intellectual properties (IPs) [1] to reduce the design cycle time and meet stringent time to market constraints. However, associated tools still lag behind when addressing the huge associated design space exposed by the combination of soft IP. In addition, failure to meet an efficient distribution in terms of performance, area, and energy consumption makes the whole design inappropriate. Although this problem is already hard to solve in the ASIC domain, it is exacerbated in the system on programmable chip (SoPC) domain. SoPC are large scale devices offering abundant resources but in fixed amount and in fixed location on chip. Implementing embedded multiprocessors on these devices presents several advantages, the most important is to be able to quickly evaluate various configurations and tune them accordingly. Indeed, embedded multiprocessor design is highly application-driven and it is therefore highly advantageous to execute applications on real prototypes. However, due to the fact that specific resources are located at fixed positions on these large chips it is hard not to take into account the important impact of place and route results on the critical paths and therefore on the overall performance. In this paper, we address this multiobjective optimization problem [2] restricted to performance and area through the combination of an efficient design space exploration (DSE) technique coupled with direct execution on an FPGA board [3]. The direct execution removes the prohibitive simulation time associated with the evaluation of embedded multiprocessor systems. A side effect of this approach is that direct execution requires actual on chip implementation of the various multiprocessor configurations to be explored which provides actual post synthesis and place and route area information. The resulting flow is fully integrated from multiprocessor platform specification to execution.

The paper is organized as follows. In Section 2, we review previous work. Section 3 describes an example of soft IP-based multiprocessor and the breadth of the problem associated with the design of such multiprocessor on a particular instance of embedded memories optimization. Section 4 presents our approach, MOCDEX, based on multiobjective evolutionary algorithms (EA) and direct execution. In Section 5 we describe a case study and validation, while Section 6 provides exploration results. Section 7 provides statistical insight in the explored design space and demonstrates the diversity of multiprocessor configurations explored during the automatic process. Finally, we conclude in Section 8 with remarks and directions for future work.

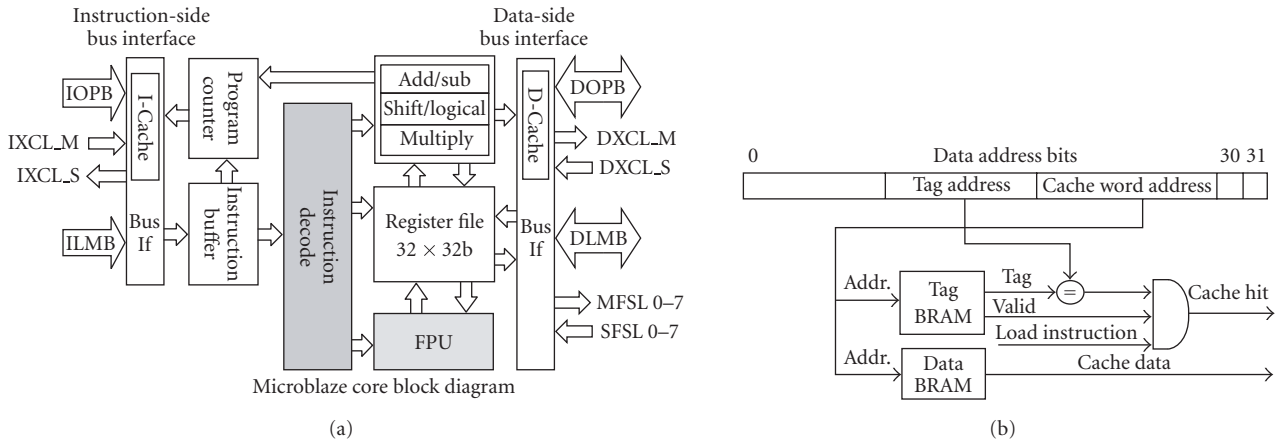


Figure 1: (a) MicroBlaze soft IP processor. (b) MicroBlaze processor cache organization.

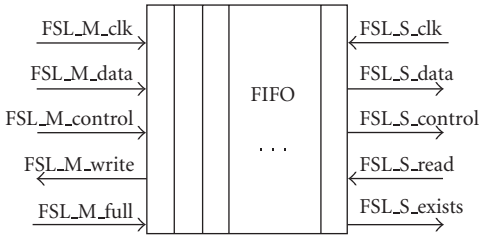


Figure 2: Fast simplex link.

2. PREVIOUS WORK

The recent emergence of multiprocessors on chip as strong potential candidates to address performance, energy, and area constraints for embedded applications has resulted in the following question: how do we design efficient multiprocessors on chip for a target application? Design automation tools fail to address this question, while traditional parallel computer architectures techniques [4] have not been exposed to the huge diversity brought by soft IP-based design methodologies and the strong constraints of embedded systems [5]. Therefore, the design of multiprocessor on chip is the convergence focus of previously unrelated techniques and as such represents a new problem on how to establish a close integration between those techniques. It is then not surprising that few works so far have been devoted to design methodologies for multiprocessors on chip. In [6] they present a design flow for the generation of application-specific multiprocessor architectures. In the flow, architectural parameters are first extracted from a high-level specification and are used to instantiate architectural components such as processors, memory modules, and communication networks. Cycle accurate cosimulations of the architectures are used for performance evaluation while all results in our case are obtained through actual execution and they do not use design space exploration algorithm. In [7], synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors is proposed based on an iterative improvement algorithm implemented in the context of a commercial design flow. The proposed algorithm is based on cycle count estimation and instruction-set simulations, and although synthesis results are used, both architecture and implementation flows are still decoupled. In [8] they propose an automated exploration framework for FPGA-based soft multiprocessor systems. Using as input the application graph that describes tasks and communication links, outputs of the exploration step are a microarchitecture configuration of processors and communication channels, a mapping of the application tasks and links onto the processors and channels of the micro-architecture. They formulate the exploration problem as an integer linear problem. The "best design" based on the ILP results is selected and synthesized to verify performance. This verification may fail because routing details are not taken into account during the exploration process. This approach still keeps decoupled design automation tools and exploration, while in our approach design space exploration fully integrates design automation tools since solutions are ranked on the area results obtained post-synthesis and place and route and performance results obtained from actual execution on board. Besides, the problem formulation ignores the arbitration overhead when computing the communication access time again due to the static nature of the design space exploration decoupled from actual execution. As pointed out by the authors, this can lead to a significant source of errors when there are a large number of masters on the bus. Finally, it should be clear that no single "best design" exists in any multiobjective optimization problem and only a Pareto set can be obtained. In [9] they present high-level scheduling and interconnect topology synthesis techniques for embedded multiprocessor system-on-chip that are streamlined for one or more digital signal processing applications. The proposed interconnect synthesis method utilizes a genetic algorithm (GA) operating in conjunction with a list scheduling algorithm which produces candidate topology graphs based on direct physical communication. The proposed algorithm

is a single objective algorithm, while the algorithm used in our work is a multiobjective algorithm; and although we use direct link we optimize also buffering capacities by trading on-chip memory among embedded processor cache memories and connection link buffers. To the best of our knowledge our work is the first to fully integrate and therefore close the gap between design automation tools and architecture design space exploration technique in a multiobjective constraints paradigm with actual execution for all multiprocessor on chip configurations explored during the design space exploration process.

3. SOFT IP-BASED EMBEDDED MULTIPROCESSOR SYSTEMS

Soft IP-based embedded multiprocessor systems are SoC fully designed with soft IPs. This includes soft IP processors, interconnect infrastructure and memories. An example of such soft IP multiprocessor is described below based on Xilinx EDK IPs [10].

3.1. MicroBlaze soft IP processor

MicroBlaze soft IP [11] is a 32-bit 3-stage single issue pipelined Harvard style embedded processor architecture provided by Xilinx as part of their embedded design tool kit.

Both caches are direct mapped, with 4-word cache lines allowing configurable cache and tag size and user selectable cacheable memory area. Data cache uses a write-through policy. MicroBlaze core configurability extends to functional unit through user selectable barrel shifter (BS), hardware multiplier (HWM), hardware divider (HWD), and floating point unit (FPU). MicroBlaze has neither static nor dynamic branch prediction unit and supports branches with delay slots. For its communication purposes, MicroBlaze uses either a bus or a direct link. The on-chip peripheral bus (OPB) is part of IBM CoreConnect bus architecture and allows the design of complete single processor systems with peripherals and uses designed hardware accelerators [12, 13]. However, even for a simple embedded-processor-based multiprocessors designs such as MicroBlaze, the OPB bus is not suitable because of its lack of scalability. Another approach is provided by "Fast Simplex Link" [14] which allows direct connection between embedded processors through FIFO channels.

3.2. MicroBlaze fast simplex link

The fast simplex link (FSL) [14] is an IP developed by Xilinx to achieve a fast unidirectional point-to-point communication between any two components. The FSL link is implemented as a 32-bit wide FIFO with configurable depth and width option. The FSL can be either a master or a slave interface depending upon its use.

MicroBlaze soft embedded processor allows up to 8 master and slave FSL interfaces. Basic software drivers are provided to simplify the use of FSL connection. They consist of read/write routines and control functions. The read/write routines can be executed in two different ways: blocking and nonblocking mechanism.

3.3. IBM interconnect

The IBM interconnect [10] represents a set of IPs used to develop SoC devices. It includes the PLB and OPB bus, a PLB-OPB bridge, and various peripherals.

3.4. MPSoC platform description

Our FPGA multiprocessor platform consists of four MicroBlaze processors with instruction and data cache units. These processors are connected with each other through FSL channels.

Each MicroBlaze is connected, as shown in Figure 3, to an OPB bus to use a timer and an interrupt controller for threads and OS execution. MicroBlaze MB0 is connected to the OPB bus which is connected to the PCI interface of the host (WS). This allows the designer to send and receive data from the host to the multiprocessor system. We implemented a soft layer of communication in each MicroBlaze which performs send and receive functions of packets. The packets consist of headers representing the destination and source addresses and the number of flits in the payload. A wormhole routing algorithm was used since it uses less memory, making it suitable for network on chip communication. As it can be seen a 4-way multiprocessor has been built based on the previously described soft IPs.

The implementation of such a soft IP multiprocessor on FPGA platform requires a variable amount of resources as each soft IP composing the multiprocessor requires a variable amount of resources depending on the configuration options [10]. Table 1 provides an insight on such variability.

Such a soft IP multiprocessor can be easily adapted to the need of a specific application adapted to a particular application. However, these systems for best efficiency and low memory latency require the use of embedded on chip memories. Unfortunately, embedded memories are scarce resources for which processors instruction and data cache memories as well as bus and network on-chip FIFO-based interfaces will compete. This competition is dominated by the absolute requirement of efficiency in performance, area, and energy consumption [5]. If we focus on cache and FSL configurability, we have for each cache memory 7 possible configurations and for the FSL 11 possible configurations. The design space associated with those parameters (74×118, thus 514 675 673 281 different configurations) requires 16 321 years of simulation for 1 minute simulation per configuration.

4. MOCDEX MULTIOBJECTIVE DESIGN SPACE EXPLORATION

4.1. Problem formulation

The design challenge represented by soft IP-based multiprocessor design is a multiobjective optimization problem [2].

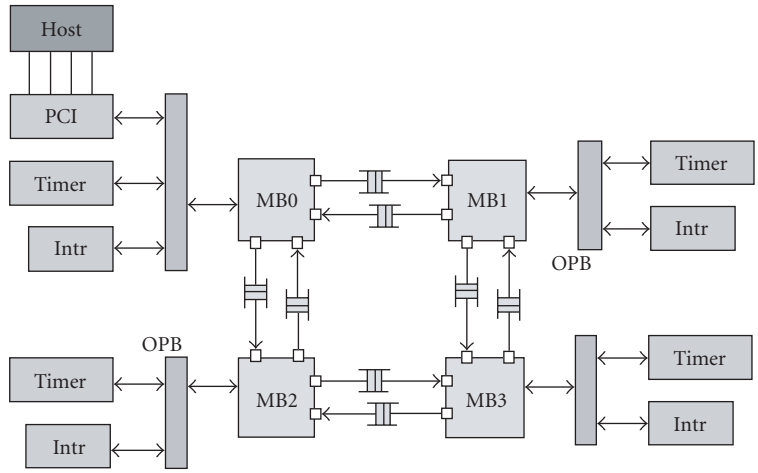


Figure 3: Mesh platform 2×2.

Table 1: Multiprocessor soft IP resources variation.

Soft IP	Slices Min Max	FF Min Max	BRAM Min Max	Parameters	Soft IP	Slices Min Max	FF Min Max	BRAM Min Max	Parameters
MicroBlaze	731 var	552 var	0 var	Cache sizes 1 K, 2 K, 4 K, 8 K, 16 K, 32 K, 64 K	OPB	46 410	5 121	N/A N/A	Data bus width, address bus width, arbiter
					OPB PCI	340 3025	445 2105	0 2+	Interface/DMA parameters
FSL width/depth	21 451	36 34	0 17	FIFO sizes 8, 16, 32, 64, 128, 256, 512, 1 K, 2 K, 4 K, 8 K	OPB timer	99 200	105 266	0	Timer counter widths
					OPB intr ctr	54 307	63 342	0	Number of interrupt inputs

The multiobjective optimization problem is the problem of simultaneously minimizing the n components (e.g., area, number of execution cycles, energy consumption), f_k, $k = 1, \ldots, n$, of a possibly nonlinear function f of a general decision variable x in a universe U, where

$$f(x) = (f_1(x), f_2(x), \ldots, f_n(x)). \tag{1}$$

The problem has usually no unique optimal solution but a set of nondominated alternative solutions known as the Pareto-optimal set. The dominance is defined as follows.

Definition 1 (Pareto dominance). A given vector $u = (u_1, u_2, \ldots, u_n)$ is said to dominate $v = (v_1, \ldots, v_n)$ if and only if u is partially less than v ($u_p < v$), that is,

$$\forall i \in \{1, \ldots, n\}, \quad u_i \leq v_i, \quad \exists i \in \{1, \ldots, n\} : u_i < v_i. \tag{2}$$

The Pareto optimality definition derives from the Pareto dominance.

Definition 2 (Pareto optimality). A solution $x_u \in U$ is said to be Pareto optimal if and only if there is no $x_v \in U$ for which $v = f(x_v) = (v_1, \ldots, v_n)$ *dominates* $u = f(x_u) = (u_1, \ldots, u_n)$.

Pareto-optimal solutions are also called efficient, nondominated, and noninferior solutions. The corresponding objective vectors are simply called nondominated. The set of all nondominated vectors is known as the nondominated set or the Pareto set (also Pareto-optimal set or Pareto-optimal front). This Pareto set can be seen as the tradeoff surface of the problem. The solution of a practical problem such as multiprocessor system on chip (MPSoC) design may be constrained by a number of restrictions imposed on a decision variable. Constraints may express the domain of definition of the objective function or alternatively impose further restrictions on the solution of the problem according to knowledge at a higher level. In the general case of system on programmable chip, the amount of on chip memory for example is fixed and represents a clear and stringent constraint. The constrained optimization problem is that of minimizing a multiobjective function $(f_1, \ldots, f_k)$ of some generic decision

variable x in a universe U subject to a positive number $n - k$ of conditions involving x and eventually expressed as a functional vector inequality of the type

$$(f_{k+1}(x), \ldots, f_n(x)) < (g_{k+1}, \ldots, g_n), \quad (3)$$

where the inequality applies component-wise. It is implicitly assumed that there is at least one point in U which satisfies all constraints although in practice that cannot always be guaranteed.

The case study of multiobjective optimization we will address in this paper is the minimization of area (BRAM $f1$ and slices resources $f2$) and execution time (number of cycles $f3$) representing a 3-objectives multiobjective problem.

4.2. Multiobjective optimization and multiobjective evolutionary algorithms (MOEA)

Multiobjective optimization have not been addressed properly by traditional optimization techniques (gradient based, simulated annealing, linear programing) since most of these techniques are mono-objective. Extending these techniques through approaches using aggregation functions does not represent true multiobjective optimization and does not produce multiple solutions. Multiobjective evolutionary algorithms (MOEA) are more appropriate to solve optimization problems with concurrent conflicting objectives and are particularly suited for producing Pareto-optimal solutions. Several Pareto-based evolutionary algorithms have been proposed during the last decade, SPEA-2, PESA, and NSGA-II, [2, 15] to solve multicriteria optimization problems. The NSGA-II [16] is an MOEA considered to outperform other MOEA [17] and is briefly presented below.

Individuals classification

Initially, before carrying out the selection, one assigns to each individual in the population a row *rank* (by using the Pareto set). All the nondominated individuals of the same row are classified in a category. To this category, we assign effectiveness, which is inversely proportional to the order of Pareto set. Figure 4 presents an example of classification in Pareto sets.

Main loop of algorithm NSGA-II [16]

Initially, a random parent population $P0$ is created. Each individual of this population is affected to an adequate Pareto rank. From the population $P0$, we apply the genetics operators (selection, mutation, and crossover) to generate the population child $Q0$ of size N. The elitism is ensured by the comparison between the current population P_t and the preceding population P_{t-1}. The NSGA-II procedure follows (see Algorithm 1).

The NSGA-II algorithm runs in time $O(GN \log^{M-1} N)$, where G is the number of generations, M is the number of objectives, and N is the population size [17]. In addition, our previous experience on multiobjective optimization of soft IP embedded processor [18, 19] emphasizes this choice.

