Architectures des Accélérateurs de Traitement Flexibles pour les Systèmes sur Puce

Pascal Benoit

ACADEMIE DE MONTPELLIER

UNIVERSITE MONTPELLIER II
- SCIENCES ET TECHNIQUES DU LANGUEDOC -

THESE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE MONTPELLIER II

Discipline : Génie Informatique, Automatique et Traitement du Signal
Formation Doctorale : Systèmes Automatiques et Microélectroniques
Ecole Doctorale : Information, Structures, Systèmes

présentée et soutenue publiquement

par

Pascal BENOIT

Le lundi 11 octobre 2004

ARCHITECTURES DES ACCELERATEURS DE TRAITEMENT FLEXIBLES POUR LES SYSTEMES SUR PUCE

JURY

CAMBON Gaston	Professeur, LIRMM, Université Montpellier II	Président du jury
ROBERT Michel	Professeur, LIRMM, Université Montpellier II	Directeur de thèse
DEMIGNY Didier	Professeur, R2D2, Université Rennes I	Rapporteur
GARDA Patrick	Professeur, LISIF, Université Pierre et Marie Curie Paris VI	Rapporteur
SENTIEYS Olivier	Professeur, R2D2, Université Rennes I	Examinateur
SASSATELLI Gilles	Chargé de Recherche CNRS, LIRMM, Université Montpellier II	Examinateur
TORRES Lionel	Maître de Conférences, LIRMM, Université Montpellier II	Membre invité

« Le progrès a encore des progrès à faire »
P. Meyer

REMERCIEMENTS

Ce document présente les travaux que j'ai effectués durant trois années au LIRMM et je tiens à remercier en tout premier lieu M. Michel Habib, directeur du laboratoire, ainsi que tout le personnel administratif et technique, pour m'avoir permis de réaliser cette thèse dans d'excellentes conditions. Je remercie également les permanents du département microélectronique et leurs représentants, M. Michel Renovell et M. Pascal Nouet, notamment pour la bonne ambiance de travail qui règne au sein de cette unité de recherche.

Je remercie M. Didier Demigny et M. Patrick Garda d'avoir accepté d'être les rapporteurs de cette thèse, ainsi que M. Olivier Sentieys pour sa participation au jury.

Je remercie mon directeur de thèse M. Michel Robert, pour la confiance qu'il m'a accordée depuis le début. Merci également à M. Gaston Cambon qui a participé à l'encadrement de ces travaux. Je tiens à remercier chaleureusement mon encadrant Lionel Torres, pour sa disponibilité, son écoute et sa bonne humeur. Aussi, je remercie sincèrement Gilles Sassatelli pour sa participation active à ce projet et sans qui ce sujet de thèse n'aurait vu le jour. Merci également à Thierry Gil pour son aide sur la carte de prototypage.

Je tiens également à remercier M. Juergen Becker, ainsi que l'ensemble du personnel de son laboratoire, à l'Université de Karlsruhe, d'avoir accepté de m'accueillir pour mon stage post-doctoral.

Je voudrais adresser toute ma sympathie aux collègues que j'ai eu le plaisir de côtoyer au cours de ces quatre années, la promo Olivier (rendez-vous en Allemagne!), Régis, Alain, les mélomanes Laurent, Serge, Mariane, Ben et Wence, les colocataires du box David, Jean-Denis, Daniel et Alex, les sportifs (Foot, Baby-foot et autres ...) Arnaud, Abou, Norbert, Seb, Fabrice, Luigi, Julien, ainsi que tous les anciens et nouveaux qui se reconnaîtront !

Enfin, un grand merci à toutes les personnes qui partagent ma vie personnelle, qui m'ont encouragé et soutenu durant ces trois années. Je pense évidement à Sylvie, à mes parents, ma sœur, ma famille proche et mes amis, montpelliérains, nîmois, lillois, niçois et parisiens…

RESUME

Les systèmes sur puce intègrent sur un même substrat de silicium l'ensemble des organes nécessaires à la prise en charge des différentes fonctionnalités du système. Pour la partie dédiée aux traitements numériques, le microprocesseur central est souvent déchargé des applications critiques (traitement du signal et des images en général) par un accélérateur de traitement. C'est par rapport à l'architecture du coprocesseur que se pose la problématique de cette thèse. En effet, de nombreuses approches sont possibles pour ce dernier, et vouloir les comparer s'avère être une tâche complexe. Après avoir fait un état de l'art des différentes solutions architecturales de traitement flexibles, nous proposons un ensemble de métriques dans une optique de caractérisation. Nous illustrons alors notre méthode par la caractérisation et la comparaison d'architectures représentatives de l'état de l'art. Nous montrons que c'est par une exploitation efficace du parallélisme que les coprocesseurs peuvent améliorer significativement leurs performances. Or, malgré de réelles aptitudes, les accélérateurs ne sont pas toujours capables de tirer parti de ce potentiel. C'est pour cela que nous proposons une méthode générale de multiplexage matériel, qui permet d'améliorer les performances par l'exploitation dynamique du parallélisme (boucle et tâches). Par son application à un cas concret, un système baptisé Saturne, nous prouvons que par l'adjonction d'un contrôleur dédié au multiplexage matériel, les performances de l'accélérateur sont quasiment doublées, et ce avec un faible surcoût matériel.

GLOSSAIRE

AAA	Adéquation Algorithme Architecture
ALU	Arithmetic-Logic Unit
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuit
CAO	Conception Assistée par Ordinateur
CDFG	Control Data Flow Graph
CDMA	Code Division Multiple Access
CISC	Complex-Instruction-Set computer
CMOS	Complementary Metal-Oxide Semiconductor
CFB	Configurable Functional Block
CLB	Configurable Logic Block
CPLD	Complex Programmable-Logic Device
CPU	Central Processing Unit
CW	ConfigWare
DCT	Discrete Cosine Transform
DFG	Data Flow Graph
DMA	Direct-Memory Access
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor (ou Processing)
EDGE	Enhanced Data GSM Environment
F	Farad
FFT	Fast Fourier Transform
FIFO	First-In, First-Out
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
FSM	Finite State Machine
GOPS	Giga-Operations Per Second
GPP	General Purpose Processor
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
HCDFG	Hierarchical Control Data Flow Graph
HDL	Hardware Description Language
HW	HardWare
I/O	Input/Output
ILP	Instruction Level Parallelism
IIR	Infinite Impulse Response
IP	Intellectual Property
JPEG	Joint Photographic Experts Group
LLP	Loop Level Parallelism
LUT	Look Up Table
MAC	Multiplication-ACcumulation
MEF	Machine à Etats Finis
MIMD	Multiple-Instruction, Multiple-Data

MIPS	Million Instructions Per Second
MOPS	Million Operations Per Second
MPEG	Moving Picture Experts Group
NOC	Network On Chip
OS	Operating System
PC	Personal Computer
PDA	Personal Digital Assistant
RAM	Random Access Memory
RISC	Reduced-Instruction-Set Computer
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SDRAM	Synchronous DRAM
SIMD	Single-Instruction, Multiple-Data (array)
SIP	System In a Package
SMT	Simultaneous Multi-Thread
SPARC	Scalable Processor ARChitecture
SOC	System On a Chip
SRAM	Static RAM
SW	SoftWare
TDMA	Time-Division Multiple Access (communications)
TSI	Traitement du Signal et des Images
UAL	Unité Arithmétique et Logique
UMTS	Universal Mobile Telecommunication System
VGA	Video Graphics Adapter
VHDL	VHSIC (Very High Speed Integrated Circuits) HDL
VLIW	Very Long Instruction Word
VLSI	Very-Large-Scale Integration

SOMMAIRE

INTRODUCTION GENERALE

INTRODUCTION

En 1971, l'histoire des circuits intégrés va être marquée par l'apparition d'un composant révolutionnaire. Cette année là, Intel met au point le premier microprocesseur, le 4004, comportant 2300 transistors et gravé dans une technologie PMOS de 8µm de longueur de canal, fonctionnant à une fréquence de 108kHz. En 2004, on estime le nombre de circuits intégrés en utilisation à 25 milliards dans le monde. A titre d'exemple, le Pentium IV Extreme Edition (cœur Prescott) contient 125 Millions de transistors en technologie CMOS 90 nm [Inte04], pour des fréquences de fonctionnement au-delà de 3 GHz. Ces quelques chiffres illustrent à eux seuls la gigantesque dynamique de recherche et de production qui s'est mise en place en à peine plus de trente ans.