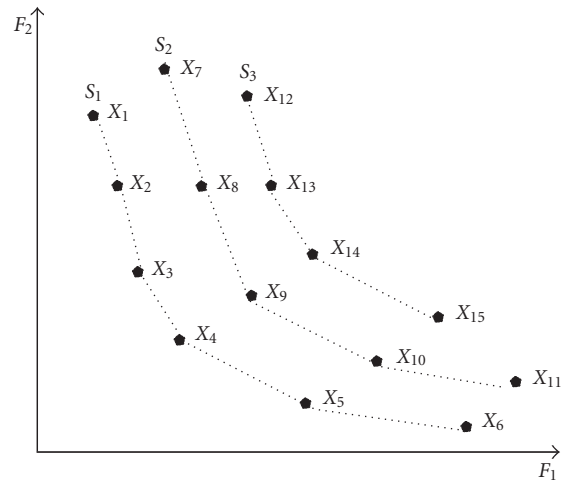


Figure 4: Classification of the individuals in several fronts according to the Pareto rank (list of Pareto sets).

$R_t = P_t U Q_t$ # combine parent and children population
$F =$ fast-nondominated-sort (R_t) # F all nondominated fronts sets
$P_{t+1} = \emptyset$ and $i = 1$ # initialization
until $|P_{t+1}| + |F_i| \leq N$ # till parent pop is filled
  Crowding-distance-assignment (F_i) # compute distance in Fi
  $P_{t+1} = P_{t+1} U F_i$ # include ith nondominated front in the parent pop
  $i = i + 1$ # check the next front for inclusion
Sort $(F_i, <_n)$ # Sort in descending order using $<_n$
$P_{t+1} = P_{t+1} U F[1 : (N - |P_{t+1}|)]$ # Choose the first $(N - |P_{t+1}|)$ elements
$Q_{t+1} =$ make-new-pop (P_{t+1}) # apply genetic operators to create new pop Q_{t+1}
$T = t + 1$ # increment to next generation

Algorithm 1: NSGA-II.

4.3. MOCDEX

It is clear that MOEAs such as NSGA-II requires the evaluation of individuals (MPSoC configurations) with regard to the 3 objectives considered, BRAM, slices and number of cycles Although, BRAM and slices, could be estimated, we advocate the full use of design automation tools including place and route to access this information. Indeed, for complex systems on large platform FPGA place and route impact cannot be overlooked and can hardly be estimated with sufficient accuracy to be used in an automatic multiobjective design space exploration tool. The execution time of multiprocessor on chip can be obtained through simulation either at RTL level which would be prohibitive for large design space exploration without massive use of computing resources (compute farms) or at TLM level (SystemC) as often advocated [20, 21].

However although SystemC level simulation has been regularly proved to outperform RTL VHDL level simulation, it does not outperform actual execution on FPGA. We argue that for large scale MPSOC, FPGA platform represents an opportunity to both reduce simulation time through actual execution and increase the design space exploration through this reduction of the evaluation of each MPSOC configuration. Our proposal follows.

MOCDEX (general)

(1) Generate random population of MPSOC configurations within soft IP parameters constraints.
(2) For all configurations,
  (a) generate hardware/software platform specification files,
  (b) generate through system EDA and IPs HW/SW model of the MPSOC,
  (c) synthesize/place and route MPSOC configuration using EDA tools,
  (d) record place and route reports,
  (e) download configuration file on FPGA platform,
  (f) execute MPSOC configuration and record execution clock cycles,
  (g) rank the solution.
(3) Generate new population using MOEA algorithm.
(4) Is the Pareto front satisfactory or the number of generations reached if no goto 3?
(5) Final Pareto front MPSOC configurations are available for selection.

As shown in Figure 5, both the DSE and physical design are executed on a host PC while the execution is achieved on a PCI-based FPGA platform which communicates execution results to the host.

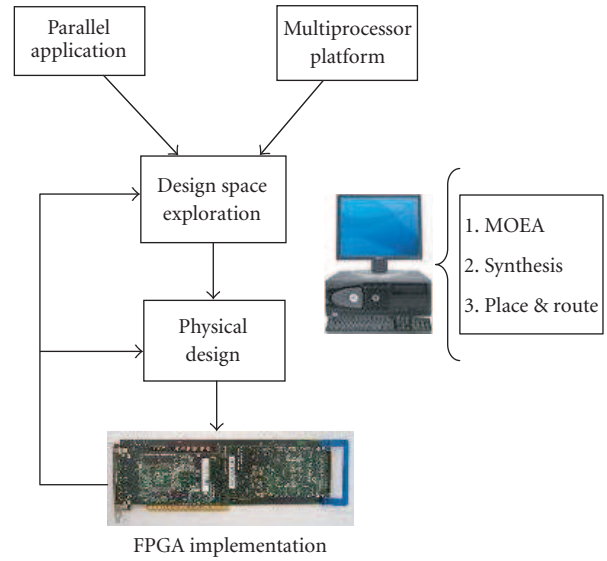


Figure 5: MOCDEX MPSOC exploration flow.

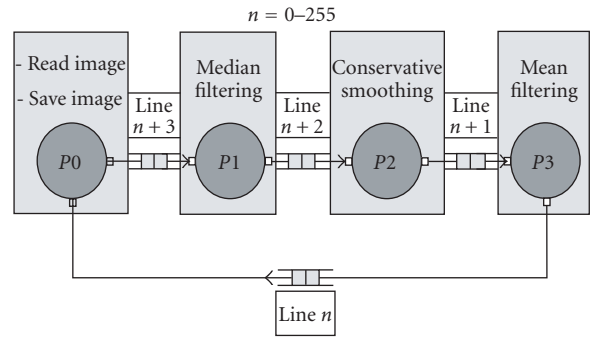


Figure 6: Image filtering application multiprocessor platform distribution.

5. CASE STUDY AND VALIDATION

The previously described design flow has been applied in the framework of Xilinx FPGA platforms.

5.1. Image filtering application

A design of four Xilinx MicroBlaze processors, communicating with eight FSL channels in a mesh topology and executing image filtering algorithms, was implemented at 100 MHz. This application was chosen because it requires extensive data processing and data communication among the filters for a good and fast testing of our exploration framework.

Figure 6 shows our filtering methodology. As we can see, the execution is achieved in a pipelined way where image lines are sent from a processor to another as soon as the previous processor has finished its work on it. Obviously, this type of execution makes us save a significant amount of time and memory which are often the major constraints for embedded systems in general and for our platform in particular. Indeed, performing this task in a pipelined way allows us to have a maximum of three image lines stored in the associated processor's memory rather than the whole image. The rest of the image lines will enter the FIFOs (FSLs) of their respective processors one by one. The processor $P0$ in Figure 6 receives image data from the host computer through the PCI bus. Once it receives the data it immediately sends it to the next processor which is $P1$. $P1$ performs a median filtering which results in noise reduction from the image. It is performed on a 3-by-3 pixel window where the center pixel value is replaced by the median of the neighboring pixel values. This value is obtained by sorting the pixels based on their numerical values and then replacing the pixel to be processed by the middle value. The processor $P2$ fetches the line coming from $P1$ and performs a conservative smoothing on it which is an operation that preserves the high spatial frequency details. Finally, the third processor $P3$ performs a mean filtering which consists of very simple method used for noise reduction where the pixel to be processed is replaced by the average

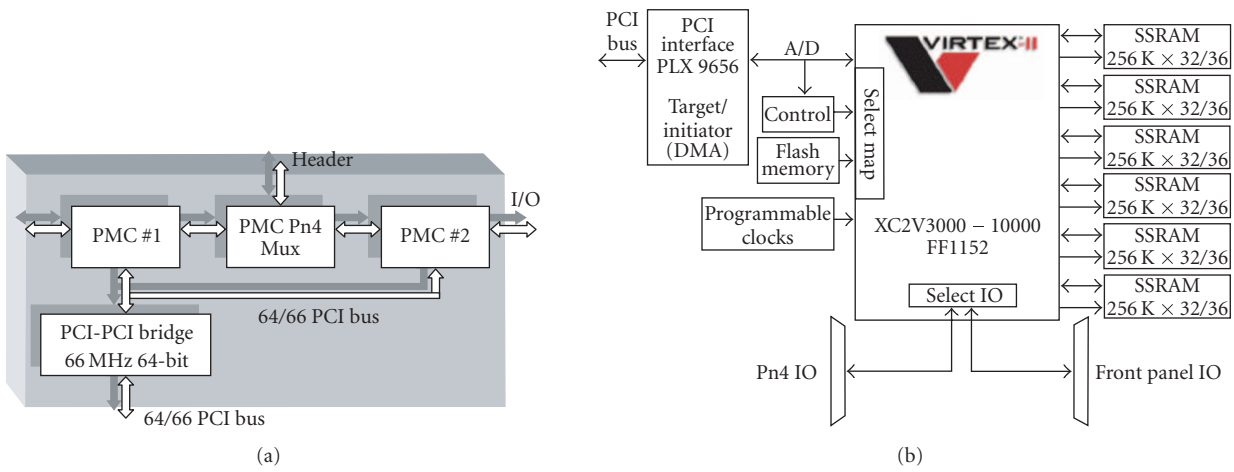


Figure 7: Alpha-data ADM-XRC-II and ADC-PMC boards.

Table 2: Multiprocessor on chip design space.

Procs	FSL1Out	FSL2Out	D-Cache	I-Cache
MB0	16...2048	16...2048	512...4096	512...4096
MB1	16...2048	16...2048	512...4096	512...4096
MB2	16...2048	16...2048	512...4096	512...4096
MB3	16...2048	16...2048	512...4096	512...4096

Table 3: Xilinx virtex-II XC2V 8000 resources.

XC2V8000	Values
Slices	46 952
BRAM (18 Kbits)	168
18 × 18 multipliers	168
DCM	12
Max. Dist RAM Kb	1456

value of its neighbors. Due to the different amount of computations required by each filter, it results in different workload for each processor. Thus the execution time for each algorithm differs and hence involves an unequal FIFOs occupancy. Therefore, the application used has to be naturally unbalanced to thoroughly analyze the problem. The problem at hand is to optimally distribute the limited on chip embedded memory among the embedded processors cache memories (instruction, data) and the communication FIFOs while optimizing execution time and area. The design space for this problem is specified in Table 2.

The possible number of different configurations is given by the product of the number of distinct configurations for each configurable architectural parameter. Each cache memory may have up to 4 different sizes and each FIFO up to 8 different sizes. The total design space represents $(4 \times 4 \times 8 \times 8)^4 = 2^{40}$ configurations. If each configuration evaluation would require 1 second, the total evaluation time would be 34 865 years of evaluation. Clearly an exhaustive evaluation technique is unfeasible and multiobjective optimization techniques are able to efficiently prune this design space while simulation is clearly outperformed by direct execution on large scale FPGA devices.

5.2. Alpha-data environment

For the implementation of MOCDEX we used the alpha-data hardware and software environment.

5.2.1. Alpha data hardware environment

The alpha-data hardware environment described in Figure 7 is composed by (1) the ADC-PMC and (2) the ADM-XRC-II. The ADC-PMC is a dual PMC adapter for PCI. It supports 64-bit 66 MHz primary and secondary PCI via an Intel 21154 PCI-PCI bridge device. The ADM-XRC-II is a high performance reconfigurable PMC (PCI mezzanine card) based on the Xilinx Virtex-II range of platform FPGAs. Features include high-speed PCI interface, external memory, high-density I/O, programmable clocks, temperature monitoring, battery backed encryption, and flash boot facilities.

On board clock generator provides a synchronous local bus clock for the PCI interface and the Xilinx Virtex-II FPGA. A second clock is provided to the Xilinx Virtex-II FPGA for user applications and can be free running or stepped under software control. Both clocks are programmable and can be used by the Virtex clock. The user clock has a maximum value of 100 MHz. The ADM-XRC-II uses a Xilinx XC2V8000-6 FF1152 device [22] whose characteristics are described Table 3.

5.2.2. Alpha-data software environment

The ADM-XRC SDK is a set of resources including an application-programing interface (API) intended to assist the user in creating an application using one of Alpha-data's ADM-XRC range of reconfigurable coprocessors. The API

Table 4: ADM XRC SDK API functions.

Group	Application
Initialization	ADMXRC2_CloseCard ADMXRC2_OpenCard ADMXRC2_OpenCardByIndex ADMXRC2_SetSpaceConfig
FPGA configuration through PCI	ADMXRC2_ConfigureFromBuffer ADMXRC2_ConfigureFromBufferDMA ADMXRC2_ConfigureFromFile ADMXRC2_ConfigureFromFileDMA ADMXRC2_LoadBitstream ADMXRC2_UnloadBitstream
Data transfer PC = FPGA board	ADMXRC2_BuildDMAModeWord ADMXRC2_DoDMA ADMXRC2_DoDMAImmediate ADMXRC2_MapDirectMaster ADMXRC2_Read ADMXRC2_ReadConfig ADMXRC2_SetupDMA ADMXRC2_SyncDirectMaster ADMXRC2_UnsetupDMA ADMXRC2_Write ADMXRC2_WriteConfig
Interrupt handling	ADMXRC2_RegisterInterruptEvent ADMXRC2_UnregisterInterruptEvent

makes use of a device driver that is normally not directly accessed by the user's application. The API library described in Table 4 takes care of open, close, and device I/O control calls to the driver. The ADM-XRC SDK is designed to be thread-safe. Table 4 describes the main API functions which allow initializing the board, configuring the FPGA though the PCI bus, and transfering data between the FPGA and the host computer and the interrupt handling.

Clearly since MOCDEX explore the design space by implementing on FPGA new multiprocessor configurations the FPGA is reconfigured through the PCI bus from the main program by executing the ADM-XRC SDK FPGA reconfiguration API using the bitfile generated from EDK synthesis and place and route. Resulting execution number of cycles are provided as well through the PCI bus to the host using ADM-XRC SDK data transfer API.

5.3. Xilinx EDK tools

The embedded development kit (EDK) bundle is an integrated software solution for designing embedded processing systems.

Table 5 and Figure 8 describe the use of each configuration file in the process of hardware platform generation, software platform generation, and software application and creation.

The MHS file defines the system architecture, peripherals, and embedded processors. It also defines the connectivity of the system, the address map of each peripheral in the system, and configurable options for each peripheral. The MHS file can be defined through XPS Gui wizards. However for the time being Xilinx wizards do not allow the design of multiprocessors platforms and therefore they should be defined directly in the MHS file. It is clear that in the purpose of design space exploration of multiprocessor architecture the MHS file is the prime target of modifications. Changing parameters value in the MHS file generates a new multiprocessor configuration and invoking the XPS tool in no window mode from a main program allows the generation of the multiprocessor netlist. Table 6 provides examples of MHS file parts.

5.4. Exploration flow description

The proposed automatic design flow described in Figure 5 can be applied in the framework of Xilinx EDA tools and the Alpha-data environment. The flow is mainly composed of 3 parts: (1) architecture design space exploration engine (DSE), (2) physical design, and (3) FPGA platform PCI board. The architecture design space exploration part controls the whole flow and runs on a host PC. First based on the user specified design space parameters and parameters range, the DSE specifies the architectural parameters of the multiprocessors configurations to be evaluated then translates those parameters into platform EDA design tool input file specifications. In our case,

(1) *MOCDEX for Xilinx FPGA platform*,
(2) generate random population of MPSoC configurations (caches and FSL variations),
(3) for all configurations,
    (a) generate hardware/software platform specification files (mhs, mpd, pao, mss, mld, mdd, files),
    (b) generate through Xilinx system XPS and Xilinx IPs HW/SW model of the MPSOC,
    (c) synthesize/place and route MPSOC configuration using Xilinx ISE 6.3,
    (d) record place and route reports generated from Xilinx ISE 6.3,
    (e) download configuration file on FPGA Alpha-data platform using ADM-XRC SDK API,
    (f) execute MPSOC configuration and record execution clock cycles using ADM-XRC SDK API,
    (g) rank the solution,
(4) generate new population using NSGA-II algorithm,
(5) is the Pareto front satisfactory or the number of generations reached if no goto 3?
(6) final Pareto front MPSOC configurations available for selection.

The Xilinx system EDA tools Xilinx platform studio (XPS) is ran in no window mode with all batch commands launched from a *C* main program. Those input file specifications are used to control the physical design part of the implementation by synthesizing, placing, and routing the multiprocessor configurations onto FPGA platform devices. The generated

Table 5: EDK specifications files.

Files	Description	Comments
MHS	Microprocessor hardware specification	The MHS defines the hardware component
MSS	Microprocessor software specification	The MSS contains directives for customizing libraries, drivers, and file systems
MDD	Microprocessor driver definition	An MDD file contains directives for customizing software drivers
MPD	Microprocessor peripheral definition	The MPD defines the interface of the peripheral
MLD	Microprocessor library definition	the MLD contains directives for customizing software libraries and operating systems
PAO	Peripheral analyze order	Contains a list of HDL files that are needed for synthesis, and defines the analyze order for compilation.