Qui aurait pu imaginer cela à en 1971 ? Et bien, un homme, Gordon Moore, cofondateur d'Intel en 1968, avait prédit que les capacités d'intégration sur les technologies silicium doubleraient tous les 2 ans. Les fréquences de fonctionnement augmentant avec la densité d'intégration, la puissance de traitement offerte par mm² de silicium croît, théoriquement, de manière exponentielle (figure 1, courbe 2). Or les processeurs, qui sont basés sur des principes architecturaux antérieurs aux premiers circuits intégrés, ne permettent pas d'exploiter la puissance de traitement exigée par la complexité algorithmique actuelle (figure 1, courbe 1). Bien évidemment, bénéficier de ce potentiel est attractif, et pour répondre à des besoins algorithmiques croissants, des approches alternatives sont nécessaires. Un exemple actuel d'application concerne les réseaux de téléphonie mobile. Une illustration est le très médiatique UMTS qui a pour vocation d'intégrer tous les réseaux de génération actuels en un seul et de lui adjoindre des capacités multimédia. L'émission d'un signal par le biais d'un tel système nécessite des traitements multimédia (annulation d'écho, bruit…), un codage de source (AMR, EFR ...), un codage de canal (Viterbi, turbo code), et un accès au canal (TDMA, CDMA…).

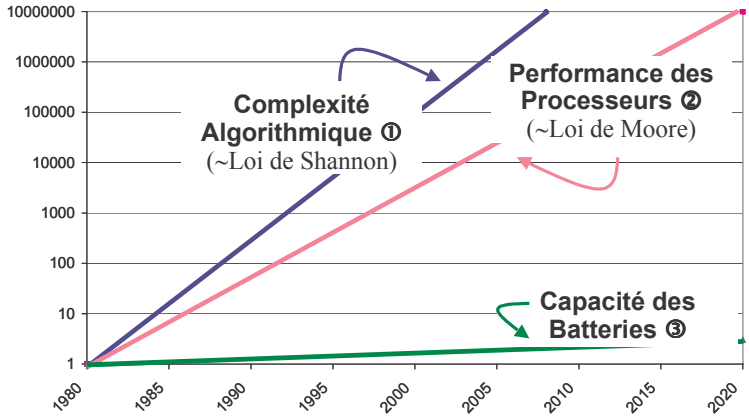


FIG. 1 - *Aperçu du problème de l'évolution des nouvelles technologies*

Parallèlement, des contraintes plus spécifiques concernant notamment le marché de l'embarqué, mettent en relief des limitations préoccupantes notamment en ce qui concerne la consommation d'énergie. On ne pourra pas, *a priori*, compter sur les progrès des méthodes de stockage chimique de l'énergie si les tendances de la figure 1 (courbe 3) se confirment.

Dans le cadre de cette thèse, nous nous limiterons à un cadre d'applications critiques par rapport aux exigences qu'elles nécessitent en termes de performances essentiellement dans le domaine du TSI. Le spectre de ces applications est relativement large : elles permettent, entre-autres, de réaliser des filtrages (FIR, IIR), des codes correcteurs d'erreurs (Turbo code), des

compressions de sons (MP3), d'images (JPEG), des compressions de flux vidéos (MPEG). Ces applications nécessitent un nombre important de calculs et une grande quantité de transferts de données (utilisation importante des mémoires). De plus, elles ont souvent des contraintes d'exécution en temps réel.

PROBLEMATIQUE

Grâce à l'augmentation des densités d'intégration, la quasi-totalité des fonctions des systèmes électroniques peut être intégrée sur un même substrat de silicium : on appelle ce type de composant un Système sur Puce (SoC). Cette thèse s'inscrit dans un contexte de prise en charge des applications TSI en temps réel dans les SoC.

La granularité des applications implique une large variété de solutions allant des architectures reconfigurables à grain fin jusqu'aux circuits dédiés. La figure 2 représente un exemple de conception de SoC hétérogène multi-grain. Cependant, la nature des applications considérées (TSI) nécessite essentiellement des traitements au niveau mot. C'est dans ce contexte que sont apparues, dans la littérature, un certain nombre d'architectures reconfigurables à grain épais comme le Systolic Ring [Sass01]. C'est par opposition à la granularité des circuits FPGA que l'on qualifie le grain de ces architectures reconfigurables. Les travaux concernant ces composants n'en étant qu'à un stade relativement précoce, nous avons dans un premier temps contribué aux validations sur des applications de l'état de l'art et mis en évidence les forces et les faiblesses du Systolic Ring. Nous avons alors apporté un certain nombre d'évolutions pour améliorer sa structure initiale.

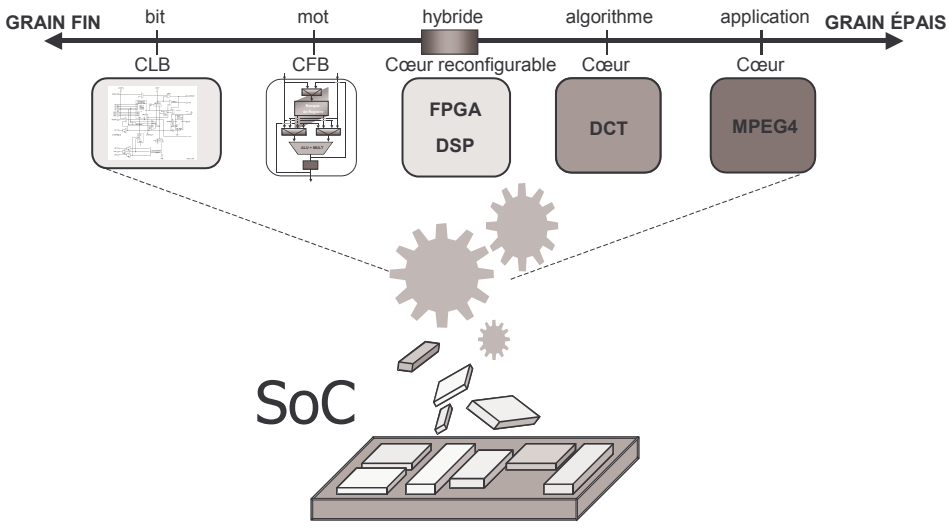


FIG. 2 - *Exemple de SoC hétérogène*

Face à la multiplication des contributions telles que le Systolic Ring dans le domaine, une réflexion sur la caractérisation et la comparaison de ces architectures a dû être abordée dans un premier temps. Même si les premiers résultats obtenus laissent penser que l'on a de bonnes raisons de croire en ce type d'accélérateur, quelle est l'objectivité des métriques utilisées ? Les résultats faisant souvent intervenir des paramètres technologiques, qu'est-ce qui permet d'affirmer que telle architecture est « meilleure » qu'une autre? Autrement dit, comment juger les compromis spécifiques d'architectures intrinsèquement différentes ? Comment positionner une architecture comme le Systolic Ring par rapport aux autres

architectures de l'état de l'art ? Comment répartir alors les différentes applications dans un système sur puce *multi-grain* ?

Ensuite, si ce type d'architecture se révèle être effectivement une alternative pour l'accélération des applications du TSI, que peut-on espérer de l'amélioration des performances ? Comment estimer la *scalabilité* de ce type d'architecture ? Ces composants visent notamment, à exploiter un maximum de parallélisme, mais jusqu'où peut-on espérer aller ? Existe-t-il une limite, architecturale ou algorithmique, à l'accélération ? Le développement d'outil de compilation pour ce type d'accélérateur est une perspective ambitieuse ; mais comment appréhender ce problème et celui de l'exploitation du parallélisme pour les générations futures de composants ou encore la compatibilité binaire du logiciel ? A partir du moment où les contraintes temps-réel sont respectées, est-ce que l'accélération des applications a toujours un sens ?

CONTRIBUTIONS

C'est à la problématique développée ci-dessus que nous allons tenter de répondre dans ce manuscrit. Celui-ci fait la synthèse de travaux que nous avons accomplis durant ces trois années au sein du LIRMM. Le contexte d'étude regroupe trois axes de recherche de la conception des SOC : il est représenté par la figure 3 et ces travaux de thèse se situent à l'intersection de ces trois axes.

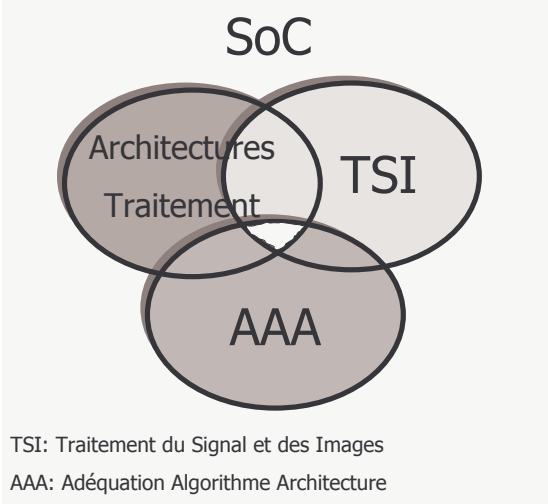


FIG. 3 - *Contexte d'étude de la thèse*

Dans ce contexte, la première contribution de cette thèse sera un état de l'art non exhaustif des architectures de traitement. Nous analyserons notamment les techniques de conception qui permettent d'améliorer les performances à travers l'exploitation des différents niveaux de parallélisme. Nous étudierons plus particulièrement des exemples aboutis dans le domaine, tels que les processeurs superscalaires multiflots, les DSP VLIW ou encore des architectures reconfigurables de granularités différentes. A l'issue de cet état de l'art, **nous établirons un premier bilan qualitatif des aptitudes de chacune des approches afin de formuler les premières hypothèses de notre travail**. Nous présenterons également dans le détail l'architecture du Systolic Ring[Sass01], en insistant sur les évolutions apportées par rapport à la structure initiale.