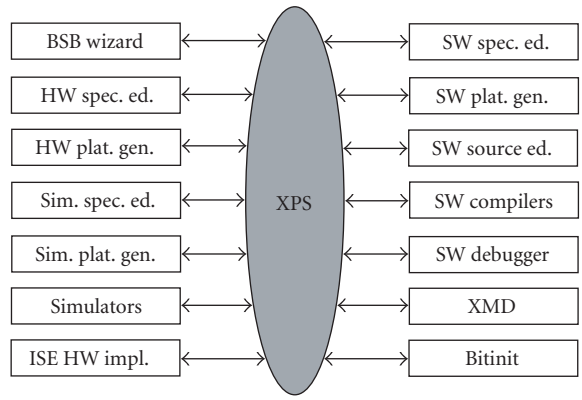


Figure 8: Xilinx EDK (XPS Xilinx platform studio).

FPGA configuration bitstream is downloaded on the FPGA device for execution and performance evaluation of the multiprocessor. The board hosting the FPGA device is an Alpha-data PCI FPGA board [3]. The implementation area and resources of the multiprocessor configurations are provided by the design automation tools composing part (2) while performance results in number of clock cycles are obtained from the actual execution of the multiprocessor configurations. These informations are automatically fed back to the DSE engine which runs on the host through the PCI bus.

The number of cycles are obtained directly from the execution, thanks to a timer connected to the MicroBlaze (MB0) OPB bus, which counts the number of clock cycles. After that, the execution time results are communicated to the host PC using an IP which bridges the MicroBlaze OPB bus to the PCI host bus. These results (occupied slices, occupied BRAM, and the execution time) are then injected as feedback input to the evolutionary algorithm for the next generation run. For this work we initially executed two explorations where the first consisted of a population size of 22 individuals and 10 generations (242 implementations with the initialization generation).

6. EXPLORATION RESULTS

6.1. Flow execution results

Figures 10 and 11 describe the corresponding results of these implementations. Figure 10(b) represents Pareto solutions for the second exploration where we attempted to increase the population size to 30 individuals and the number of generations to 14 in order to observe the behavior of the evolutionary algorithm for bigger explorations. From the results of second exploration it is obvious that the algorithm is converging to optimal solutions showing that for larger population size and generation size, potential of convergence is increased in NSGA-II algorithm as was expected. From the two preceding exploration flow executions, it appears as expected since we focused on embedded memories that the number of occupied slices does not vary much across multiprocessor configurations. However the variations are much more significant concerning both the number of occupied BRAMs and the execution time. So we decided to continue the execution of the proposed exploration flow in order to see its evolution.

Table 6: MHS file parts: Microprocessor IP, FSL IP, BRAM controller IP.

MicroBlaze processor	FSL communication	BRAM controller
BEGIN MicroBlaze	BEGIN fsl_v20	BEGIN lmb_bram_if_cntlr
PARAMETER INSTANCE = MicroBlaze_0	PARAMETER INSTANCE = fsl_v20_7	PARAMETER INSTANCE = ilmb_cntlr3
PARAMETER HW_VER = 3.00.a	PARAMETER C_FSL_DEPTH = 8	PARAMETER HW_VER = 1.00.b
PARAMETER C_FSL_LINKS = 2	PARAMETER HW_VER = 2.00.a	PARAMETER C_BASEADDR
BUS_INTERFACE MFSL0 = fsl_v20_2	PARAMETER C_EXT_RESET_HIGH = 0	= 0 × 00000000
BUS_INTERFACE SFSL0 = fsl_v20_1	PARAMETER C_IMPL_STYLE = 1	PARAMETER C_HIGHADDR
BUS_INTERFACE DLMB = dlmb0	PARAMETER C_USE_CONTROL = 0	= 0 × 00003fff
BUS_INTERFACE ILMB = ilmb0	PORT SYS_Rst = lreseto_l	BUS_INTERFACE SLMB = ilmb3
BUS_INTERFACE DOPB = mb_opb0	PORT FSL_Clk = lclk	BUS_INTERFACE BRAM_PORT
BUS_INTERFACE IOPB = mb_opb0	PORT FSL_M_Clk = lclk	= ilmb_port3
PORT INTERRUPT = Interrupt 0	PORT FSL_S_Clk = lclk	END
PORT CLK = lclk	END	
END		

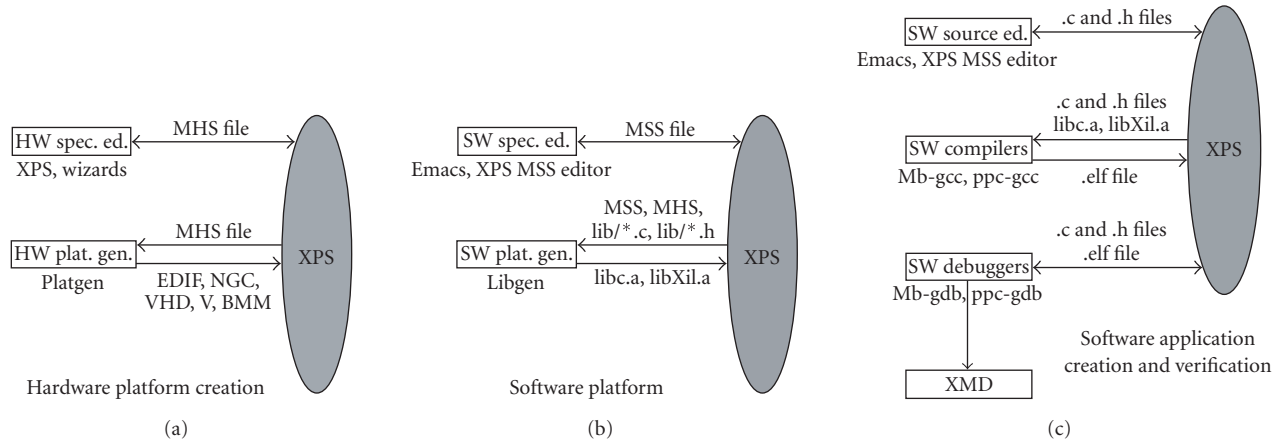


Figure 9: Xilinx EDK. (a) Hardware platform generation. (b) Software platform. (c) Simulation and verification.

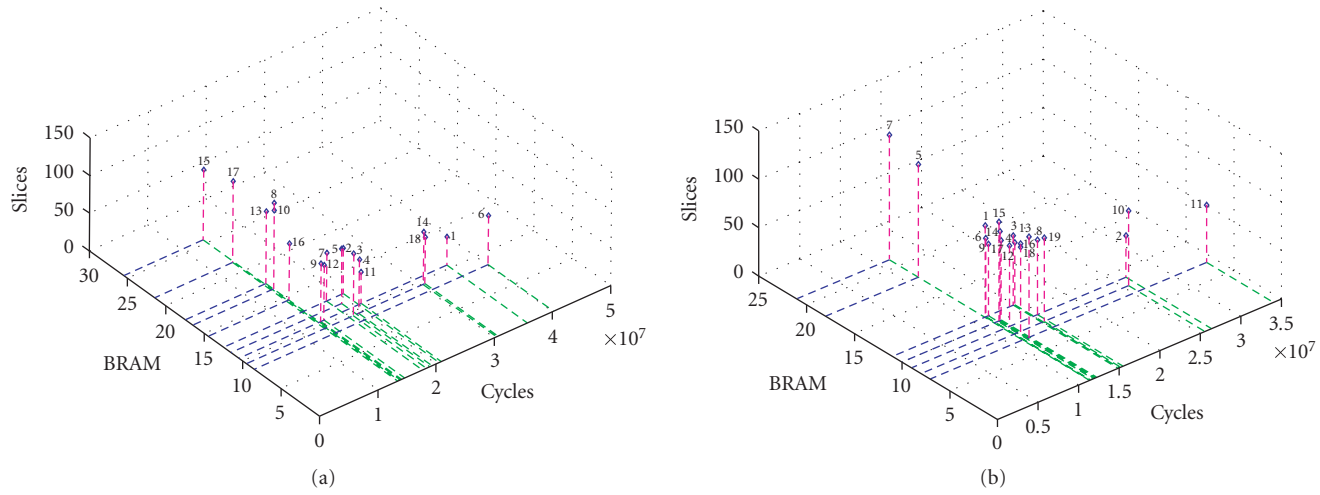


Figure 10: (a) For 10 generations-popsize = 22. (b) For 14 generations-popsize = 30.

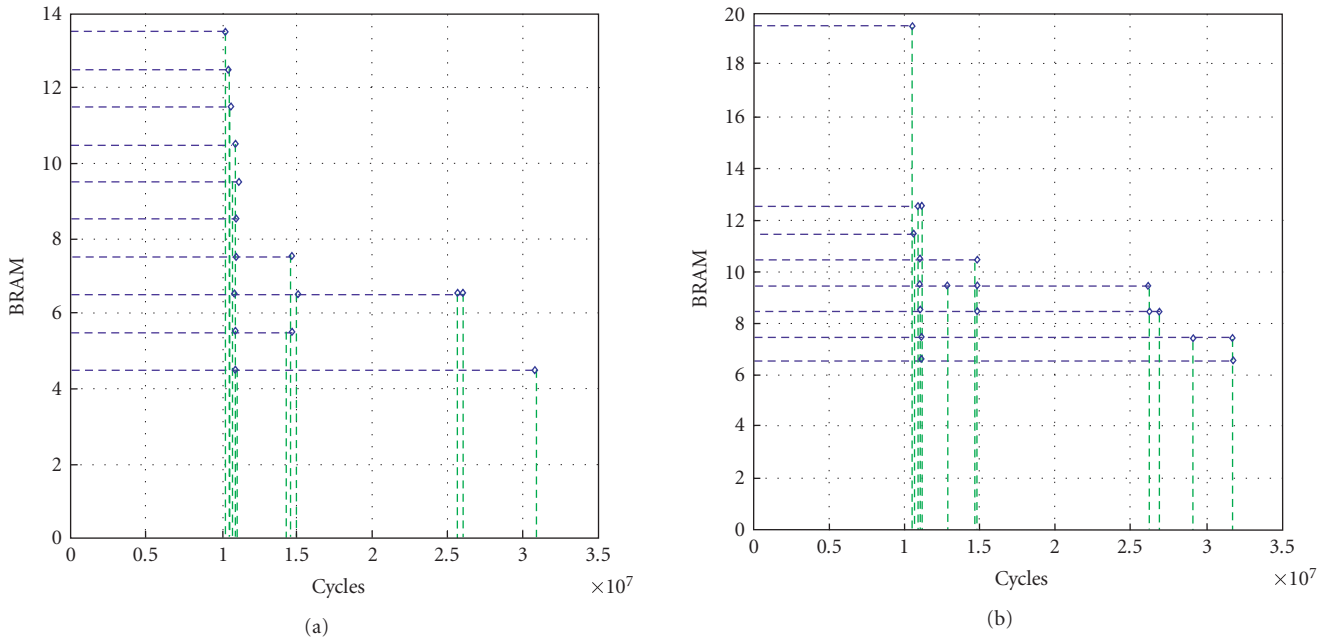


Figure 11: (a) For 30 generations-popsize = 30. (b) For 60 generations-popsize = 30.

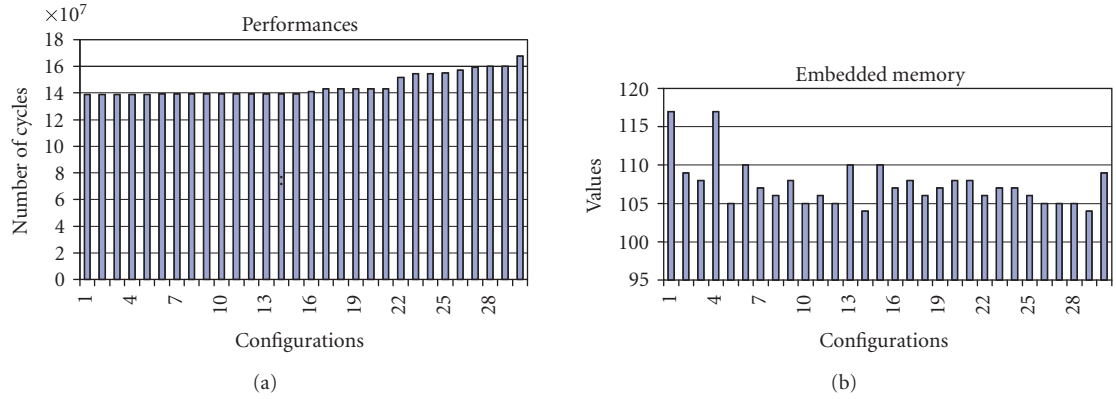


Figure 12: Pareto front. (a) Pareto front performance distribution. (b) Pareto front BRAM distribution.

For this second part of the exploration, we fixed the population size to 30 individuals and changed the number of generation to 30 and finally 60 generations. The results for each execution are, respectively, described in Figures 11(a) and 11(b). From these different figures we can clearly observe that the NSGA-II evolutionary algorithm tends to converge to the optimal Pareto solutions front which proves the correct implementation of the algorithm. The figures show different execution times for the same BRAM occupation meaning that using more BRAM will not systematically result in performance improvements.

However, to achieve better results BRAM resources need to be well distributed among the IPs where it would be used for getting optimal resource utilization.

Figure 12 shows the distribution of performance in the final Pareto front and clearly few configurations demonstrate superior performance while BRAM distribution for the same front demonstrates an uneven use of BRAM. This clearly shows the impact of BRAM careful distribution.

Examples of final Pareto front configurations are given in Table 7. The configurations chosen represent, respectively, 69.64%, 61.90%, and 64.88% of all BRAM resources. 11.11% BRAM reduction is obtained in the second configuration for a 0.004% increase in execution time while a 6.8% BRAM reduction is obtained in the third configuration for a 0.009% increase in the execution time.

6.2. *Flow execution time*

The results achieved in the previous section required the performance evaluation of 3120 different multiprocessor

Table 7: The design space associated with those parameters (74×118, thus 514 675 673 281 different configurations) requires 16 321 years of simulation for 1 minute simulation per configuration.

(a) Cycles: 138 974 816 BRAM: 109.

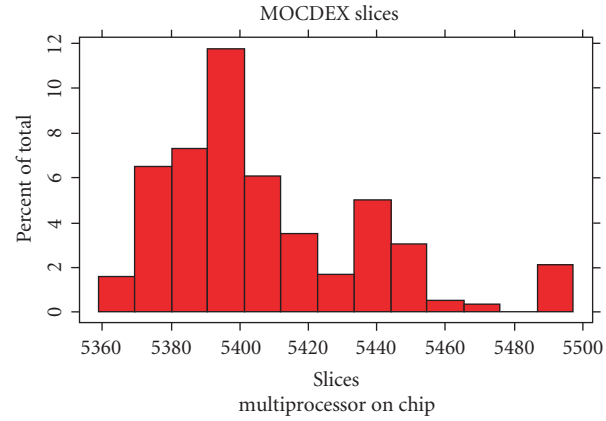
Procs	FSL1Out	FSL2Out	D-Cache	I-Cache
MB0	2048	2048	1024	4096
MB1	512	512	1024	1024
MB2	2048	512	2048	2048
MB3	1024	1024	4096	4096

(b) Cycles: 138 844 064 BRAM: 117.

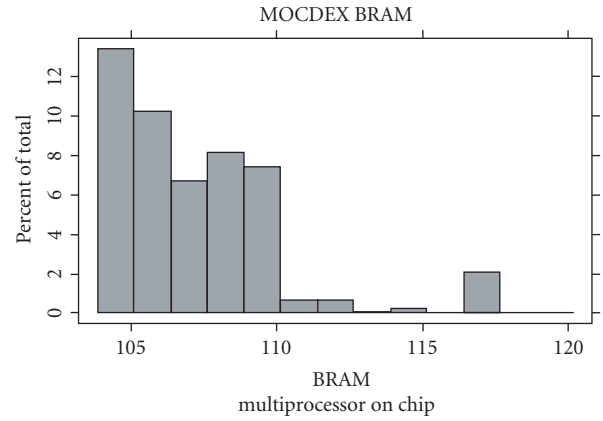
Procs	FSL1Out	FSL2Out	D-Cache	I-Cache
MB0	2048	128	2048	2048
MB1	256	32	2048	512
MB2	512	16	4096	512
MB3	1024	32	512	2048

Table 8: Flow execution time direct execution versus simulation.

Flow main steps	Functions	Time
Multi-objective evolutionary algorithm (ms)	Indi. Gene.	190
	Obj functions eval.	293
	Selection	0.116
	Crossover	0.033
	Mutation	1.118
Synthesis (sec)	Synthesis	523.503
	P and R	655.174
	P/R & Bitgen	797.856
Evaluation	Exploration 60×30	2250 days
	Sim. 64×64	1.39 hour
	Direct exec. 256×256	



(a)



(b)

Figure 13: Explored design space. (a) Slices histogram. (b) BRAM histogram.

on-chip configurations. These evaluations have been cycle-accurate after actual implementation on single-chip large scale FPGA devices. Contrary to traditional board-based multiprocessor, multiprocessors on chip are implemented on single chip; and due to the complexity of these architectures and the scale of the target devices, it is not possible to overlook the impact of place and route on the number of cycles required for various operations and on the cycle time.

It results from this fact that comparing different multiprocessors on chip configurations on the number of execution cycles is meaningless if one does not take into account the impact of place and route on each distinct configuration resulting from actual implementation. From this point mainly two alternatives exist: (1) post place and route simulation which will accurately represent the multiprocessor on chip behavior, and (2) emulation through direct execution. We conducted cycle accurate simulations using a powerful multi-language (SystemC, VHDL, Verilog-HDL) simulator ModelSim 6.0. Indeed, ModelSim 6.0 can handle large and complex designs and allow their simulation in a post-synthesis and post-place and route modes. Table 8 describes the very important time savings while using direct execution instead of simulation. Simulation would require 2250 days of simulation versus 1.39 hour for direct execution.