Suite à cet état de l'art, nous présenterons alors la deuxième partie de ce travail, à savoir les **métriques pour les architectures de traitement flexibles**. Après avoir rappelé les efforts de caractérisation notamment au niveau logiciel ou extrinsèque, nous proposerons un ensemble d'outils dédiés à la caractérisation intrinsèque des architectures d'accélérateurs

flexibles. Nous donnerons un ensemble de résultats qui illustrent l'utilisation de ces métriques dans le cadre de l'accélération matérielle des applications TSI, dans un contexte systèmes sur puce. Nous analyserons ces résultats afin de donner des éléments de réflexion pour l'utilisation efficace du parallélisme dans le cadre d'architectures reconfigurables à grain épais.

Enfin, c'est dans le prolongement de ces deux contributions que nous arriverons à la dernière partie de notre travail, à savoir l'utilisation efficace des différents niveaux de parallélisme pour l'accélération des applications TSI. Nous proposons à cet effet une méthode originale basée sur un **multiplexage matériel des ressources permettant d'améliorer significativement les performances initiales**. Pour valider notre concept, nous avons réalisé une architecture baptisée *Saturne*, basée sur le Systolic Ring et l'unité de multiplexage matériel. L'intérêt de cette approche est démontré par les résultats obtenus. Cet intérêt est renforcé par le fait que cette méthode pourrait simplifier la programmation de l'architecture.

PLAN DU MEMOIRE

Ce document s'articule en quatre parties.

Dans le premier chapitre, nous présenterons le contexte général de ces travaux concernant la conception des Systèmes sur Puce. Nous rappellerons leur flot d'intégration et l'ensemble des critères de performance à considérer lors de l'élaboration de leur cahier des charges. Nous poursuivrons notre étude par une analyse des différentes architectures de traitement et leurs modèles d'exécution. C'est à partir de cet état de fait que nous justifierons le choix des trois contributions de cette thèse, notamment la définition de métriques de caractérisation ainsi que la mise en œuvre d'une méthode d'exploitation dynamique du parallélisme.

Le deuxième chapitre sera consacré à un état de l'art des architectures de traitement. Nous décomposerons ce chapitre en deux parties, en étudiant d'abord les processeurs puis les architectures reconfigurables. Nous ferons alors le bilan des différentes approches et nous présenterons l'architecture du Systolic Ring. C'est à partir de l'analyse de l'existant que nous proposerons alors les orientations de la suite de ce mémoire.

Nous poursuivrons ce mémoire par la définition, dans le chapitre 3, d'un ensemble de métriques. Nous chercherons à répondre dans un premier temps, au problème de la caractérisation des accélérateurs de traitement flexibles. Les résultats obtenus nous permettront alors de comparer les différentes architectures existantes et de tirer des conclusions sur leurs performances potentielles. Ces métriques seront également utilisées pour caractériser en fonction de l'application, l'architecture la plus adéquate. Dans le but de caractériser la *scalabilité* des architectures paramétrables, nous proposerons également une méthode visant à étudier l'évolution des métriques en fonction de leurs paramètres.

Le quatrième chapitre sera consacré à la définition puis la mise en œuvre du concept de multiplexage matériel. Nous proposerons pour cela une architecture, baptisée Saturne, capable d'exploiter dynamiquement le parallélisme au niveau boucle et tâche. Nous exposerons nos résultats, en reprenant les métriques du chapitre 3, afin de mesurer le gain effectif apporté par cette nouvelle approche.

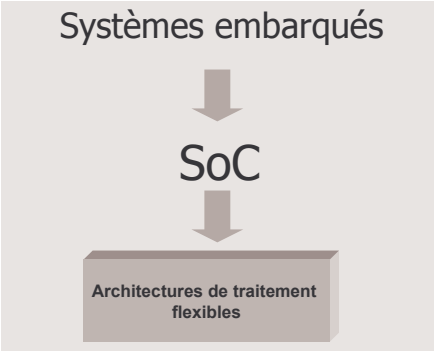
Pour finir, nous présenterons nos conclusions par rapport à la problématique et nous proposerons des perspectives à de nouveaux travaux de recherche dans le domaine.

Chapitre 1

SYSTEMES SUR PUCE ET ARCHITECTURES DE TRAITEMENT

INTRODUCTION

Pour commencer ce chapitre, nous présenterons le contexte général de ces travaux concernant la conception des Systèmes sur Puce. Nous rappellerons leur flot d'intégration et l'ensemble des critères de performances à considérer lors de l'élaboration de leur cahier des charges. Nous poursuivrons notre étude par une analyse des différentes architectures de traitement et leurs modèles d'exécution. C'est à partir de cet état de fait que nous justifierons le choix des trois contributions de cette thèse, notamment la définition de métriques de caractérisation ainsi que la mise en œuvre d'une méthode d'exploitation dynamique du parallélisme.



SOMMAIRE DU CHAPITRE

1. LES SYSTEMES SUR PUCE

Nous présentons dans cette première section le contexte de cette thèse en exposant pour commencer une vue d'ensemble des systèmes microélectroniques sur silicium. Nous nous intéressons ensuite à la méthodologie de conception et d'intégration avant de terminer sur les critères de performances de tels systèmes.

1.1. Intégration des systèmes sur puce

Le terme SoC ou système sur puce désigne une classe de circuits intégrés actuels. Ces composants permettent d'intégrer sur un même substrat de silicium l'ensemble des blocs que l'on trouvait auparavant sur une carte. Les intérêts sont multiples (miniaturisation accrue, meilleures performances etc.) et ont donc largement contribué à pousser dans cette voie la majorité des fabricants de semi-conducteurs.

Pour concevoir rapidement, valider et fabriquer un SOC, les industriels ou universitaires proposent des plate-formes pour lesquelles des bibliothèques d'IP permettent de personnaliser, suivant les besoins, le compromis de performances. Pour cela, on réutilise des blocs pré-conçus et validés, afin de réduire les temps de mise sur le marché du produit final. Il existe des blocs de natures différentes : ils permettent soit de traiter de l'information (macro-blocs), soit de connecter les macro-blocs entre-eux.

1.1.1. Eléments d'un Système sur puce

a. Macro-blocs

Ce sont des cœurs de processeurs, des cœurs dédiés, des mémoires, des cœurs reconfigurables, des MEMS[1], des systèmes RF[2], etc. Parmi les macro-blocs CMOS, on peut référencer plusieurs types suivant le degré de personnalisation possible. Les *Soft-cores* sont par exemple des blocs décrits dans un langage de description matériel synthétisable : ils sont indépendants de toute technologie et sont donc très souples d'utilisation. Les *Firm-cores* sont des blocs ayant subis une optimisation en surface et en vitesse par des techniques de placement relatif. Décrits en HDL structurel, ils font appel à des composants élémentaires d'une librairie générique. Cette implantation autorise des évaluations plus fines de ressources utilisées et de performances. Les « *Hard cores* » sont quant-à eux optimisés et placés-routés pour une technologie donnée. Le bloc est donc implicitement caractérisé en termes de performances et de surface.

b. Interfaçage et connexion des macro-blocs

Il s'agit le plus souvent de bus, mais compte-tenu de leur inefficacité lorsque le nombre de macro-blocs augmente, on voit apparaître de plus en plus de structures inspirées des réseaux de télécommunication : les NOC ou réseaux sur puce. En termes de réalisations, nous pouvons citer le bus Sonics [Soni04] basé sur des techniques de TDMA et la norme OCP, le réseau SPIN [Guer00] et la norme VCI ou encore le réseau Hermès [Riso04]. Dans une approche plus classique, on trouve le fabricant ARM avec ses bus hiérarchiques AMBA [Arm04].

A noter cependant qu'une alternative aux SOC est possible : elle se base sur une *encapsulation* verticale des cœurs, dans un boîtier. Cette approche est appelée SIP. Là où dans un SOC on répartit sur un même plan de masse l'ensemble des cœurs, dans un SIP on va

[1] MEMS: Micro Electro-Mechanical System
[2] RF : Radio-fréquences

empiler les cœurs les uns sur les autres. Cette solution pourrait peut-être se développer plus massivement dans les années à venir car elle pourrait s'avérer plus simple à mettre en œuvre.

1.1.2. Exemple de plate-forme SOC

La figure 4 illustre un exemple de plate-forme actuelle : il s'agit du système OMAP[Omap04] basé sur des cœurs des sociétés ARM et TI. Au niveau des contributions universitaires dans le domaine, nous pouvons citer l'initiative française SOCLIB [Socl04] qui a pour but de fournir une bibliothèque d'IP décrites en SystemC [Syst04] et normalisée VCI.

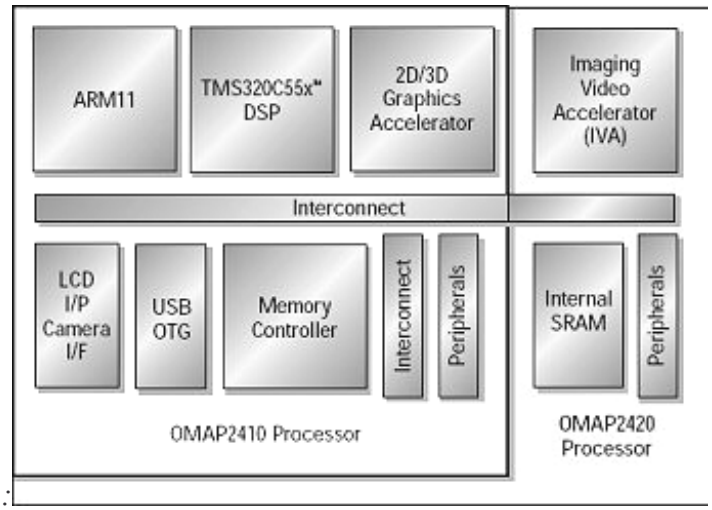


FIG. 4 - *La plate-forme OMAP 24x. Elle est principalement constituée d'un processeur embarqué ARM11, d'un DSP C55 et d'un accélérateur graphique. Il est possible également de connecter un second accélérateur dit IVA pour Image Video Accelerator.*

1.1.3. Flot de Conception d'un SOC

Le cahier des charges d'un système va spécifier la ou les applications ciblées et les performances souhaitées. Pour faciliter la conception de ces systèmes, les outils de CAO offrent des logiciels qui permettent la description et la simulation des circuits à différents niveaux d'abstraction. De plus, la complexité accrue des systèmes a naturellement tendance à augmenter le temps de conception et donc l'automatisation de certaines phases permet de limiter le *time to market*. Historiquement, les premiers outils remontent aux années 1980. Jusque là, les masques des circuits étaient dessinés « à la main ». Par un principe basé sur la réutilisation d'éléments matériels, au début des blocs de portes simples, puis de blocs complexes pré-caractérisés jusqu'aux cœurs de composants actuellement, la complexité des systèmes n'a jamais cessé de croître.

Actuellement, on peut représenter le flot de conception de ces systèmes par la figure 5. Le point crucial de ce flot est le choix de l'architecture de traitement, car c'est elle qui va largement fixer le compromis de performances.

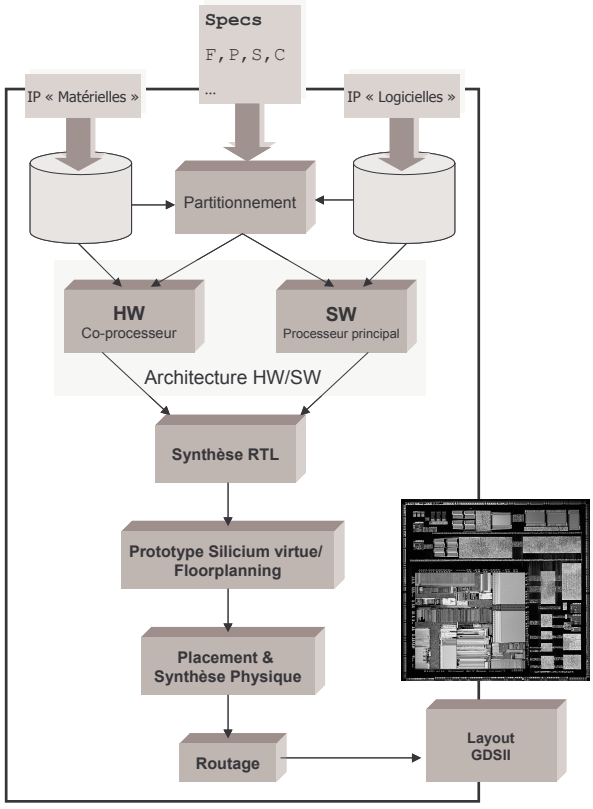


FIG. 5 - *Flot de conception simplifié d'un Système sur Puce. On s'intéresse ici en particulier à la conception conjointe d'un système basé sur un processeur central et un ou plusieurs co-processeurs. A partir des spécifications et de la plate-forme de conception (bibliothèques d'IP), le concepteur crée une ou des partitions de l'application en répartissant les tâches dites logicielles ou matérielles. L'architecture processeur/co-processeur est ensuite co-simulée et co-vérifiée à chaque étape du flot.*

1.2. Les Critères de performance des systèmes

Un des aspects fondamentaux d'un système est qu'il fonctionne correctement suivant les performances fixées lors des spécifications. Les performances d'un SoC peuvent être analysées suivant plusieurs critères que nous nous proposons d'expliciter ici.

1.2.1. Les performances temporelles

Ce premier critère concerne tous les « aspects temporels » d'un système. Ceux-ci peuvent être de plusieurs ordres. D'un point de vue global, on peut dire qu'un système A est n fois plus rapide qu'un système B pour une tâche donnée si :

$$\frac{\text{Temps exécution B}}{\text{Temps exécution A}} = n$$

Le **temps d'exécution** est donc le premier élément que l'on observe pour évaluer la performance d'un système. Le temps d'exécution peut s'écrire comme une fonction de la **latence** et de la **cadence**. Pour une tâche donnée, la latence L correspond au temps mis par le système entre l'acquisition d'une entrée et la production de la première sortie. La cadence C caractérise le rythme auquel le système produit chacune de ses sorties (le nombre d'échantillons produits par seconde). Le nombre d'échantillons traités est N. Suivant ce que l'on cherche à observer, le temps d'exécution peut également se référer à plusieurs tâches. Ainsi, la cadence peut correspondre au nombre de tâches traitées à la seconde.

$$\text{Temps exécution(secondes)} = L + \frac{N}{C}$$

Dans les systèmes temps réel, les traitements pris en charge sont contraints par des échéances de traitement. Le contrôle de processus industriels ou encore les systèmes de sécurité sont des applications des systèmes temps réels et font partie des nombreuses cibles potentielles des processeurs enfouis. Pour respecter les contraintes de temps réel, il est nécessaire qu'une tâche soit traitée dans un temps inférieur ou égal à celui imposé par l'échéancier. Il est souvent nécessaire, dans ce cas là, de diminuer la latence et/ou d'augmenter la cadence de traitement du système. Ces paramètres dépendent souvent de la manière dont l'architecture exploite le parallélisme.

1.2.2. La consommation

Le terme consommation regroupe les dissipations de puissance et d'énergie. Lors de la phase de conception, la consommation d'énergie est étudiée pour dimensionner les batteries du système (durée de vie)[Raba00]. Celle-ci est bien sûr liée à la puissance consommée par l'application mais également au temps d'exécution. Ensuite, l'analyse de la consommation de puissance permet d'anticiper sur les dissipations thermiques, pour concevoir les circuits de refroidissement (surtout pour les applications embarquées où le système de refroidissement requiert une surface, une masse et un coût non négligeables). La puissance dynamique est composée de 2 termes, un premier pour la puissance statique (courants de fuite) et un autre pour la puissance dynamique (commutations du transistor), calculé de la manière suivante :

$$P(W) = \alpha * C_{eq} * F * Vdd^2$$

La puissance dynamique est directement fonction du taux de commutation du circuit α, de la capacité équivalente du circuit C_{eq}, de la fréquence de fonctionnement **F** et de la tension d'alimentation V_{dd} au carré. D'une part, la capacité équivalente des circuits et les fréquences de fonctionnement ont plutôt tendance à augmenter. D'autre part, les tensions d'alimentation diminuent avec l'augmentation de la finesse de gravure. Ce paramètre intervenant au carré dans la formule précédente, l'effet de la diminution des tensions d'alimentation a donc un impact important. Toutefois, comme le montre le Tableau 1, cette diminution n'est pas assez importante pour contrebalancer l'augmentation de la consommation due aux autres paramètres et due aux courants de fuite (puissance statique).

Tab. 1. *Evolution de la consommation des familles de processeurs INTEL et AMD [Bur01].*

Processeur	*Technologie (µm)*	*# Transistors (10^6)*	*V_{DD}(Volts)*	*Fréquence (MHz)*	*Puissance crête (Watts)*
INTEL					
386SX	1	0,275	5	33	2
486DX	0,8	1,2	5	50	5
P5	0,8	3,1	5	66	16
P6	0,35	5,5	3,3	166	29,4
AMD					
K5	0,35	4,3	3,5	120	14
K6	0,35	8,8	3,2	233	28,3
Athlon	0,25	22	1,6	700	50

La consommation reste un point critique puisqu'elle engendre une dissipation thermique qui n'est pas toujours facile à évacuer. Le coût des systèmes de refroidissement, leur taille ainsi que leur poids constituent autant de handicaps pour des applications portables. En outre,

l'échauffement des composants provoque également un vieillissement prématuré des composants[Raba96]. Enfin, les capacités des batteries évoluant beaucoup plus lentement que le besoin de puissance des applications (cf. Introduction Générale), la prise en compte de ce critère parait indispensable.