In order to reach the same evaluation speed at this level of accuracy it would require a compute farm (grid computing) of well over 25 000 workstations. The proposed flow execution time is obviously very competitive with regard to SystemC approaches [20, 21] for platform-based design. Similar observations have been drawn for embedded processors design space exploration [18, 19].

7. EXPLORED DESIGN SPACE STATISTICAL ANALYSIS

If we analyze in detail the complexity landscape of such a design space exploration we obtain the configurations distribution found in Figures 13 and 14. Clearly from these histograms we see that slices, BRAM, and performance (execution time) distributions in the explored design space are very different and demonstrate that the design space exploration was not confined in a limited subspace but explored a large diversity of multiprocessor configurations. The explored design landscape is given in Figure 15.

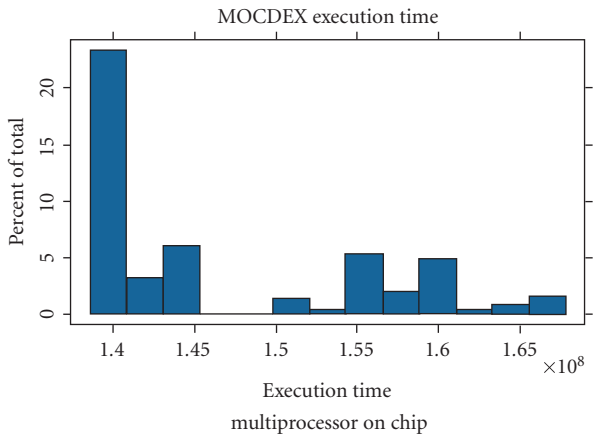


Figure 14: Explored design space execution time histogram.

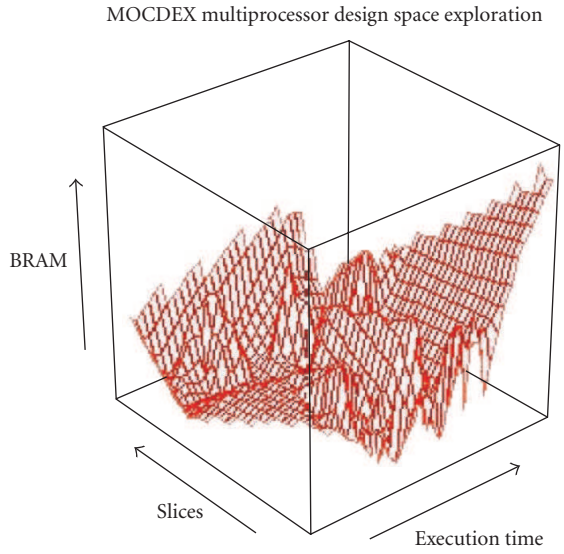


Figure 15: MOCDEX explored design space.

Figure 15 demonstrates the complexity of the design landscape and emphasizes the need to match this complexity with appropriate applied mathematics optimization techniques.

8. CONCLUSION

The design complexity of multiprocessors on chip requires efficient design methodologies. We propose in this paper a novel technique which fully integrates architectural design space exploration with design automation tools, where all area and performance results are obtained from actual post-synthesis place and route and actual execution on large scale FPGA platforms. To the best of our knowledge, our work is the first to fully integrate and therefore close the gap between design automation tools and architecture design space exploration technique in a multiobjective constraints paradigm with actual execution for all multiprocessor on chip configurations explored during the design space exploration process. It is important to note that actual execution reduces exploration time and can be exploited for either reducing design cycle time (i.e., TTM) and/or exploring even larger design space by including additional parameters. This work can be easily extended to include more parameters at various abstraction levels from architecture to circuit allowing interesting tradeoffs between usually uncorrelated various abstraction levels in the general design flow.

REFERENCES

[1] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs*, Springer, New York, NY, USA, 2002.

[2] C. A. C. Coello, D. V. Veldhuizen, and G. B. Lamont, *Evolutionary Algorithms for Solving Multi-Objective Problems*, vol. 5 of *Genetic Algorithms and Evolutionary Computation*, Kluwer Academic, Dordrecht, The Netherlands, 2002.

[3] Alpha-Data, ADM-XRC-II PCI mezzanine card, http://www.alpha-data.com.

[4] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, San Francisco, Calif, USA, 1999.

[5] A. A. Jerraya and W. Wolf, *Multiprocessor Systems-on-Chips*, Morgan Kaufman, San Francisco, Calif, USA, 2004.

[6] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya, "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 518–523, Las Vegas, Nev, USA, June 2001.

[7] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors," in *Proceedings of the 18th IEEE International Conference on VLSI Design*, pp. 551–556, Kolkata, India, January 2005.

[8] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer, "An automated exploration framework for FPGA-based soft multiprocessor systems," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES '05)*, pp. 273–278, New York, NY, USA, September 2005.

[9] N. K. Bambha and S. S. Bhattacharyya, "Joint application mapping/interconnect synthesis techniques for embedded chip-scale multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 2, pp. 99–112, 2005.

[10] Xilinx, Embedded system tools guide, http://www.xilinx.com/ise/embedded/edk_docs.htm.

[11] Xilinx microblaze soft core processor, http://www.xilinx.com/ise/embedded/mb_ref guide.

[12] I. Aouadi, R. B. Mouhoub, and O. Hammami, "System on a programmable chip oriented JPEG-2000 entropy coder implementation for multimedia embedded systems," in *Proceedings of the IEEE International Conference on Consumer Electronics (ICCE '05)*, pp. 447–448, Las Vegas, Nev, USA, January 2005.

[13] R. B. Mouhoub, I. Aouadi, and O. Hammami, "System on programmable chip platform based design of JPEG- 2000 entropy coder," in *Proceedings of the 12th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI '04)*, pp. 103–106, Kanazawa, Japan, October 2004.

[14] Xilinx Fast Simplex Link IP, http://www.xilinx.com.

[15] C. A. C. Coello, "An updated survey of GA-based multiobjective optimization techniques," *ACM Computing Surveys*, vol. 32, no. 2, pp. 109–143, 2000.

[16] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[17] M. T. Jensen, "Reducing the run-time complexity of multiobjective EAs: the NSGA-II and other algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 5, pp. 503–515, 2003.

[18] K. Ghali and O. Hammami, "Embedded processor characteristics specification through multiobjective evolutionary algorithms," in *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE '03)*, vol. 2, pp. 907–912, Rio de Janeiro, Brazil, June 2003.

[19] K. Ghali and O. Hammami, "Embedded processors optimization with hardware in the loop," in *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE '04)*, vol. 1, pp. 561–564, Ajaccio, France, May 2004.

[20] F. Fummi, S. Martini, G. Perbellini, and M. Poncino, "Native ISS-SystemC integration for the co-simulation of multi-processor SoC," in *Proceedings of the IEEE Conference and Exhibition on Design, Automation and Test in Europe (DATE '04)*, vol. 1, pp. 564–569, Paris, France, February 2004.

[21] F. Ghenassia, *Transaction-Level Modeling with SystemC TLM Concepts and Applications for Embedded Systems*, Springer, New York, NY, USA, 2005.

[22] Xilinx Virtex-II Platform FPGA, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_platform_fpgas/index.htm

Riad Ben Mouhoub received the Electronics Engineering degree in 2002 from the University of Algiers USTHB. He also received a Master degree in electronic systems and data processing from the University of Paris Sud Orsay in 2003. He currently holds a Doctorate position in electrical and computer engineering from the University of Paris Sud in the Department of Electronics and Computer Engineering at École Nationale Supérieure de Techniques Avancées in Paris (ENSTA). His research interests include design automation, design methodologies for multiprocessor system on programmable chips (MPSoPC), and NoC synthesis. He is a Student Member of the IEEE.

Omar Hammami is an Associate Professor at ENSTA/DGA since 2000. Prior to that he was Assistant Professor from 1991 to 1993 with ENSEEIHT, Toulouse, and Associate Professor with the University of Aizu, Japan, from 1993 to 2000. He received his Ph.D. degree in computer science and electrical engineering from Paul Sabatier University, Toulouse, in 1993 and has since worked in the field of circuits, system level design methodologies, embedded parallel architectures, and system on chip (SOC) for multimedia and wireless communications. He has been involved in numerous international and national research and industrial projects in those areas and have been funded by various government and funding agencies. He is a regular reviewer for various journals (IEEE, EURASIP, etc.) and conferences as Program Committee Member.

MOCSOC: Multiprocessor on Chip Synthesis from OCCAM

Riad Ben Mouhoub
ENSTA
32, Bvd Victor
Paris, 75739
riad.benmouhoub@ensta.fr

Omar Hammami
ENSTA
32, Bvd Victor
Paris, 75739
hammami@ensta.fr

Abstract— System on chip are increasingly complex to design and although multiprocessor on chip emerges as strong candidates for SOC architectures their design remains challenging. In addition, design space exploration strategies for multiprocessors on chip need to be conducted in conjunction with parallel software porting. Indeed, varying multiprocessors configurations affect parallel program design and therefore design space exploration of multiprocessors is bound by parallel software portability. Reversing the problem by synthesizing multiprocessors on chip from parallel applications written using parallel languages ease the complete design of embedded multiprocessors on chip based parallel applications. In this paper, we propose MOCSOC a multiprocessor on chip synthesis methodology from Occam. A case study of a neural network application demonstrates the validity of our approach.

I. Introduction

System on chip are increasingly becoming complex to design, test and fabricate. SoC design methodologies make intensive use of intellectual properties (IPs) [13] to reduce the design cycle time and meet stringent time to market constraints. However, associated tools still lag behind when addressing the huge associated design space exposed by the combination of soft IP. In addition, failure to meet an efficient distribution in terms of performance, area and energy consumption makes the whole design inappropriate. Although this problem is already hard to solve in the ASIC domain, it is exacerbated in the system on programmable chip (SoPC) domain. SoPC are large scale devices offering abundant resources but in fixed amount and in fixed location on chip. Implementing embedded multiprocessors on these devices present several advantages the most important being to be able to quickly evaluate various configurations and tune them accordingly. Indeed, embedded multiprocessor design is highly application driven and it is therefore highly advantageous to execute applications on real prototypes. Multiprocessors on chip can be quickly built by exploiting the design opportunities offered by SOPC. However, the design of efficient multiprocessors on chip cannot be considered without taking into account parallel applications to be run on it. Indeed, contrary to traditional general purpose multiprocessors design where multiprocessor architecture and hardware design and applications design are decoupled application specific embedded on chip multiprocessors can exploit application characteristics for joint parallel application-multiprocessor on chip design.The paper is organized as follows. In Section 2, we review previous work. Section 3 describes an example of soft IP based multiprocessor and the breadth of the configurations opportunities associated with the design of such multiprocessor . Section 4 presents our approach MOCSOC based on Occam based parallel program synthesis and direct execution. In Section 5 we describe a case study and validation while Section 6 provides exploration results. Section 7 describes future work while section 8 concludes.

II. Previous Work

The recent emergence of multiprocessors on chip as strong potential candidates to address performance, energy and area constraints for embedded applications has resulted in the following question: how do we design efficient multiprocessors on chip for a target application ? Design automation tools fail to address this question while traditional parallel computer architectures techniques [8] have not been exposed to the huge diversity brought by soft IP based design methodologies and the strong constraints of embedded systems [2]. Therefore, the design of multiprocessor on chip is the convergence focus of previously unrelated techniques and as such represents a new problem on how to establish a close integration between those techniques. It is then not surprising that few works so far have been devoted to design methodologies for multiprocessors on chip. In [9] they present a design flow for the generation of application-specific multiprocessor architectures. In the flow architectural parameters are first extracted from a high-level specification and are used to instantiate architectural components such as processors, memory modules and communication networks. They do not use parallel programs as starting base. In [20], of application-specific heterogeneous multiprocessor architectures using extensible processors is proposed based on an iterative improvement algorithm implemented in the context of a commercial design flow. Both architecture and implementation flows are still decoupled. In [12], they propose an automated exploration framework for FPGA-based soft multiprocessor systems. Using as input the application graph that describes tasks and communication links, outputs of the exploration step are a microarchitecture configuration of

processors and communication channels, a mapping of the application tasks and links onto the processors and channels of the micro-architecture. In [5], they present high-level scheduling and interconnect topology synthesis techniques for embedded multiprocessor systems-on-chip that are streamlined for one or more digital signal processing applications. To the best of our knowledge our work is the first to fully integrates and therefore close the gap between design automation tools and parallel programming by automatically synthesizing multiprocessors on programmable chip directly from parallel programs with automatic validation by execution.

III. Soft IP Based Embedded Multiprocessor System

Embedded multiporcessor systems based on soft IPs are SoC including ; soft IP processors, interconnect infrastructure and memories. An example of such a system is described below. It is mainly based on Xilinx Embedded Development Kit (EDK) IPs [23].

A. The Microblaze soft IP processor

The Microblaze soft IP [24] is a 32-bit 3-stages single issue (1) pipelined Harvard style embedded processor architecture provided by Xilinx as part of their embedded design tool kit.

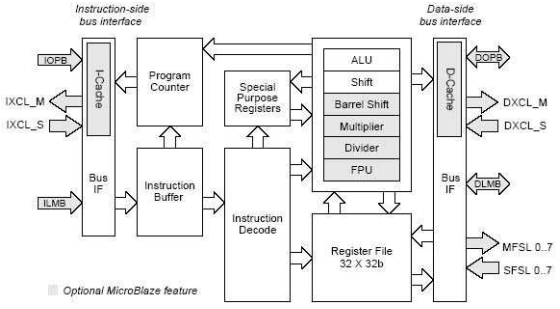


Fig. 1. Microblaze soft IP processor

Both caches are direct-mapped, with 4 word cache line allowing configurable cache and tag size and user selectable cacheable memory area. Data cache uses a write-through policy. The Microblaze core configurability extends to functional unit through user selectable barrel shifter (BS), hardware multiplier (HWM), divider (HWD) and floating point unit (FPU). The Microblaze has neither static nor dynamic branch prediction unit and supports branches with delay slots. For its communication purposes, the Microblaze uses either a bus or a direct link. The On-Chip-peripheral Bus (OPB) is part of IBM CoreConnect [23] bus architecture and allows the design of complete single processor systems with peripherals and user designed hardware accelerators (e.g. [6, 4]). However, even for a simple embedded processor such as the Microblaze, the OPB bus is not suitable for multiprocessors designs because of its lack of scalability. Another approach is provided by "Fast Simplex Link" [25] which allows direct connection between embedded processors through FIFO channels.

Caches	Sizes
Instruction	1K, 2K, 4K, 8K, 16K, 32K, 64K
data	2K,4K,8K,16K,32K,64K

TABLE I
Cache memories configurable values

B. Microblaze Fast Simplex Link

The Fast Simplex Link (FSL) [25] is an IP developed by Xilinx to achieve a fast unidirectional point-to-point communication between any two components (2). The FSL link is implemented as a 32-bit wide FIFO with configurable depth and width option. The FSL can be either a master or a slave interface depending upon its use.

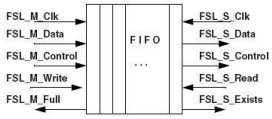


Fig. 2. Fast Simplex Link

The Microblaze soft embedded processor allows up to 8 masters and slaves FSL interfaces. Basic software drivers are provided to simplify the use of FSL connection. They consist of read/write routines and control functions. The read/write routines can be executed in two different ways: blocking and non blocking mechanism.

C. IBM Interconnect

The IBM Coreconnect [23] represents a set of IPs used to develop SOC devices. It includes the PLB and OPB bus, a PLB-OPB bridge and various peripherals.

D. MPSoC platform description

An example of FPGA multiprocessor platform consists of four Microblaze processors with instruction and data cache units. These processors are connected with each other through FSL channels. Each Microblaze is connected, as shown in Figure 3 to an OPB bus, to use a timer and an interrupt controller for threads and OS execution. Microblaze MB0 is connected to the OPB bus which is connected to a PCI of the host bus (WS). This allows the designer to send and receive data from the host to the multiprocessor system. We implemented a soft layer of communication in each Microblaze which performs send and receive functions of packets. The packets consist of headers representing the destination and source addresses and the number of flits in

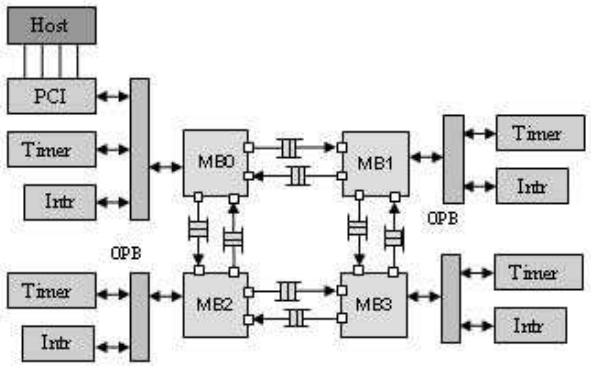


Fig. 3. Mesh Platform 2x2

the payload. A wormhole routing algorithm was used since it uses less memory, making it suitable for network on chip communication. As it can be seen a 4 way multiprocessor have been built based on the previously described soft IPs The implementation of such a soft IP multiprocessor on FPGA platform requires a variable amount of resources as each soft IP composing the multiprocessor requires a variable amount of resources depending on the configuration options. Such a soft IP multiprocessor can be easily adapted to the need of a specific adapted to a particular parallel application or completely generated using the previously described IPs from a parallel program specification.