1.2.3. La flexibilité

On entend par flexibilité la possibilité d'évolution ou d'adaptation d'un système. Il n'est pas toujours possible de prévoir toutes les applications qu'un système devra prendre en charge. Un critère indirect de performance est donc la capacité d'un système à pouvoir s'adapter à diverses applications. En effet, cette flexibilité peut permettre de prolonger la durée de vie d'un système en le faisant évoluer vers de nouvelles applications, nouveaux standards ou nouvelles normes qui apparaissent sur le marché.

1.2.4. Les coûts

Les coûts sont aussi, indirectement, des facteurs de performance. Il est évident qu'un système aussi rapide qu'un autre mais moins onéreux sera considéré comme une solution plus attractive : nous pouvons alors considérer qu'il est indirectement plus performant. Le coût est fonction de nombreux éléments et il est donc particulièrement difficile de se faire une idée réaliste du coût réel du circuit final : la surface de silicium, le rendement de fabrication, le packaging, l'effort de conception, le coût de la programmation, le temps de mise sur le marché, la réutilisation, le test… sont autant de facteurs influençant le coût final du produit.

2. LES ARCHITECTURES DE TRAITEMENT

Le maillon central de tout système numérique est son architecture de traitement. Afin de respecter l'ensemble des performances que nous venons de rappeler, il est nécessaire d'apporter une attention toute particulière au choix de cette architecture. Celle-ci sera en général dépendante des applications ciblées par le système. Il existe en fait différentes familles d'architectures, différents modèles de traitement ainsi que diverses possibilités de couplage suivant le partitionnement de l'application : c'est ce que nous présentons dans cette deuxième partie.

2.1. Familles d'architectures de traitement

Une fois numérisées, les données sont stockées en mémoire et manipulées par ce qu'on appelle de manière générale, un *processeur*. A l'origine, les processeurs sont basés sur un principe nommé « paradigme de Von Neumann »[von45]. A l'origine, il s'agissait d'un composant plutôt volumineux destiné, dans un ordinateur ou une autre machine, à interpréter et à exécuter des instructions. Les processeurs ont ensuite été intégrés sur des puces de silicium, d'où le nom de *microprocesseur*.

Aujourd'hui, il existe trois familles de composants pour effectuer les traitements numériques. Elles sont présentées sur la figure 6.

Le choix de l'architecture pour un système va dépendre de l'ensemble des critères fixés lors de la spécification. La solution dédiée ASIC permet d'atteindre les meilleurs temps de traitement pour un minimum de surface et de puissance consommée. De plus, les coûts initiaux relativement importants de ces circuits peuvent être amortis si le nombre de pièces fabriquées est élevé. Cependant, ces architectures étant dédiées, elles manquent de flexibilité. Pour cette raison les concepteurs préfèrent parfois se tourner vers des solutions plus souples comme les microprocesseurs ou les architectures reconfigurables. Ces approches possèdent en outre l'avantage de réduire le temps de conception.

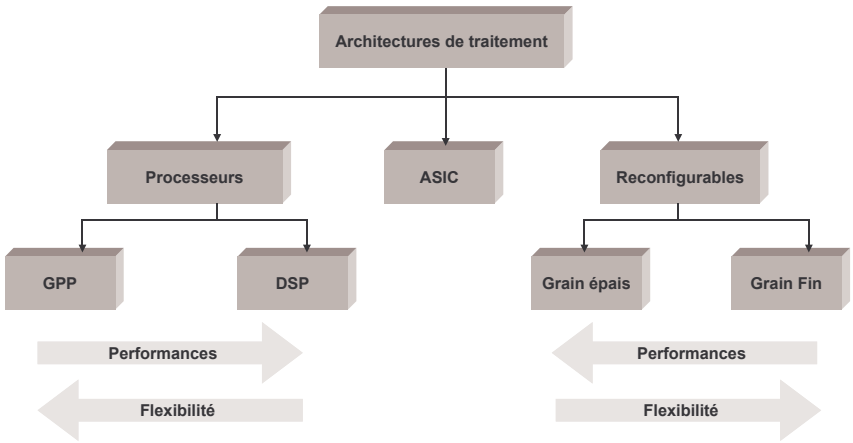


FIG. 6 - *Trois approches pour le traitement des données numériques. La spécialisation des opérateurs augmente les performances. Pour une application donnée, la solution optimale sera donc l'approche dédiée ou ASIC. Selon la classe d'applications, les contraintes de performances et de flexibilité, le concepteur a la possibilité de choisir l'une de ces approches.*

2.2. Modèles de traitement

Chaque famille d'architecture est basée sur un modèle de traitement : la figure 7 illustre deux de ces modèles. Le premier (SW), pour les processeurs, est basé sur une exécution séquentielle de l'algorithme à réaliser alors que le second (HW), pour les circuits dédiés,

permet de réaliser une projection spatiale du motif de calcul. Un troisième modèle pour les architectures reconfigurables étend le modèle HW en lui adjoignant une dimension temporelle (CW[1]). Nous exposons ici ces différents concepts.

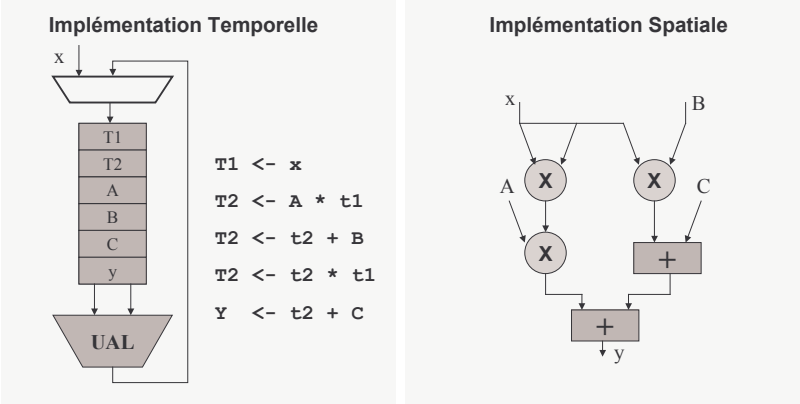


FIG. 7 - *Deux modèles de traitement. L'exemple choisi est un calcul de polynôme du second degré. La première solution réalise chaque opération en série en n'utilisant qu'un seul opérateur et nécessite donc à chaque cycle un stockage intermédiaire du résultat. La deuxième approche permet une répartition des différentes opérations *sur plusieurs opérateurs distincts.*

- Implémentation temporelle (SW)

Dans ce cas, le modèle de traitement est séquentiel. Les architectures basées sur ce modèle utilisent un programme qui définit les opérations à réaliser et les opérandes à aller chercher en mémoire. Elles n'utilisent en général qu'un faible nombre de ressources de calcul et des registres qui sont réutilisées dans le temps. Ces architectures sont généralement très flexibles, mais ne permettent pas toujours d'atteindre des performances élevées.

- Implémentation spatiale (HW)

Les circuits intégrés dédiés ASIC sont basés sur cette approche. Dans ce cas, afin d'exploiter le parallélisme de l'application, chaque opérateur existe en différents points de l'espace et traite des opérandes directement acheminés sur ses entrées. La multiplicité des ressources de calcul autorise leur spécialisation. Les circuits conçus suivant ce schéma affichent des performances potentiellement maximales pour une application donnée. Néanmoins, la flexibilité de ces circuits reste faible et se limite au changement de certains paramètres caractéristiques de l'algorithme implémenté.

- Implémentation « spatio-temporelle » (CW)

Les architectures reconfigurables sont également basées sur un modèle de traitement spatial et donc disposent de réelles aptitudes au parallélisme. Afin d'apporter une souplesse supérieure aux circuits dédiés, la fonctionnalité de ces architectures peut être modifiée dans le temps et dans l'espace. Cette caractéristique du modèle de traitement est une véritable valeur ajoutée car elle confère une flexibilité certaine au matériel qui peut dès lors potentiellement s'adapter à n'importe application.

2.3. Architectures mixtes

Dans le cadre de conception de systèmes, le concepteur n'est pas forcément limité à une seule famille d'architecture. Il est tout à fait possible d'envisager un système mixte où

[1] CW : Configware

plusieurs cœurs basés sur différents modèles de traitement se partagent l'exécution des applications. Les systèmes intégrant pour la plupart des processeurs classiques, cette approche consiste au départ à remédier aux insuffisances d'une exécution logicielle pure. Les parties critiques d'une application sont d'abord identifiées puis prises en charge par un cœur de traitement dédié ou reconfigurable, afin d'en accélérer le traitement. Dans une approche mixte, le partitionnement de l'application influence le niveau de couplage entre le processeur (CPU : Central Processing Unit) et le co-processeur [Rubi97]. La finesse du partitionnement va fixer le degré de communication nécessaire entre chaque partition.

3. CONTRIBUTIONS

A l'heure actuelle, le nombre de composants d'origine industrielle ou universitaire pour chaque famille d'architecture de traitement est très important. Ceci peut donner un ordre d'idée sur le nombre de combinaisons de couplage possibles. Dans le cadre de nos travaux, nous nous intéressons plus particulièrement aux applications qui nécessitent de lourds traitements et pour lesquelles les approches classiques telles que les processeurs généralistes ne sont pas efficaces. Nous avons évoqué précédemment que le modèle spatial permettait d'exprimer le parallélisme entre les opérations et c'est ce qui fait sa force. Savoir tirer parti du parallélisme est d'ailleurs l'objectif le plus souvent recherché des concepteurs. Cependant, comme nous l'avons vu, la consommation, la flexibilité et les coûts sont autant de critères supplémentaires à prendre en compte dans le contexte des systèmes sur puce.

L'objectif de ce mémoire consistera à analyser l'existant, caractériser chaque approche, comparer les différents modèles et enfin mettre en œuvre une méthode permettant d'améliorer l'exploitation du parallélisme. Afin de bien illustrer le contexte et la problématique, nous utiliserons à chaque étape de ce mémoire le flot présenté figure 8.

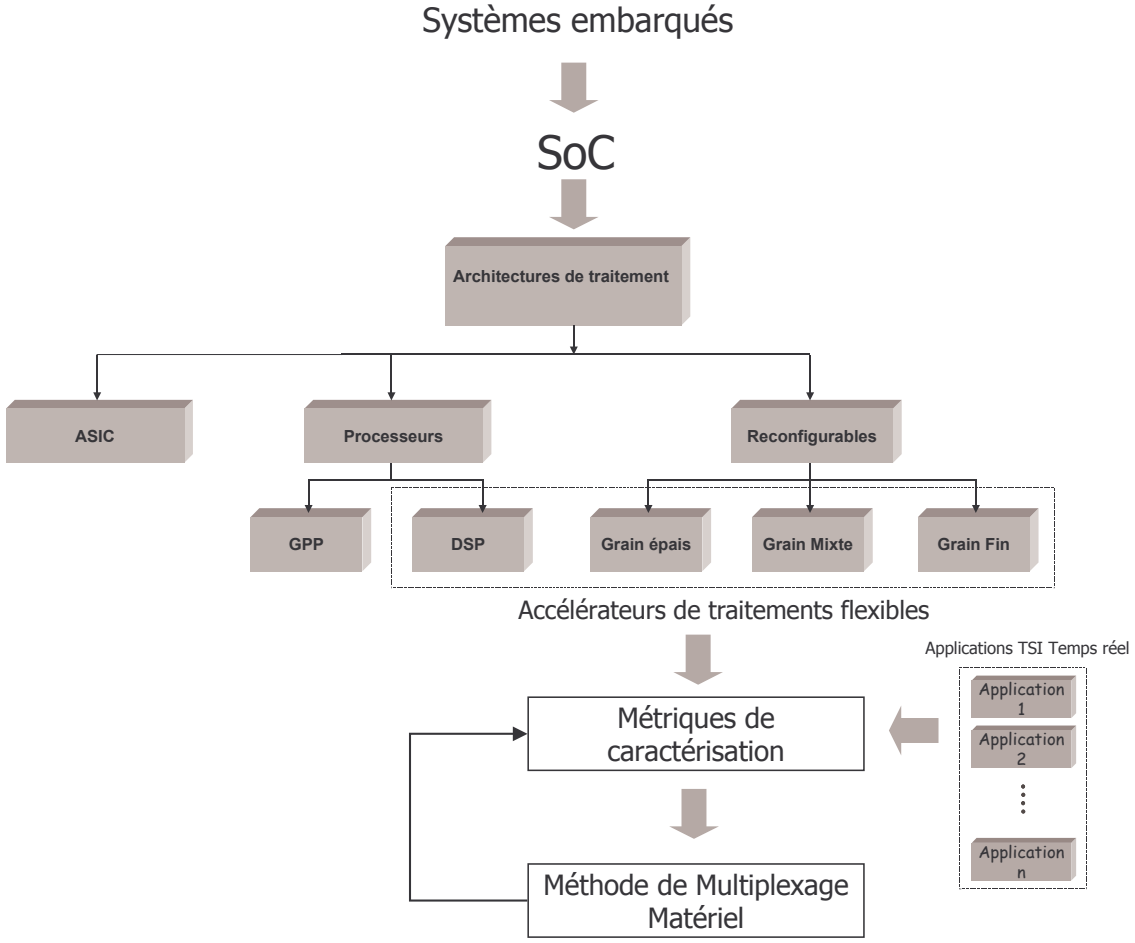


FIG. 8 - *Représentation des contributions de la thèse*

3.1. Etat de l'art des architectures de traitement flexibles

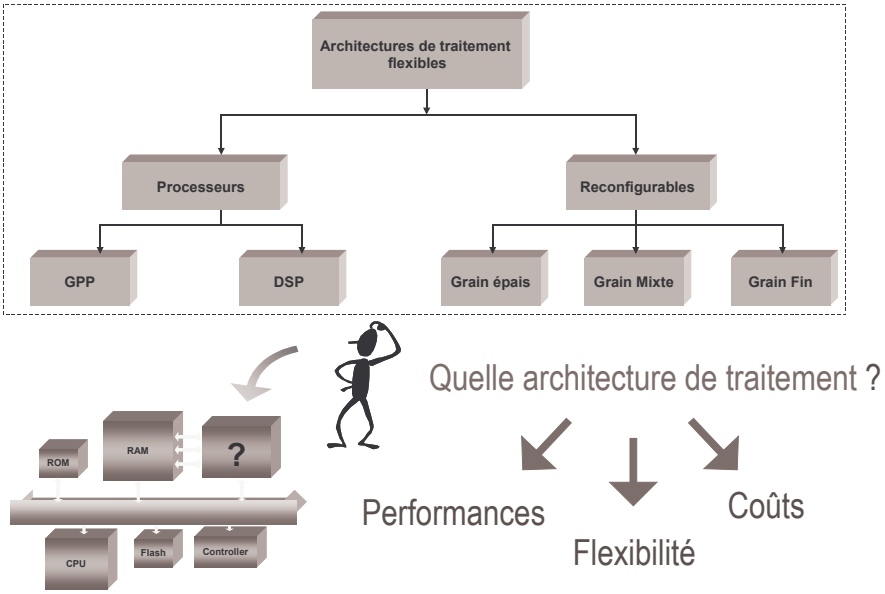


FIG. 9 - *Problématique de l'état de l'art. Dans une optique d'amélioration des performances, l'exploitation du parallélisme est souvent le moyen le plus utilisé. Nous proposons un état de l'art des différentes familles d'architectures de traitement et présentons les mécanismes architecturaux permettant de tirer parti du parallélisme.*

Nous poursuivrons ce mémoire par un chapitre consacré aux architectures de traitement flexibles (figure 9). Par rapport au champ d'applications ciblées, la technique utilisée pour améliorer les performances consiste le plus souvent à tirer parti du parallélisme. Lorsque, par exemple, deux opérations ne dépendent pas l'une de l'autre, il est possible de les traiter en parallèle. L'intérêt premier de la prise en charge du parallélisme est le gain de temps qu'il induit sur le temps d'exécution. Les applications prises en charge par les systèmes intégrés sont constituées de toutes sortes d'opérations (arithmétiques, logiques, chargement mémoire etc.) qu'il est parfois possible d'exécuter en parallèle. Pour établir cette possibilité, on étudie les dépendances de données entre les différentes actions, souvent à l'aide de graphes. Compte-tenu de ce parallélisme potentiel, il est alors possible d'envisager des moyens matériels qui permettent de l'exploiter. Ces moyens peuvent être spécifiques à une application donnée (ASIC) ou bien plus génériques et donc adaptés à un domaine d'application (accélérateurs flexibles *i.e.* DSP et architectures reconfigurables).

L'exploitation du parallélisme, d'un point de vue macroscopique ou microscopique, peut être envisagé à différents niveaux du système. Du point de vue de l'utilisateur, lorsqu'un système est capable d'exécuter simultanément un logiciel de messagerie et un de traitement de texte, il est capable de traiter en parallèle deux processus. Or, si l'on y regarde de plus près, il s'agit souvent d'un multiplexage temporel des processus sur l'unité principale de traitement (CPU). C'est le système d'exploitation qui attribue, parfois par le biais de l'utilisateur, des slots temporels pour chaque processus, avec également la possibilité d'établir des priorités (du type temps réel). A ce niveau, on parlera de **parallélisme au niveau système**. De la même manière, lorsque le système d'exploitation va répartir des processus sur le CPU ou les accélérateurs de traitement, nous parlerons également de parallélisme au niveau système.

Il est cependant possible que le parallélisme entre deux processus, ou deux tâches, soit pris en charge par le même bloc matériel. Ainsi, récemment, on a beaucoup entendu parler des architectures multi-flots [Tul95] avec sa mise en œuvre dans un processeur commercial

(pentium IV HT). Dans ce cas, l'architecture de traitement est capable de gérer au moins deux contextes d'exécution, ce qui correspond à la prise en charge de deux tâches. A ce niveau là, nous parlerons de **parallélisme de tâche**.