E. Xilinx Platform FPGA EDA Tools

Large scale FPGA platforms devices unavoidedly raised the issue of system design tools in order to increase productivity. Xilinx propose two tools for Xilinx chips design : (1) EDK (2) ISE. The Xilinx XPS tool [23] provides a complete

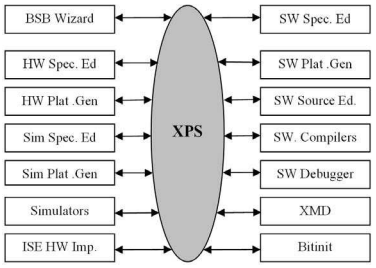


Fig. 4. Xilinx XPS tool

set of software tools and IPs for the design, simulation and hardware validation of multiprocessors on programmable chip.

IV. MOCSOC

A. Problem Formulation

The problem addressed by this paper is: how to automatically generate an embedded multiprocessor on chip from a parallel program designed with a parallel language ?

B. Parallel Languages and System Languages

Parallel languages [19, 22] have been the focus of intensive research for the past 3 decades in the context of parallel machines with fixed resources. A whole body of literature addresses [8] the issues of application mapping, scheduling and data distribution. Debates over parallel programming paradigms [17] and potentials of compilation for parallel systems [16] were bound to the static nature of those systems. On the hardware side and system level design, a couple of system level languages have emerged to ease the design of system on chip. The two main languages proposed are: SystemC 2.1 and SystemVerilog 3.1a. SystemC [1] provides hardware-oriented constructs within the context of C++ as a class library implemented in standard C++. Its use spans design and verification from concept to implementation in hardware and software [10]. SystemVerilog - the recently ratified hardware description and verification language (HDVL) standard [21] - is a major extension of the established IEEE 1364-2001 Verilog language, and was developed by Accellera to dramatically improve productivity in the design of large gate count, IP-based, bus-intensive chips. SystemVerilog is targeted primarily at the chip implementation and verification flow, with powerful links to the system level design flow. However, although both languages provide modeling capabilities for system on chip they are not convenient languages for the design of parallel programs. In addition they do not represent true parallel languages. Another parallel programming paradigm is MPI [14]. MPI program consists of autonomous processes, executing their own code, in a MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible. MPI-1 provides an interface that allows processes in a parallel program to communicate with one another. MPI-1 specifies neither how the processes are created, nor how they establish communication. Moreover, an MPI-1 application is static; that is, no processes can be added to or deleted from an application after it has been started. Other extension to MPI-1 exist such as MPICH , MPICH2 and MPI-2. The target language for the problem at hand must: (1) embody concurrent constructs inside the language (2) have demonstrated record of parallel programming (3) have demonstrated record of actual multiprocessor execution (4) be based on strong theoretical foundations regarding its concurrency semantics and be formally verifiable.

C. Occam

The parallel language Occam [18] is based on the strong theoretical foundation of communicating sequential pro-

cesses (CSP) [11, 7]. Occam enables an application to be described as a collection of processes where the processes execute concurrently, and communicate with each other through channels. Each process in such an application describes the behaviour of a particular aspect of the implementation and each channel describes a connection between two processes. This approach has two important consequences. Firstly, it gives the program a clearly defined and simple structure. Secondly, it allows the application to exploit the performance of a system which consists of many parts. Concurrency and communication are the prime concepts of the Occam model. Occam captures the hierarchical structure of a system by allowing an interconnected set of processes to be regarded as a unified single process. At any level of detail, the programmer is only concerned with a small manageable set of processes. Occam semantics simplify the task of program verification by allowing application of mathematical proof techniques to prove the correctness of programs. Transformations which convert a process from one form to a directly equivalent form can be applied to the source of an occam program to improve its efficiency in any particular environment. Occam makes an ideal language for specification and behavioural description. Occam programs are easily configured onto the hardware system or indeed may specify the hardware of a system. An Occam program is built upon a combination of elementary processes with the following possible constructs : Communications : communication is an essential part of occam programming. Values are passed between concurrent processes by communication on channels. Each channel provides unbuffered, undirectionnal point-to-point communication between two concurrent processes. The format and type of communication on a channel is specified by a channel protocol. The simplest protocols consist of a data type. Sequential protocols speciify a protocol for communication which consists of a sequence of simple protocols. A case protocol specifies a number of possible formats for communication on a single channel. SEQ and PAR A sequence (SEQ) combines processes into a construction in which one process follows another. A parallel combines a number of processes which are performed concurrently. The parallel terminates when all combined processes have terminated. Changing the order of the processes combined in a parallel does not change the effect of that parallel. Parallels may be nested to form the hierarchical structure of a program. Conditional : a conditional combines a number of processes each of which is guarded by a boolean expression. The conditional evaluates the boolean expressions in sequence; if a boolean expression is found to be true the associated process is performed and the conditional terminates. Alternation : an alternation combines a number of processes only one of which is executed. Each of the combined processes is guarded by a guard which may or may not be ready to proceed.

D. Porting Occam to Multiprocessors

Porting Occam onto soft IP multiprocessors can be achieved with different approaches one of them is to proceed through an intermediate step of translation towards a declarative sequential language enhanced with a library of communication primitives. This approach when developed in the context of established languages and libraries allows a dual validation of parallel applications. The first validation is functional and comes in the context of Occam code with compiler such as KRoC [15] while the second comes closer to the final architecture in the context of established language and libraries. In addition, portability is guaranteed accross platforms. We decided to translate Occam to C processes using the MPI library for communications. We ported a light version of MPI 1.0 environment on embedded multiprocessors on FPGA platform with limited embedded memory by selecting the following MPI calls: MPI_Send , MPI_Recv , MPI_ISend, MPI_IRecv, MPI_Ssend , MPI_Barrier(MPI_COMM Comm). The Occam to C+MPI translation flow is described by the following figure.

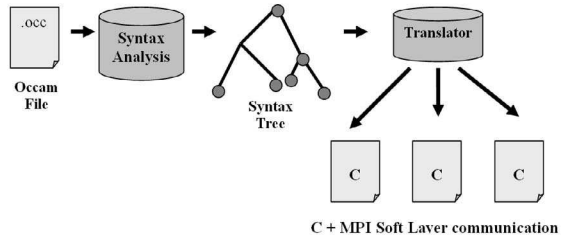


Fig. 5. Occam program translation

A message passing interface paradigm was preferred over a shared memory paradigm [8] due to almost a one to one mapping which can be achieved using soft IP multiprocessors between Occam parallel program structures and embedded processors using FSL IPs for their communications. The neat semantics of Occam ease this translation process.A lexical analyzer and a syntax analyzer have been developed based on Yacc tool. Concurrency expressed by PAR (as well as PLACED PAR) in Occam programs is therefore simply translated towards multiple C process with the assumption of execution on independent embedded processors. This minimalist approach is appropriate with the strong embedded memory constraints. Indeed, operating systems running on these multiprocessors on chip platforms should be tailored in order to reduce their memory footprint. Nested PAR constructs are therefore translated to SEQ constructs and does not affect program semantics.Starting from an Occam parallel program a syntax tree is built which is then decomposed into independant C programs with MPI routines for communication. In order to specify that a process needs to execute on predefined processor the langage construct PLACED PAR is used. In addtion to the translation of Occam parallel program to C+MPI processes the translator builds a communication graph of the Occam parallel program. This communication graph is used for deadlock analysis as well as communication estimation based on Occam code. This later informa-

tion is of particular interest for tailoring channel bandwidth in a GALS paradigm and will be the scope of future work.

E. Mocsoc

With regard to the above analysis steps we propose MOCSOC (Multiprocessor on Chip Synthesis from OCCAM) as a reliable and efficient design flow for multiprocessors on programmable chip. The key ideas of our approach are the following:

- soft IP multiprocessors reverse the usual parallel program mapping problem on fixed multiprocessor architectures to embedded multiprocessor mapping to parallel program
- large scale programmable FPGA platform allows for non conventional network on chip and allows for one-to-one mapping between parallel program communication graph and network on chip,
- large scale programmable FPGA platform allows for a large number of elementary customizable embedded processors encouraging therefore spatial parallel programming,
- portability of parallel programs should target soft IP multiprocessors not hard multiprocessors
- the parallel system can be represented by the parallel program provided that automatic translation of parallel program to complete running embedded on chip multiprocessors is achieved.

Our proposal follows: The flow is mainly composed of 3 parts (See Figure 6): (1) Occam translation to C+MPI parallel software (2) embedded multiprocessor generation (DSE) (3) physical design (4) FPGA platform PCI board. The Occam translation, embedded multiprocessor generation, physical design run on an host PC. Firs the user provide an Occam parallel code and may specify the IPs for target generation then translate those parameters into platform EDA design tool input file specifications. MOCSOC (Xilinx FPGA Platform):
Specify a system application through parallel programming using Occam for all configurations

- generate hardware/software platform specification files, (mhs,mpd, pao,mss, mld, mdd, files) generate through Xilinx system XPS and Xilinx IPs HW/SW model of the MPSOC
- synthesize/place and route MPSOC configuration using ISE 6.3
- record place and route reports,
- download configuration file on FPGA platform,
- execute MPSOC configuration and record execution clock cycles,

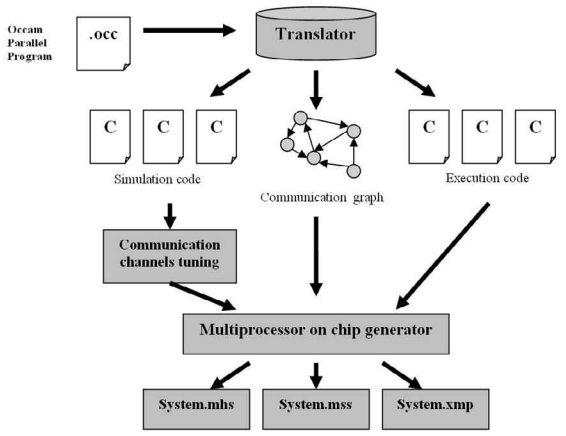


Fig. 6. MoCSoC - Occam to multiprocessor on programmable chip flow

Final Pareto front MPSOC configurations available for selection. The Xilinx system EDA tools XPS (Xilinx Platform Studio) is ran in no window mode with all batch commands launched from a C main program. Those input file specifications are used to control the physical design part of the implementation by synthesizing, place and route the multiprocessor configurations onto FPGA platform devices. The generated FPGA configuration bitstream is downloaded on the FPGA device for execution and performance evaluation of the multiprocessor. The board hosting the FPGA device is an Alpha-Data PCI FPGA board [22]. This board consists of a Virtex-II 8000 which is the largest of the Virtex-II family with 8 millions system gates and offers 3,024 Kbits of BRAM [3].The number of cycles are obtained directly from the execution, thanks to a timer connected to the Microblaze (MB0) OPB bus, which counts the number of clock cycles. After that, the execution time results are communicated to the host PC using an IP which bridges the Microblaze OPB bus to the PCI host bus. These results (Occupied slices, occupied BRAM and the execution time) are thenfed back to the parallel program designer for potential Occam code rewriting.

V. Case Study and Validation

The previously described design flow have been applied in the framework of Xilinx FPGA platforms. As a case study of our proposed multiprocessor on chip design flow we propose its application on a widely used neural network application the self organising map (7). The main objectives of this case study are the : (1) correct automatic software generation of the application software (2) correct automatic hardware generation of the multiprocessor platform (3) automatic validation of the system execution on large scale FPGA platform. The architecture of SOM networks consists of a two-dimensional array of neurons with each neuron connected to all input nodes. Further, all the neurons also have lateral connections to each other.The strength of

lateral connections follow a Mexican hat function. By involving a neighborhood function to achieve the effect of the Mexican hat lateral interactions, Kohonen devised a SOM network in which no lateral connections exist. There are n input units and the weight connecting input units to the output neuron i is denoted by the vector Wi. When an input pattern X is applied each output neuron computes the Euclidean distance between the input vector and the weight vector of that neuron. Competitive learning rule is then used and the neuron whose weight vector is closest to the input vector is chosen as the winner that is if the winning neuron is r then :

$$\|X - W_r\| = min_i \|X - W_i\| \quad (1)$$

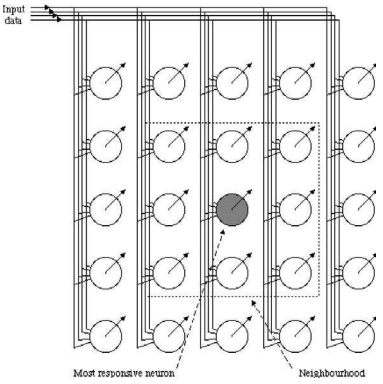


Fig. 7. Self Organizing Map algorithm

The network learns by changing appreciably the weights of the winning neuron and its neighbors by dragging their weight vectors toward the input pattern X whereas those far away from the winning neuron experience little change to their weights. This is how the topology preservation is achieved. The weight update equation is given by :

$$W_i^{new} = W_i^{old} + \alpha N\,(i,r)\,(X - W_i) \quad (2)$$

The function $N(i,r)$ is called the neighborhood function and its value is 1 for $i = r$ and falls off as the distance between the neurons r and i increases. A is the learning rate. The range of $N(i,r)$ and the value of a are reduced gradually as learning progresses. A common choice for $N(i,r)$ is :

$$N\,(i,r) = exp\left(-\|d_i - d_r\|^2 / \left(2\sigma^2\right)\right) \quad (3)$$

where d_i and d_r are vectors indicating the positions of neurons i, r and s is a width parameter that controls the range of $N(i,r)$ and is gradually decreased as learning proceeds. The concurrent version of the algorithm is a master-slave version where in a first step a master process send each input vector to all slave processes which compute the distances. The master receive in return all the computed

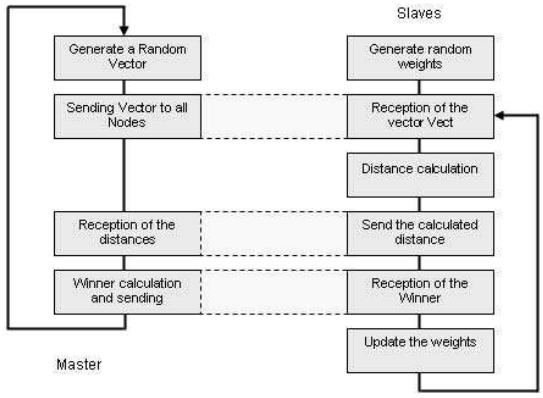


Fig. 8. Parallel self organizing map algorithm

distances and determine the winning neuron. This information is sent back in order for the concerned neurons to update their weights.

In our case the Occam implementation of SOM have been achieved with 1 master process and 8 slave processes. The general structure of the program is quite simple and is fully specified by a PAR Occam structure. After compiling the Occam program we obtain C code sources for each generated embedded processor of the architecture as well as a communication graph and the communication graph expose exactly the communication scheme of the application.

VI. Implementation Results

Figure 9 and Figure 10 describe the corresponding results of these implementations. Figure 9 represents the generated architecture out of the Occam communication graph while figure 10 describes the XPS tool output fully describing the hardware implementation.

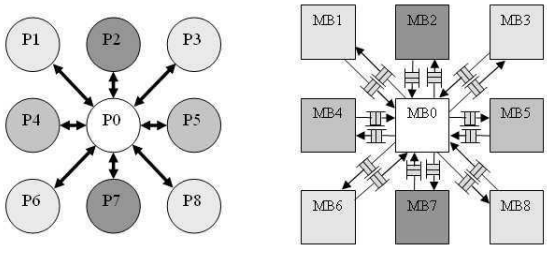


Fig. 9. Synthesized architecture

Figure 10 is flattened with regard to the actual place and route results.

The implementation uses 99 out of 168 RAMB16s (58%) and 10891 out of 46592 slices (23 %). Execution takes 8.6 s for a full learning and execution phase.

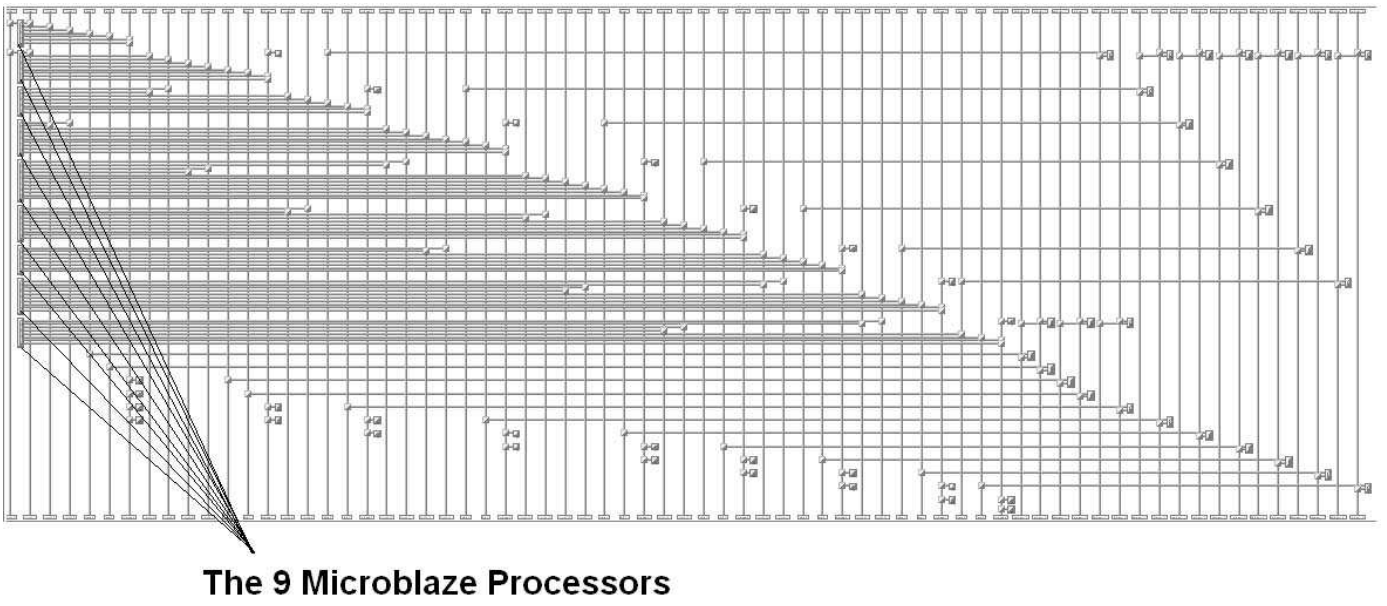


Fig. 10. Multiprocessor generation

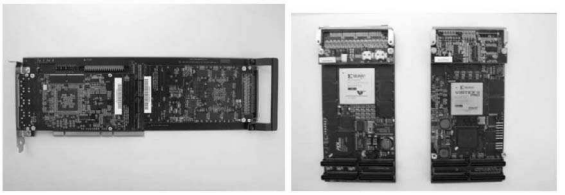


Fig. 11. FPGA Board Xilinx XC2V8000

VII. Conclusion

The design complexity of multiprocessors on chip requires efficient design methodologies. We propose in this paper a novel technique which fully integrates architectural design space exploration with design automation tools where all area and performance results are obtained from actual post-synthesis place and route and actual execution on large scale FPGA platforms. To the best of our knowledge our work is the first to fully integrate and therefore close the gap between design automation tools and parallel programming tools by allowing complete generation of a embedded multiprocessor platform fully tailored to the requirements of the parallel program. This transparent process is validated by automatic actual execution on system on programmable chip. Future work wil extend the flow by adding design space exploration at various levels of the design flow including Occam automatic rewriting.

References

[1] Systemc 2.1 lrm. http://www.systemc.org/.

[2] A.A.Jerraya and W.Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, June 2004.

[3] Alpha-Data. Adm-xrc-ii pci mezzanine card. Available on: http://www.alpha-data.com/adm-xrc-ii.htm.

[4] I. Aouadi, R. Benmouhoub, and O. Hammami. System on a programmable chip oriented jpeg-2000 entropy implementation for multimedia embedded systems. In *The International Conference on Consumer Electronics (ICCE)*, pages 447–448, Jan 2005.

[5] N. K. Bambha and S. S. Bhattacharyya. Joint application mapping/interconnect synthesis techniques for embedded chip-scale multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, volume 16, pages 99–112, Feb 2005.

[6] R. Benmouhoub, I. Aouadi, and O. Hammami. System on programmable chip plateform based design of jpeg-2000 entropy coder. In *Workshop on synthesis and system integration of mixed information technologies (SASIMI'04)*, pages 103–106, Oct 2004.

[7] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. In *Journal of the ACM (JACM)*, volume 31, pages 560 – 599, July 1984.

[8] D.E.Culler, A.Gupta, and J.P.Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1997.

[9] D.Lyonnard, S.Yoo, A.Baghdadi, and A.A.Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *DAC*, pages 518–523, 2001.

[10] F.Ghenassia. *Transaction-Level Modeling with SystemC TLM Concepts and Applications for Embedded Systems*. Springer, 2005.

[11] C. Hoare. Communicating sequential processes. In *Communications of the ACM*, volume 21, pages 666–677, 1978 1978.

[12] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer. An automated exploration framework for fpga-based soft multi-

processor systems. In *International Conference on Hardware/Software Codesign and System Synthesis(CODES-05)*, September 2005.

[13] M.Keating and P.Bricaud. *Reuse Methodology Manual for System-On-A-Chip Designs*. Springer, 2002.

[14] MPI. Message passing interface. Available on: http://www.mpi-forum.org/.

[15] P. H. W. Neil C. Brown. *An Introduction to the Kent C++ CSP Library*. WoTUG, 2003.

[16] K. R.Gupta, S.Pande. Compilation techniques for parallel systems. In *Parallel Computing 25*, pages 1741–1783, 1999.

[17] R.J.Anderson and L.Snyder. A comparison of shared and nonshared memory models of parallel computation. In *Proc. of the IEEE*, volume 79, April 1991.

[18] SGS-THOMSON. Occam 2.1 reference manual. Technical report, THOMSON, May 1995.

[19] D. Skillicorn and D.Talia. Models and languages for parallel computation. In *ACM Computing Surveys (CSUR)*, volume 30, June 1998.

[20] F. Sun, N.K.Jha, S.Ravi, and A.Raghunathan. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *In 18th International Conference on VLSI Design*, pages 551–556, 2005.

[21] SystemVerilog. Systemverilog. Available on: http://www.systemverilog.org/.

[22] W.Hasselbring. Programming languages and systems for prototyping concurrent applications. In *ACM Computing Surveys (CSUR)*, volume 32, March 2002.

[23] Xilinx. Embedded system tools guide. Available on: http://www.xilinx.com/ise/embedded/edk_docs.htm.

[24] Xilinx. Microblaze soft core processor. Available on: http://www.xilinx.com/.

[25] Xilinx. Xilinx fast simplex link ip. Available on: http://www.xilinx.com/.

NoC Monitoring Hardware Support for Fast NoC Design Space Exploration and Potential NoC Partial Dynamic Reconfiguration

Riad Ben Mouhoub
ENSTA
32 Bvd Victor 75739 Paris
riad.benmouhoub@ensta.fr

Omar Hammami
ENSTA
32 Bvd Victor 75739 Paris
hammami@ensta.fr

Abstract

The Multiprocessor systems on chip are strongly emerging in various embedded systems to support dramatic growth of complex embedded applications performance requirements. Due to the increasing scale of embedded systems bus-based communication no longer meet bandwidth requirements and therefore networks-on-chip (NoC) are increasingly used to process communication in embedded parallel applications. So far, neither development environments and tools for embedded systems nor profiling and debugging techniques of embedded systems tackled the issue of network on chip monitoring. Due to the complexity of future multiprocessors systems on chip parallel programmers will unavoidably need to be able to get accurate profiles of communication patterns on various network on chip links and this in order to optimize their applications through timing analysis, timing predictability, and real-time scheduling analysis. We propose in this paper a scalable network on chip real time hardware monitoring feedback for multiprocessors systems on chip parallel programmers. Implementation of our scheme for a 2x2 mesh based multiprocessor systems on chip demonstrates the validity of our approach for an image processing application.

1. Introduction

Multiprocessor systems on chip are strongly emerging in various embedded systems to support dramatic growth of complex embedded applications performance requirements [1]. Due to severe area and energy consumption constraints efficiency of these systems is an absolute requirement. Poor performance tuning either on existing devices or during the design of such systems will unavoidably results in excessive energy consumption and/or increased fabrication cost for devices targeted at consumer electronic market where considerable cost and competition pressure exists. Although, so far MPSOC have been based on traditional bus structure new complex multimedia applications are strongly suggesting the use of network on chips for communication. An alternative to bus based communication is network-on-chip [2, 3]. This is essentially based on on-chip packet switched communication thanks to on-chip integrated routers [4]. Using Network-on-chip offers several advantages such as communication scalability, better performance, modularity and structured communication schemes [5]. This communication platform fulfils the requirements of on chip cores communication such as in multi-media applications where the bandwidth reaches the GBytes/s range. It also allows perfect flexibility and reusability in the design of complex SoC applications and fits the natural communication scheme of any given application. These properties of Network-on-chip have been verified in several research works and had been implemented in both VHDL [6, 7] and systemC [8] description.

Although, a full and reliable exploitation of these interconnects requires the system designer to well parameterize the different communication components. To know how the different communication components of a network-on-chip platform behaves helps the system designer to have an optimal use of the network. Once we get the behaviour and utilization of the communication components such as routers and links, we can decide of the most suitable routing method, optimal depths of communication buffers as well as the communication topology. What we are presenting in this paper is an instantaneous Network-on-chip performance evaluation of any given application that takes into account real place and route implementation constraints. Taking advantage of the easy programmability and the large integration capacity of actual FPGA, helps us to provide a faster performance evaluation methodology by bypassing the problem of the prohibitive simulation time. In this situation, including specific Programmable IPs in the design for hardware monitoring seems to be a good way to provide the designer with instantaneous and real results, allowing him to improve his application implementation. Hence, we connected monitor IPs to the Xilinx Fast Simplex Link and NoC routers so we can analyze their occupation during run time. As a case study

1-4244-0777-X/06/$20.00 ©2006 IEEE

for our approach, we implemented a 2x2 Network-on-chip with four routers performing image filtering. The paper is organized as follows. In Section 2, we review previous work. Section 3 describes briefly the Microblaze based NoC and the Network-on-chip platform. Monitor IP architecture is described in section 4. Section 5 gives an overview of simulation of System-on-chip designs. In section 6 we give details on the different implementation of the filtering application and its relative monitoring results. Finally, we conclude in section 7 with remarks.

2. Previous Work

Network on chip monitoring for multiprocessors systems on chip can be mainly divided in two categories: (1) simulation based approaches and (2) emulation/hardware based approaches. Simulations based approaches (e.g. [9], [10], [11]) suffer from prohibitive simulation time which is not compatible with complex embedded systems on chip applications and can hardly provide real time feedback for parallel programmers. Emulation/hardware based approaches started in the last decade in traditional parallel systems (e.g. multi-processors) and several ideas are relevant to this work among them the non intrusiveness of the measurements, dedicated hardware monitoring unit and speed of events sampling [12, 13]. However, multiprocessors systems on chip differ by the various constraints on area, place and route resulting from a single chip implementation.

In [14], authors present an approach for NoC monitoring based on traffic generators and an FPGA emulation platform. Realistic traffics are generated to explore the behaviour of the network although only one traffic generator (TG) have been used. This study is a NoC oriented performance study and is of little help for a parallel programmer dealing with the optimization of its own application. Our work differ by providing real time NoC monitoring feedback to the programmer and the emulation platform is used in that sense. The work which is closest to ours is [15] in which a complete NoC monitoring framework is proposed and where hardware monitoring area requirements are analyzed on both a 3x1 mesh and 2x3 mesh. They targeted a 20% area overhead which is about the same as ours for a 2x2 mesh. Their proposal conducted in the framework of Aethereal design flow have been implemented for area analysis but no actual execution monitoring have been reported but rather simulation based results. Although there is no doubt that their proposal might easily results in an actual chip so far no actual execution is reported. Our work has been validated through actual execution of a 2x2 mesh based multiprocessor on a single chip large scale FPGA device running an image processing application and therefore not based on traffic generator. It is of immediate use for parallel programmers.

3. Microblaze Based Embedded Multiprocessor System-on-chip

As a case study for our approach, we have designed a multiprocessor architecture based on 4 microblaze embedded processors.

3.1. The Microblaze soft IP processor

The Microblaze soft [16] is a 32-bit 3-stages single issue pipelined Harvard style embedded processor architecture provided by Xilinx as part of their embedded design tool kit. Both caches are direct-mapped, with 4 word cache line allowing configurable cache and tag size and user selectable cacheable memory area. Data cache uses a write-through policy. The Microblaze core configurability extends to functional unit through use of a selected barrel shifter, hardware multiplier, divider and floating point unit. The Microblaze has neither static nor dynamic branch prediction unit and supports branches with delay slots. For its communication purposes, the Microblaze uses either a bus or a direct link. The On-Chip-peripheral Bus (OPB) is part of IBM Coreconnect bus architecture and allows the design of complete single processor systems with peripherals and uses designed hardware accelerators [17, 18]. However, even for a simple embedded processor based multiprocessors designs such as the Microblaze, the OPB bus is not suitable because of its lack of scalability. Another approach is provided by ”fast Simplex Link” [16] which allows direct connection between embedded processors through FIFO channels.

3.2. Microblaze Fast Simplex Link

The Fast Simplex Link (FSL) [19] is an IP developed by Xilinx to achieve a fast unidirectional point-to-point communication between any two components. The FSL link is implemented as a 32-bit wide FIFO with configurable depth and width option. The FSL can be either a master or a slave interface depending upon its use. The Microblaze soft embedded processor allows up to 8 master and slave FSL interfaces. Basic software drivers are provided to simplify the use of FSL connection. They consist of read/write routines and control functions. The read/write routines can be executed in two different ways: blocking and non blocking mechanism.

3.3. Network-On-Chip Platform Description

Our FPGA multiprocessor platform consists of four Microblaze processors with instruction and data cache units. These processors are connected with each other through a packet switched interconnect network. Each Microblaze is connected, as shown in Figure 1 to an OPB bus, in order to use a timer and an interrupt controller for threads and OS execution. Microblaze MB0 is connected to the OPB bus which is further connected to a PCI of the host bus (WS). This allows the designer to send and receive data from the host to the Multiprocessor system.

Microblaze processors are connected to the routers local port through their FSL (Fast simplex Link) described in subsection 3.2. The used router has a routing logic and five bi-directional communication ports : West, North, East, South and Local. At each Input/Output port there exists a queue consisting of buffers, for temporary storage of packets, with a parametrizable depth capacity. The local port is connected to the local processor (Microblaze) while the resting ports are connected to the other router ports, each port consists of an 8-bit vector. The packets consist of headers representing the destination, source addresses and the number of flits in the payload, where each flit consists of 8-bit.

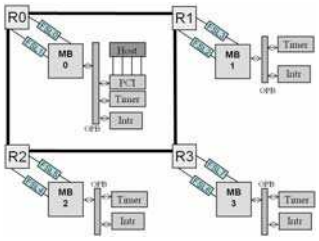


Figure 1 Mesh Platform 2x2

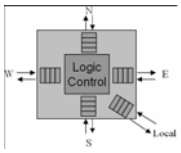


Figure 2 Routeur

The remaining 16 bits in the header packet were used for time stamping, of which more details will be given in section 4. For an optimal use, destination and source addresses are coded in four bits for addressing a maximum of 16 processors. This addressing capacity can easily be raised up. Routing is executed in a deterministic XY mode where Wormhole routing algorithm was used since it requires less memory, making it suitable for network-on-chip communication. In this routing mode, the header flit reserves the Input/Output buffers of the switch in order to trace the channels where the rest of the packet will follow, the last packet flit frees the reservation. As said before, the processors are connected to their respective routers through FSL channels. An interruption is used to read the packets sent to the processors. A signal detects the existence of data in the FSLs, if this is the case, it triggers an interruption routine that collects these data elements.

3.4. Alpha-Data Board

The hardware monitoring was performed on an Alpha-Data FPGA board. The Alpha data hardware environment is composed by : (1) the ADC-PMC and (2) the ADM-XRC-II. The ADC-PMC is a Dual PMC adapter for PCI. It supports 64-bit 66MHz primary and secondary PCI via an Intel 21154 PCI-PCI bridge device. The ADM-XRC-II 3 is a high performance reconfigurable PMC (PCI Mezzanine Card) based on the Xilinx Virtex-II range of Platform FPGAs. Features include high speed PCI interface, external memory, high density I/O, programmable clocks, temperature monitoring, battery backed encryption and flash boot facilities. On board clock generator provides a synchronous local bus clock for the PCI interface and the Virtex-II FPGA.

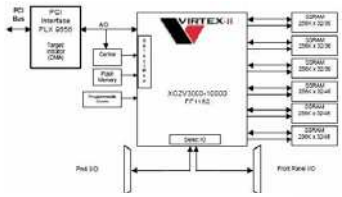


Figure 3 Alpha-Data FPGA Board

XC2V8000	VALUES
Slices	46,952
BRAM (18 Kbits)	168
18x18 multipliers	168
DCM	12
Max. Dist RAM Kb	1,456

Table 1 VIRTEX-II 8000

A second clock is provided to the Virtex-II FPGA for user applications and can be free running or stopped under software control. Both clocks are programmable and can be used by the Virtex-II Clock. The user clock has a maximum value of 100 Mhz. The ADM-XRC-II uses a Xilinx XC2V8000-5 FF1152 device [20]. The ADM-XRC SDK is a set of resources including an application-programming interface (API) intended to assist the user in creating an application using one of Alpha Data's ADM-XRC range of reconfigurable coprocessors. The API makes use of a device driver that is normally not directly accessed by the user's application. The API library takes care of open, close and device I/O control calls to the driver. The ADM-XRC SDK is designed to be thread-safe. The following table describes the main API functions which allow initializing the board, configuring the FPGA though the PCI bus, and transferring data between the FPGA and the host computer and the interrupt handling.