L'idée de base de l'exploitation du parallélisme repose sur la duplication des unités de calcul afin d'avoir les moyens matériels de prendre en charge les opérations parallèles. Mais il est parfois impossible d'avoir autant d'opérations en parallèle que d'unités parallèles disponibles ce qui implique une sous-utilisation des ressources. Une technique consiste à utiliser le parallélisme de boucle, en mettant en œuvre des techniques de pipeline logiciel ou de déroulage de boucle. L'idée générale consiste à faire apparaître artificiellement du parallélisme à partir des différentes itérations d'une boucle donnée. On parlera à ce niveau de parallélisme au **niveau boucle**.

Lorsqu'on regarde encore plus finement les applications, on en arrive aux opérations de base. A ce niveau, il est alors possible d'exploiter le **parallélisme d'opération**. Il se manifeste de deux façons essentiellement : le *parallélisme spatial* (deux opérations s'exécutent en même temps et n'ont pas de dépendance de données), et le *parallélisme temporel* ou pipeline (deux opérations s'exécutent en même temps, ont une dépendance de données, mais les opérateurs ne travaillent pas sur les mêmes partitions temporelles). Pour les processeurs, à ce niveau on parle aussi de parallélisme au niveau instruction (ILP : Instruction Level Parallelism).

La figure 10 représente un zoom du système vers les opérations de base en spécifiant à chaque niveau le parallélisme potentiel exploitable.

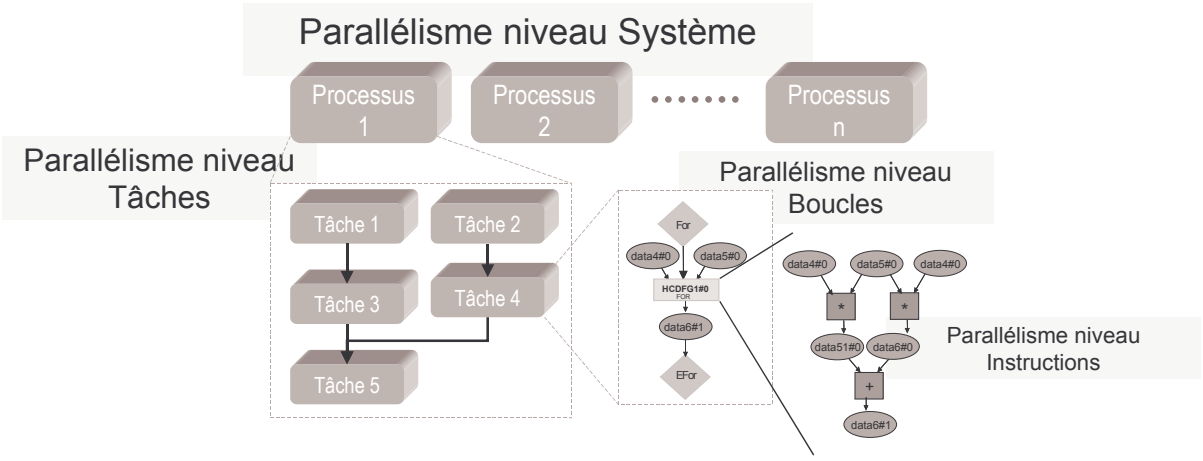


FIG. 10 - *Représentation schématique des différents niveaux de parallélisme : du système à l'opération.*

L'état de l'art de cette thèse consistera à faire une étude complète des solutions architecturales flexibles pour le traitement des données numériques. Nous établirons une synthèse des travaux réalisés dans le domaine des processeurs généralistes, puis spécialisés, ainsi que dans le domaine des architectures reconfigurables. Nous porterons une attention particulière aux techniques mises en œuvre pour exploiter les différents niveaux de parallélisme présentés ici. Pour terminer ce chapitre, nous dresserons un bilan qualitatif des performances et présenterons le support de nos travaux, un modèle d'architecture reconfigurable dynamiquement à grain épais, le Systolic Ring. Celui-ci nous servira de référence pour l'étude pour les travaux de caractérisation et de base de validation pour le concept de multiplexage matériel.

3.2. Métriques de caractérisation

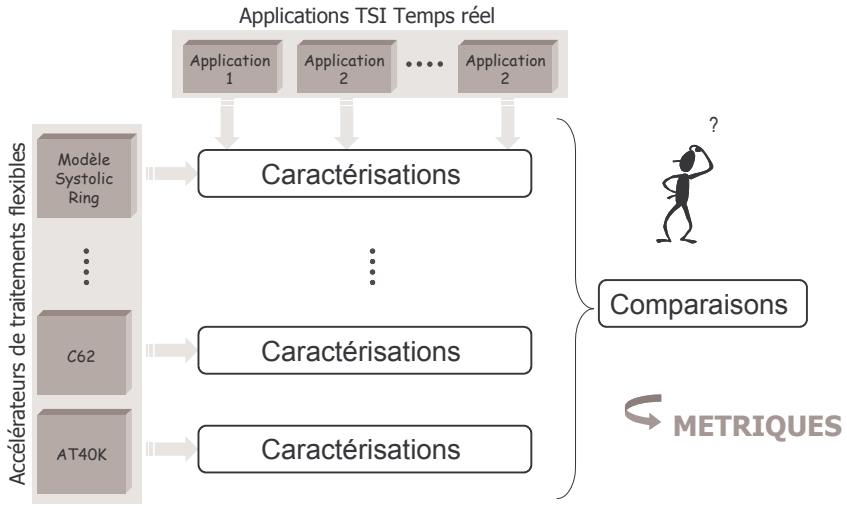


FIG. 11 - *La problématique de caractérisation et de comparaison. Nous proposons pour cela un ensemble de métriques intrinsèques.*

En considérant chaque famille d'architecture de traitement, ainsi que les différents niveaux de couplage, on peut imaginer toute l'étendue des solutions envisageables (figure 11) qu'un intégrateur système se devrait d'évaluer : dans la pratique ce n'est évidemment pas possible… Afin de faciliter la comparaison des architectures, nous allons définir un ensemble de métriques permettant de caractériser chaque architecture. Dans le chapitre 3, plusieurs métriques sont proposées à cet effet.

La première étape de ce travail consiste à définir des outils d'évaluation de performances basés sur les caractéristiques intrinsèques des architectures. D'autres travaux se basent sur une caractérisation des applications ou encore sur l'efficacité extrinsèque du système. A partir d'hypothèses faites sur les caractéristiques du modèle de calcul, nous tentons d'identifier les architectures pouvant, théoriquement, atteindre le meilleur compromis de performances. Après un ensemble de définitions, nous caractérisons un certain nombre d'architectures que nous comparons ensuite.

La deuxième partie de ce travail concerne la caractérisation de la *scalabilité* de modèles d'architectures génériques. Les densités d'intégration ne cessant de croître, l'utilisation de modèles génériques permet de réduire les temps de conception. Mais pour analyser la faisabilité des futures générations de circuits basées sur des modèles génériques, il faut être capable de prévoir les performances. Nous proposons dans cette optique une méthode d'utilisation de ces métriques à la caractérisation de modèles d'architectures reconfigurables à grain épais, en analysant la *scalabilité* de modèles d'architecture pour les futures générations de processus CMOS. Nous proposons une illustration pratique d'utilisation de cette caractérisation pour l'architecture du Systolic Ring.

3.3. Méthode d'exploitation dynamique du parallélisme

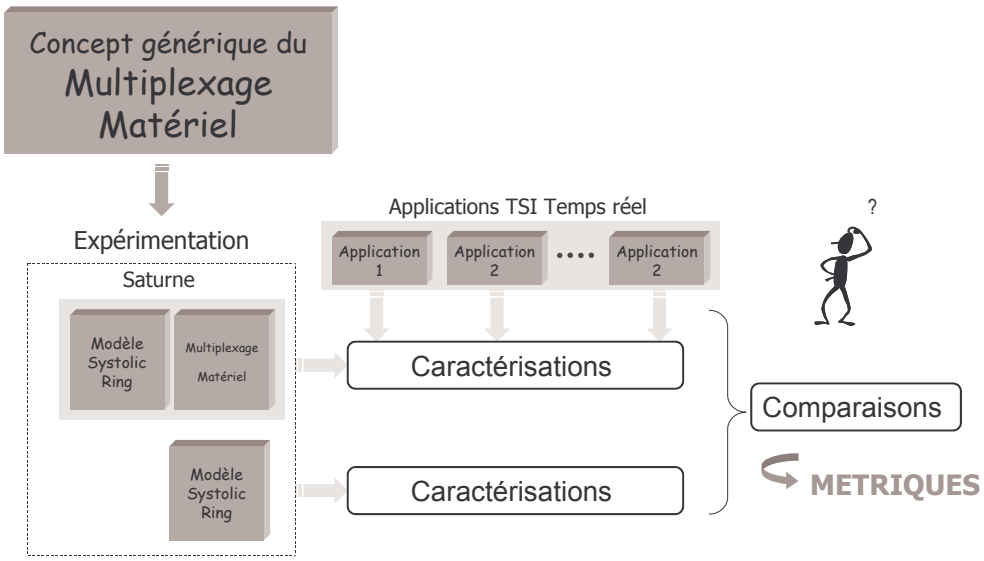


FIG. 12 - *Le multiplexage matériel pour améliorer l'exploitation du parallélisme et sa mise en œuvre avec le modèle du Systolic Ring.*

Si les architectures reconfigurables à grain épais, comme le Systolic Ring, possèdent de réelles vertus leur permettant potentiellement de surclasser les autres architectures de traitement, il faut leur fournir les moyens de pouvoir les exprimer. Ainsi, ces architectures semblent disposer de caractéristiques en adéquation avec les applications du TSI, comme nous le confirment les résultats du chapitre 3. Cependant, l'hypothèse faite sur l'exploitation maximale du parallélisme ne peut s'avérer exacte sans de solides méthodes permettant à ce type d'architectures d'utiliser efficacement leur potentiel opératif.