4. IP Monitor Description

4.1. Monitoring Framework

The proposed monitoring framework is depicted in Figure 4. The purpose of this idea is to allow the designer to gather real time information about its application by mean of integrated monitor IPs. The problem is to keep the overall application behaviour by using non-intrusive and compact monitoring IPs. We decided to use a dedicated embedded processor allowing an easier utilization of the monitoring IPs.

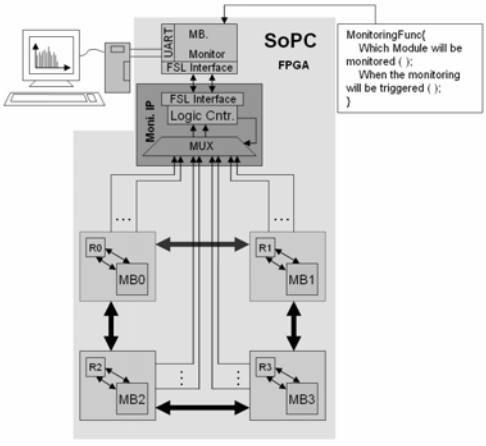


Figure 4 Monitoring Framework

It also permits a faster reprogrammability of the different registers of the monitoring IPs without FPGA reconfiguration. In this work, two parameters are collected from the network-on-chip platform execution. The first information is about the FSL FIFO occupancy. At each cycle the monitoring process verifies if there is a change in the FSL occupancy. If that is the case, the value of the occupancy and the time where the write or the read operation occurs are sent to the monitoring processor. We defined a cycle count process to provide time information. The second information consists of the time spent to route one packet. To get this time, we insert in the last four flits of the header packet (16-bit), the information time corresponding to the beginning of the packet routing. Once the packet reached its destination, this time information is compared with a global time counter to extract the routing time data. Information gathered (FSL Occupancy or Packet routing time) are sent to the monitor processor through an FSL channel. This FSL channel has a depth of 8192. Once data is received by the processor it can be stored in a RAM or sent directly to the host computer through a UART connection. The UART interface is easy to make but the offered data transfer speed is not satisfactory, we used it to achieve quickly the framework of the monitoring platform and we plan to replace it with a faster interface such as the USB or Ethernet. From the monitor processor we can assign the FSL channel to be monitored and the cycle at which we want to start the monitoring process. If the RAM space is not enough we restart the application execution where it stopped and collects the rest of the data (In case of a deterministic application).

4.2. Monitor IP Description

Figure 5 illustrates the block diagram of the Monitor IP. It consists of registers, comparators, counters (For cycles) and multiplexers. From the monitor processor and through its FSL channel, multiplexer receives three information components: the chosen component for monitoring (SEL signal), the information to be gathered (Info Type signal) and the clock cycle at which the monitoring process will start (Cycle Startsignal).

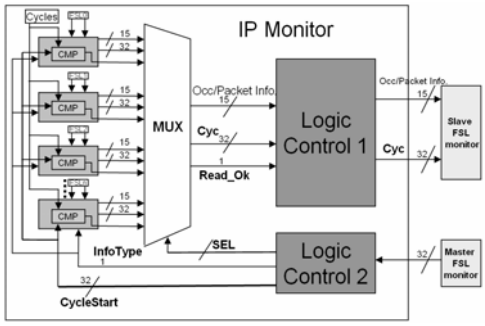


Figure 5 Monitor IP

The SEL signal is the selection input for the multiplexer. The outputs of the multiplexer are the OCC/PacketInfo signal for the FSL occupancy or information about packet routing time, the CYC signal for timestamps and the Read Ok signal to inform the control state machine about the arrival of a new data. A resource utilization of the FPGA is shown in Table 2 (U.M: Used resources by the monitoring IP and U.App: Used resources for the application). It principally includes the monitor IP, the Microblaze processor, the UART and external RAM controllers. This resources utilization varies depending on how they are used. For example the FSL receiving data has a depth parameter that can vary from 2 to 8192. From the Table 2 we remark that even for a maximum resource utilization of the monitoring IP, number of consumed slices is still low and acceptable. The most important problem comes from the FPGA embedded Block RAMs (14% for a maximum use) but this problem can be easily resolved by using external RAM.

	Min		Max	
	Used M	U.M/U.App.	Used M	U.M/U.App.
Slices	1,480	0.19	1,986	0.26
LUTs	1,672	0.12	2,628	0.19
FFs	1,406	0.18	1,990	0.24
Block RAMs	6	0.05	25	0.23

Table 2 Resources utilization

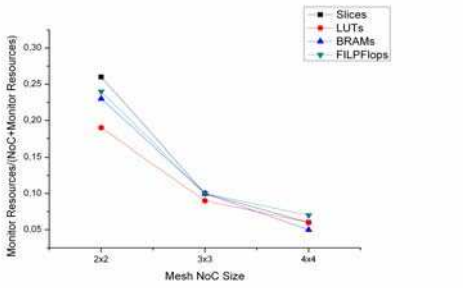


Figure 6 Monitor IP Resources compared with the whole design resources

The comparison has been done with different mesh network-on-chip sizes. Depending on these results we remark that the resources used for the monitor IP does not compromise the implementation of the whole design and this is verified even for bigger NoC designs.

5. Experimental Validation and Case Studies

5.1. Filtering Application

The filtering application, presented in figure 7, was chosen because it requires extensive data processing and data communication among the filters for a good and fast testing of our monitoring framework. We explored two programming modes. In the first programming mode MB0 receives image data form the workstation and through the PCI bus. Data is directly sent to the MB1, where it is stored in the local memory. MB1 performs a median filtering which results in noise reduction from the image. It is performed on a 3 by 3 pixel window where the centre pixel value is replaced by the median of the neighbouring pixel values.

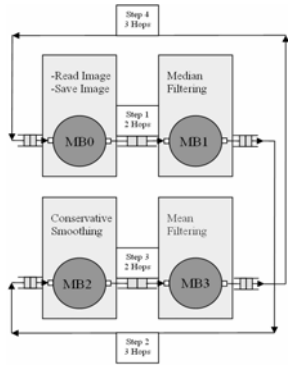


Figure 7 Filtering platform

This value is obtained by sorting the pixels based on their numerical values and then replacing the pixel to be processed by the middle value. The treated image is then sent to the MB2 where a conservative smoothing is applied on it.

This operation preserves the high spatial frequency details. Finally, the third processor MB3 performs a mean filtering which consists of very simple method used for noise reduction where the pixel to be processed is replaced by the average value of its neighbors. In this first programming version, image is initially stored in the local memory before filtering operation are applied. The second programming version is performed in a pipelined way where image lines are sent from a processor to another as soon as the previous processor has finished its work on it. Obviously, this type of execution makes us save a significant amount of time and memory which are often the major constraints for embedded systems in general and for our platform in particular. Indeed, performing this task in a pipelined way allows us to have a maximum of three image lines stored in the associated processor's memory rather than the whole image. The rest of the image lines will enter the FIFOs (FSLs) of their respective processors one by one. We remark that the three filtering operations have clearly different behaviours and use various arithmetic operators. Thus the execution time for each algorithm differs and hence involves an unequal network occupation. Data image is sent form MB0 to MB1 then from MB1 to MB2, MB2 to MB3 and finally from MB3 to MB0 in order to be transmitted to the host PC. This path is not the optimal existing path, but it has been chosen in order to get a different number of hops in each data transmission so we can examine the routing process in different scenarios.

5.2. Monitoring Results

We tested our monitoring methodology on an application implemented in two different ways as explained in subsection 5.1. Figure 8 and 9 respectively depict the execution of the filtering application in a non-pipelined and pipelined way.

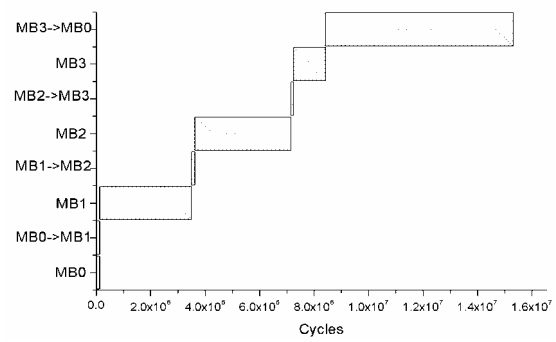


Figure 8 Without Pipeline

Figure 8, shows the non-pipelined aspect where communication and execution are clearly performed in a separate way. For all the communications, the FSL interruption, used for triggering the processor lecture process, were desactivated except for the processor MB0. That explains the fact that all interconnects communications times are about equal except for the communication time between the processor MB3 and MB0, showing the overhead of the interruption use. As stated earlier the communication times are about equal and not totally equal because there are two communication manners. The first passes through two routers (Two hops : MB0-MB1, MB2-MB3) while the second uses three routers (Three hops : MB1-MB2, MB3-MB0). The implementation was improved as shown in figure 9 where the execution recovers the communication process. This pipelined programming method allows us to meet a speed up of 2.14. Figures 10 and 11 give an exhaustive representation of all FSLs usage and occupation during the application runtime.

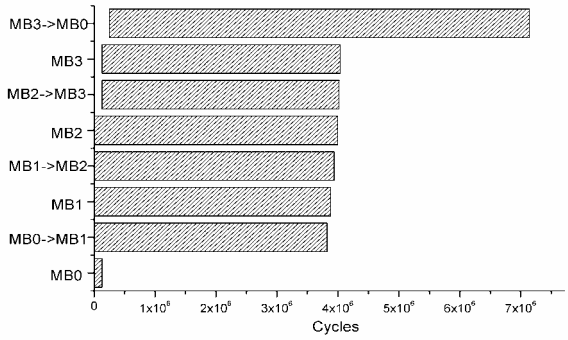


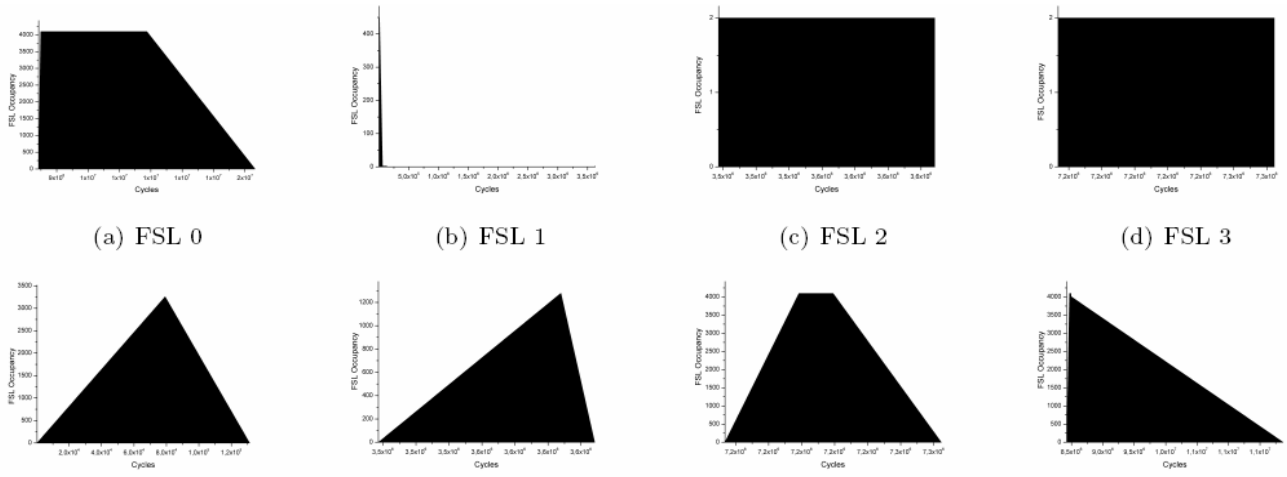
Figure 9 Pipeline

In this work we fixed a maximum FSL depth so we can see the maximum depth that is reached for each FSL IP regarding to the maximum FPGA resources. Figure 10 represents FSL occupancy for the non-pipelined version of the filtering application. We can have a global view of the whole FSLs platform behaviour by performing performance analysis on it. Figure 12 and 13 give respectively the average and coefficient of variance of the FSL depth. These results can guide the system designer to choose the most suitable depth for each FSL or to decide of an appropriate design space exploration. The second information concerns packets routing time. Table 3 gives the maximum and minimum time value for packets transmission. Packets that are sent form MB1 to MB2 and

MB3 to MB0 take longer transmission time than those transmitted from MB0 to MB1 and MB2 to MB3. This is due to the fact that they pass through an additional router to be transmitted. Slave FSLs regarding the microblaze processors (from where processors read incoming packets) are FSL0, FSL 1, FSL 2 and FSL3. Average occupancy in first version for these FSLs vary form 2 to 448 except for FSL 0 where its average occupancy value is 3053. This significant difference is mainly due to the fact that only MB0 uses interruption to read its incoming packets. In this same programming version of the filtering application, processors read data from their respective FSLs until they empty them out. This explains the fact that these FSLs do not fill. The best example is given by the FSL 1, where its occupancy in the first programming version does not exceed 448 while it reaches 4095 in the second version where the processor intermittently communicates and execute the filtering process. In the second version, FSL 2 and FSL3 should have the same behaviour as FSL 1 by reaching high occupancy values. The difference is due to the incoming packet speed in these FSLs which is related to the packet sent from the upstream processor. This processor sends one image data line at a time followed by performing filtering, except for the last data send where it sends 2 data image lines at a time. This is shown, in figure 13 where coefficient of variance for FSL 2 and FSL3 passes form 0.8 to 4 between the two versions. This same aspect appears in FSL 5, FSL 6 and FSL 7. From FSL occupancy traces in figure 10 and 11, we observe that the pipelined filtering application makes us save a significant execution time (Speed up of 2.14) and memory since average maximum value for FSL occupancy in the first version is 2158 and 1494 for the pipelined one.

6. Impact on NOC partial dynamic reconfiguration

The above data collected through hardware NOC monitoring can be exploited for numerous purposes either statically or dynamically. One specific point which can be observed is the suboptimal use of FIFO resources. Indeed, some FIFO might experience some peak utilization (triangular shape) and most of the time being under this peak value. On the other hand other FIFO may need scarce on chip memory resources when other FIFOs are underutilized. Partial dynamic reconfiguration of FIFO sizes could help match FIFOs dynamic behaviour needs under constant on chip memory budget. On Xilinx platform FPGA partial dynamic reconfigurability (PDR) is possible under some stringent place and route constraints.



(a) FSL 0 (b) FSL 1 (c) FSL 2 (d) FSL 3

(e) FSL 4 (f) FSL 5 (g) FSL 6 (h) FSL 7

Figure 10 Data traces on FSLs : 1St Programming Version

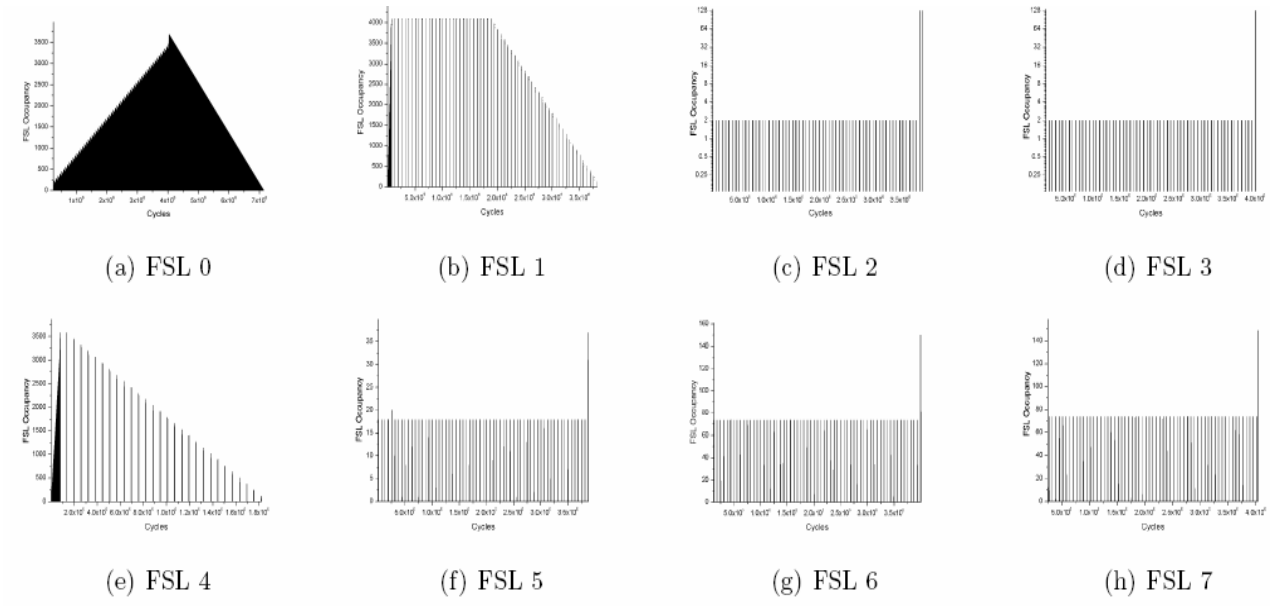


Figure 11 Data traces on FSLs : 2nd Programming Version

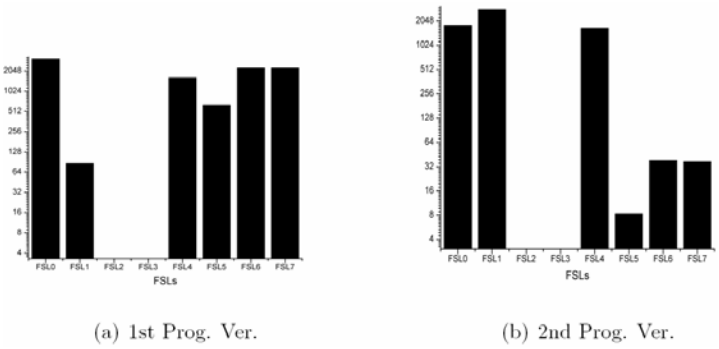


Figure 12 Average FSL values

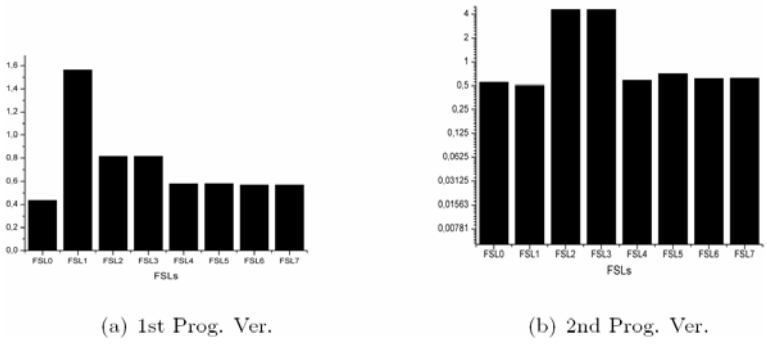


Figure 13 Coefficient of variance

Source	Desti.	Min (Cyc)	Max (Cyc)	BW(Mbit/s)
MB0	MB1	44	71	13.08
MB1	MB2	54	90	12.91
MB2	MB3	44	71	13.12
MB3	MB0	54	86	7.60

Table 3 Routing Time

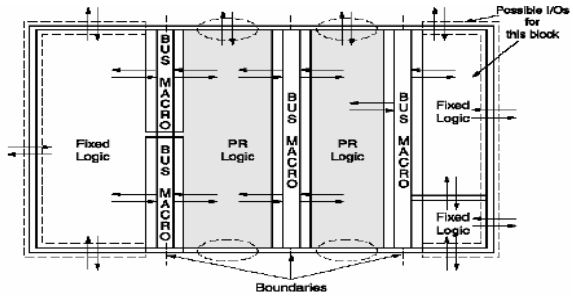


Figure 14 Xilinx device PDR floorplan

Xilinx FPGA chips impose full device height tile-based (4 slices) PDR and therefore the area to be dynamically reconfigured should be carefully planned in order to exploit optimally this potential **[27, 28]**. The question is then what are the relationships and tradeoffs between routers, FIFO sizes (flits number) and tiles? In other words for a given maximum on chip memory budget and a maximum FIFO size how many routers can be placed on PDR zone?

The following figure describes such tradeoffs.

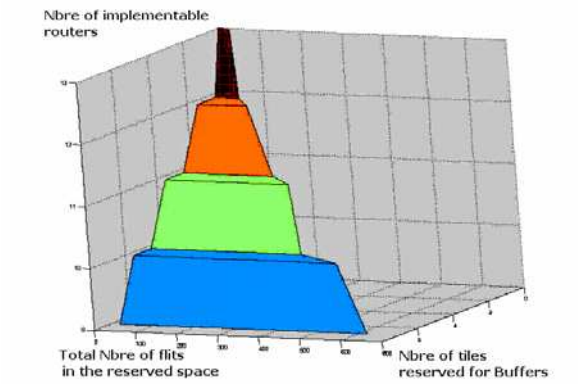


Figure 15 NoC Router PDR Tradeoffs

It appears clearly from this figure that increasing FIFO sizes (number of flits) will reduce the number of routers which can be implemented. Also increasing the number the FIFO size will increase the number of tiles to be used which as a ripple effect question the optimal use of the logic inside these tiles but not of course the optimal use of on chip memory. However, the key feature of this analysis is that this function is a stair based function which means that for given stair level i.e. a constant number if routers the number of tiles remains constant. This means that the coarse granularity of tile based PDR becomes an advantage since on chip memory dynamic regular trade between various routers can occur at a certain number of routers without requiring modification of the general structures (i.e. tiles) of the NOC. This key feature allows for full exploitation of NOC monitoring based PDR of NOC routers. The extension of this appears in the number of routers. The figure describe such tradeoffs as 13 routers with minimum PDR effort and minimum FIFO sizes and 9 routers with maximum FIFOs sizes and maximum PDR effort. Increasing the number of routers by 50 % (9 to 13) requires changing the topology of the NOC and is not appropriate with regular NOC. It can however, makes sense for irregular NOCs where direct routing capabilities are more important than routers and links buffering. This open problem will be the scope of future research.

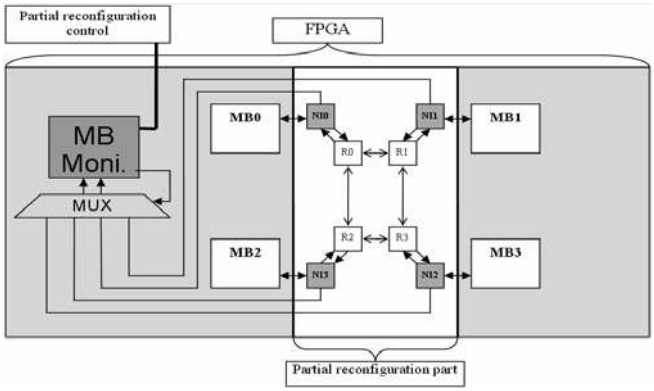


Figure 16 NoC monitoring and PDR

In the frame work of the partial dynamic reconfiguration presented in Figure 16, the monitoring Microblaze will have to manage two tasks. The first is the platform monitoring while the second task concerns the control of the partial dynamic reconfiguration once the monitoring is stopped.

7. Impact on NOC Design Methodologies

Network-on-chip monitoring can be used beneficially in different contexts. In the first context, network-on-chip monitoring is available as a permanent resource, the same way as hardware counters are available in high performance processors. In this case, the main application is for parallel programmers to fine tune their applications with real time feedback as hardware counters do. The implicit assumption is that the multiprocessors systems on chip have been designed as a platform for platform based design [21] and therefore the network-on-chip have been designed for a target set of applications. In the second application context closer to embedded systems, the systems on chip architecture is a parameterized platform for which a design space exploration is required for the specification of the various computation (embedded processors characteristics) and communication (network-on-chip characteristics) parameters. Previous work [22, 23] on multiobjective design space exploration of multiprocessor systems on chip did not propose an application specific approach for the specification of the networks on chip parameters intervals besides a priori settings. This paper fills this gap by suggesting links parameters intervals after actual execution monitoring. It is then possible to further reduce the design space exploration without affecting the quality of results, resulting in a substantial gain in the design time. In this framework, parallel programmers may explore various parallel computation-communication patterns implementations based on the feedback of the design space exploration results of the complete platform.

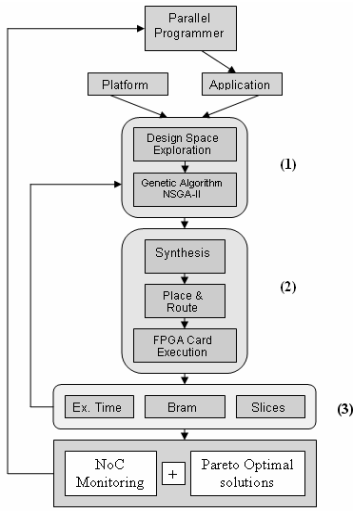


Figure 17 Parallel programming framework with design space exploration (Area/Performance) and NoC

This is possible in the framework of an FPGA based system [22] since as clearly emphasized in [24] prohibitive multiprocessors system-on-chip simulation time make it impossible for parallel programmers to get a quick feedback of their coding decisions. Finally, recent work [25, 26] suggests the usefulness of automatic network-on-chip synthesis from parallel applications. In both cases parallel programmers codes are used but an eventual feedback is not planned for fine tuning the parallel applications.

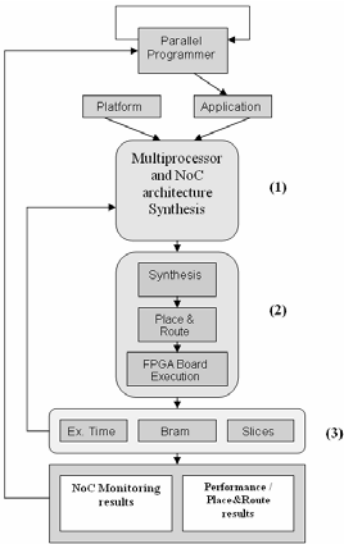


Figure 18 Parallel programming framework with MPSoC synthesis (Area/Performance) and NoC monitoring

This paper again fills this gap by providing potential NOC solutions resulting from the parallel applications with the addition of NOC monitoring. Parallel programmers are therefore in a better position to modify their codes.

8. Conclusion

Multiprocessors systems on chip efficient parallel programming requires fine grain real time non intrusive monitoring of both computation and communication. Increasingly diverse and large network-on-chips used for multiprocessors systems-on-chip communication, challenge this requirement and leave parallel programmers without help for fine tuning their code. We propose in this paper a scalable network-on-chip monitoring scheme providing real time feedback on networks links activity allowing parallel programmers to explore various parallel computation-communication patterns. We demonstrated the validity of our approach by implementing a 2x2 mesh based embedded multiprocessor on a large scale single chip FPGA device with the additional monitoring hardware. A 20% area penalty is expected for a 2x2 mesh while it reduces to 7% for a 4x4 mesh emphasizing its scalability for up to medium size multiprocessors sytems on chip. Future work will consider potential energy consumption tradeoffs with area requirements on actual devices.

References

[1] A.A.Jerraya and W.Wolf. Multiprocessor Systems-on-Chips. Morgan Kaufmann, June 2004.

[2] L. Benini and G. De Micheli. Networks on chips: A new soc paradigm. In Computer, pages 70–78, Jan 2002.

[3] W. Dally and B. Towles. Route packets not wires: On-chip inttercnnection networks. In Design Automation Conference, pages 684–689, 2001.

[4] F.Moraes, N.Calazans, A.Mello, M.Leandro, and L.Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. Integr. VLSI J., 38(1):69–93, 2004.

[5] S. Kumar and al. A network on chip architecture and design methodology. In Proc. Symposium on VLSI, editor, Proc. Symposium on VLSI, pages 117–124, Apr 2002.

[6] J.-Y.; Nollet V.; Marescaux T.; Verkest D.; Vernalde S.; Lauwereins R.; Bartic, T.A.; Mignolet. Highly scalable network on chip for reconfigurable systems. In IEEE International Symposium on System-on-Chip, pages 79–82, Nov 2003.

[7] F.Moraes, N.Calazans, A.Mello, L.Moller, and L.Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. Integr. VLSI J., 38(1):69–93, 2004.

[8] D.Bertozzi, A.Jalabert, S.Murali, R.Tamhankar, S.Stergiou, L.Benini, and G.De Micheli. Noc synthesisflow for customized domain specific multiprocessor systems-on-chip. IEEE Transaction on Parallel andDistributed Systems, Vol. 16:113–129, Feb 2005.

[9] L.Tedesco, A.M.D.Garibotti, N.Calazans, and F.Moraes. Traffic generation and performance evaluationfor mesh-based nocs. In 18th ACM Symposium on Integrated Circuits and Systems Design, pages 184–189,2005.

[10] M.; Ivanov A.; Saleh R. P.P.Pande; Grecu, C.; Jones. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. In IEEE Transactions on computers, volume 54, pages 1025– 1040, Aug 2005.

[11] G.M.; Kocarev L.; Hegedus, A.; Maggio. A ns-2 simulator utilizing chaotic maps for network-on-chip traffic analysis. In IEEE International Symposium on Circuits and Systems, ISCAS., volume 4, pages 3375–3378, May 2005.

[12] L.Noordergraaf and R.Zak. Smp system interconnect instrumentation for performance analysis. In Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, pages 1–9, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[13] M.Martonosi, D.Ofelt, and M.Heinrich. Integrating performance monitoring and communication in parallel computers. In SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pages 138–147, New York, NY, USA, 1996. ACM Press.

[14] N. Genko, D. Atienza, G. De Micheli, L. Benini, J.M. Mendias, R. Hermida, and F. Catthoor. A novel approach for network on chip emulation. In Circuits and Systems, 2005. ISCAS 2005. IEEE, volume 3, pages 2365–2368, May 2005.

[15] C.Ciordas, T.Basten, A.Radulescu, K.Goossens, and J.V.Meerbergen. An event-based monitoring service for networks on chip. ACM Trans. Des. Autom. Electron. Syst., 10(4):702–723, 2005.

[16] Xilinx. Available on: http://www.xilinx.com/.

[17] R. Benmouhoub, I. Aouadi, and O. Hammami. System on programmable chip plateform based design of jpeg-2000 entropy coder. In Workshop on synthesis and system integration of mixed information technologies (SASIMI'04), pages 103–106, Oct 2004.

[18] I. Aouadi, R. Benmouhoub, and O. Hammami. System on a programmable chip oriented jpeg-2000 entropy implementation for multimedia embedded systems. In The International Conference on Consumer Electronics (ICCE), Jan 2005.

[19] Xilinx. Xilinx fast simplex link ip. Available on: http://www.xilinx.com/.

[20] Alpha-Data. Adm-xrc-ii pci mezzanine card. Available on: http://www.alpha-data.com/adm-xrc-ii.htm.

[21] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. L. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform–based design. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, December 2000.

[22]R.Benmouhoub and O.Hammami. Mocdex: Multiprocessor on chip multiobjective design space exploration with direct execution. In The 13th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), pages 254–261, Apr 3-4 2006.

[23] R.Benmouhoub and O.Hammami. System-level design methodology with direct execution for multiprocessors on sopc. In IEEE Seventh International Symposium on Quality Electronic Engineering Design, pages 781–786, Mar 2006.

[24] R.Benmouhoub and O.Hammami. Multiprocessor on chip : Beating the simulation wall through multiobjective design space exploration with direct execution. In 5th IEEE International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (IPDPS), Apr 25-29 2006.

[25] R.Benmouhoub and O.Hammami. Mocsoc: Multiprocessor on chip synthesis from occam. In The 13th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), pages 246–253, Apr 3-4 2006.

[26] W.H.Ho and Pinkston T.M. A design methodology for efficient application-specific on-chip interconnects.In IEEE Transactions on Parallel and Distributed Systems, volume 17, pages 174 – 190, Feb 2006.

[27] XAPP 290 (v1.2) : Two flows for partial reconfiguration : Module based or difference based. 09/09/2004. http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf

[28] UG012 : « Virtex-II Pro and Virtex-II Pro X FPGA User Guide » (v4.0) 23 March 2005 http://www.xilinx.com/bvdocs/userguides/ug012.pdf

Résumé :
L'augmentation continue de la capacité d'intégration d'une part, la complexité croissante des applications embarquées d'autre part, ont conduit aux systèmes sur puce (SoC) puis aux systèmes multiprocesseurs sur puce (MPSoC). Le problème fondamental associé à ces systèmes sur puces de grande taille est celui des méthodologies de conception et de la crise de productivité en résultant ne permettant pas d'exploiter de manière efficace ces circuits. Cette crise de productivité est le résultat d'approches ad-hoc et manuelle de la conception alors que le problème doit être posé comme un problème d'optimisation multi-objectif dont la résolution doit faire appel à des techniques d'optimisation automatique. Dans cette thèse, nous présentons une méthodologie de conception pour les systèmes multiprocesseurs sur circuits logiques programmables, dont l'originalité porte sur trois aspects : (1) l'exploration évolutionnaire multi objectif de l'espace de conception afin de mener une recherche intelligente, (2) l'utilisation des circuits logiques programmables de grande taille pour l'évaluation rapide par émulation largement supérieure à la simulation., et enfin (3) l'utilisation de la synthèse MPSoC depuis un langage de programmation parallèle haut niveau (Occam) et de la prise en compte du monitoring sur puce. Des cas d'études sur circuits ont démontré l'efficacité d'une telle méthodologie pour résoudre le problème de la crise de productivité de la conception.

Abstract :
The continuous increase of capacity integration on one side, the exponential increase of embedded application complexity on the other side led to system on chip (SoC) then to multiprocessor system on chip (MPSoC). The fundamental problem associated with the large scale system on chip is the one of design methodologies and the resulting design productivity crisis preventing efficient exploitation of these circuits. This design productivity crisis is the result of ad-hoc and manual design while the problem should be established as a multiobjective optimization problem which should be solved by automated optimization techniques. In this PhD thesis, we present a design methodology for multiprocessor systems on chip implementation on programmable circuits, whose originality is based on three main points: (1) multiobjective evolutionary algorithm based exploration of the design space in order to undertake an intelligent solution search, (2) the use of programmable logic circuits for fast performance evaluation through emulation which is far superior to simulation and finally (3) the use of MPSoC synthesis from a high level parallel programming language (Occam) and a monitoring on chip framework. Case studies on circuit's implementations prove the efficiency of such a design methodology to solve the problem of the design productivity crisis.