Nous proposons dans cette optique un concept simple et général appelé « multiplexage matériel » permettant à n'importe quelle architecture reconfigurable dynamiquement, de déployer un parallélisme de niveau boucle ou de niveau tâche afin d'augmenter l'utilisation des ressources de traitement (figure 12). Ce déploiement est réalisé à l'aide d'un contrôleur matériel dédié intervenant directement sur la configuration initiale. L'objectif est de rendre les architectures reconfigurables plus performantes et plus flexibles vis-à-vis de leur environnement. Afin de valider ce concept, nous proposons une réalisation baptisée Saturne, basée sur l'architecture du Systolic Ring et la mise en œuvre du multiplexage matériel. Les résultats du chapitre 3 et ceux obtenus par le multiplexage matériel permettront d'évaluer l'intérêt de cette nouvelle approche.

4. CONCLUSION

Nous avons présenté dans ce chapitre le flot d'intégration des systèmes sur puce, et les critères de performances à considérer lors de l'élaboration de leur cahier des charges. Nous nous sommes intéressés, plus particulièrement dans ce contexte, à l'architecture dédiée à la prise en charge des traitements numériques. Pour cela, différentes familles et modèles ont été présentés, et il a été montré que ceux-ci pouvaient conduire à différents compromis de performances.

Un des éléments critiques de la conception de l'architecture de traitement concerne l'accélération des applications consommant du temps CPU. Les applications TSI temps réel représentent la majeure partie des algorithmes ciblés. C'est par rapport à cette problématique d'accélération que se placent nos trois principales contributions : un état de l'art des architectures de traitement, la définition d'un ensemble de métriques intrinsèques pour la caractérisation et la comparaison d'architectures existantes, et enfin une méthode de multiplexage matériel pour l'exploitation dynamique du parallélisme.

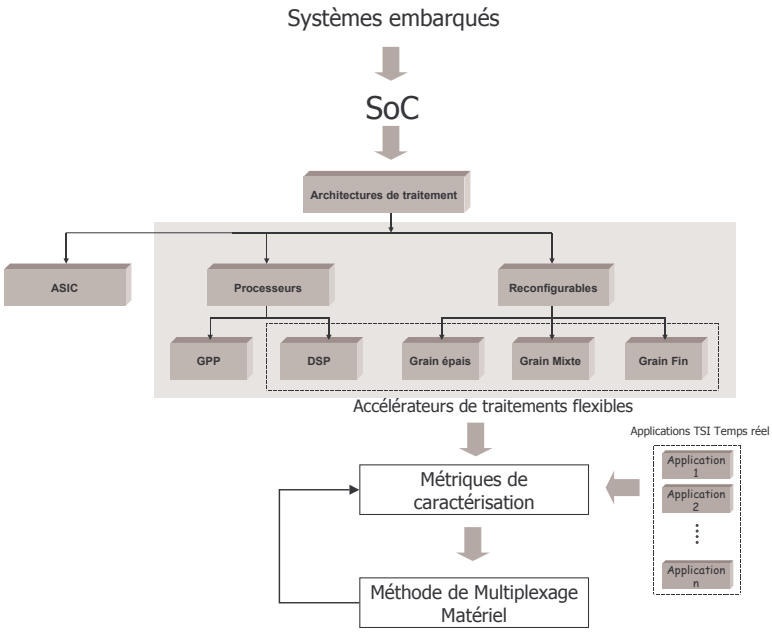
Nous poursuivrons ce mémoire par un état de l'art des architectures de traitement flexibles, en nous intéressant plus particulièrement aux techniques mises en œuvre pour exploiter toujours plus de parallélisme.

Chapitre 2

ETAT DE L'ART DES ARCHITECTURES DE TRAITEMENT FLEXIBLES

INTRODUCTION

Trois familles de composants permettent de traiter des données numériques : les processeurs, les architectures reconfigurables et les composants spécifiques. Nous nous intéressons aux deux approches les plus flexibles. Nous présentons à travers un état de l'art les principes permettant une exploitation efficace du parallélisme pour améliorer les performances. A l'issue de cette synthèse, nous établissons un bilan qualitatif de performances et nous présentons l'architecture du Systolic Ring, le support d'étude et de validation de cette thèse. C'est à partir de l'analyse de l'existant que nous proposons alors les orientations de la suite de ce mémoire.



SOMMAIRE DU CHAPITRE

1. ARCHITECTURES DES PROCESSEURS

Depuis 1971 et le 4004 d'Intel, 7 générations de processeurs se sont succédées. Le tableau 2 illustre les principales générations des processeurs grand public.

Tab. 2. *Evolutions technologiques des processeurs. Ce tableau illustre les différentes générations de processeur, la technologie silicium utilisée, le nombre de transistors sur une puce et la surface du composant.*

CPU	*Génération*	*Technologie*	*Nombre de transistors*	*Surface*
486	4	1.0 micron	1,200,000	79 mm^2
Intel Pentium	5	0.5 micron	3,100,000	161 mm^2
AMD K6	6	0.25 micron	8,000,000	68 mm^2
Intel Pentium II	6	0.35 micron	7,500,000	131 mm^2
AMD ATHLON	7	0.25 micron	22,000,000	184 mm^2
Intel Pentium III CuMine	6+	0.18 micron	28,000,000	106 mm^2
AMD ATHLON "Thunderbird"	7	0.18 micron	37,000,000	117 mm^2
Intel Pentium 4	7	0.09 micron	125,000,000	112 mm^2

Nous allons étudier dans cette section les principaux aspects des processeurs, en illustrant notre exposé d'exemples. Nous verrons dans un premier temps les principes de base de l'architecture des processeurs et des modèles de programmation. Nous avons choisi de présenter ensuite l'architecture des processeurs superscalaires et des processeurs VLIW qui illustrent deux des aboutissements majeurs des processeurs de cette famille.

1.1. Principes généraux

Les processeurs sont basés sur le principe d'une projection temporelle de l'algorithme à réaliser. Aux débuts de l'informatique, une seule unité de calcul prend en charge tout l'algorithme. On comprend alors que la seule limitation de cette approche révolutionnaire est le temps. En effet, l'application est découpée en une succession d'instructions qui sont ordonnancées et exécutées par le processeur. Dès lors, les concepteurs et les fabricants de processeurs n'ont cessé de mettre en œuvre des techniques qui ont permis d'améliorer les performances des processeurs.

Avant d'exposer ces procédés, nous allons revenir sur les principes généraux des processeurs : structure interne, architecture mémoire et pipeline. Nous exposerons ensuite deux classifications des processeurs suivant leur jeu d'instruction ou suivant la structure de leur flot de donnée(s) et d'instruction(s). Enfin, nous terminerons par les techniques actuelles utilisées pour exploiter plus de parallélisme.

1.1.1. Modèles d'architectures

Dans ce paragraphe, nous allons exposer les fondements de la conception des processeurs, en présentant d'abord la première architecture qui a été proposée par *John Von Neumann*. Nous verrons ensuite que les besoins en performances ont motivé la définition de modèles plus efficaces.

a. Architecture *minimale*

L'architecture minimale d'un processeur est constituée de deux éléments : le chemin de données et le contrôleur (figure 13).

Le **chemin de données** est constitué de l'unité arithmétique et logique servant aussi bien pour le traitement de l'algorithme lui-même que pour le calcul des adresses mémoires lors de l'accès à la mémoire. Il peut comporter également des registres d'usage général dans lesquels peuvent être stockées les données manipulées par le programme, de manière temporaire.

Le **contrôleur** contient le compteur du programme (PC) et le registre d'instructions (IR). A partir des instructions lues en mémoire, il séquence et contrôle toutes les opérations effectuées à l'intérieur du processeur, ainsi que les adressages de la mémoire.

De manière très simplifiée, l'exécution de chaque instruction se décompose en 4 phases:

- Cycle « *Fetch* » : une adresse de la mémoire est générée ce qui permet de stocker dans IR l'instruction à exécuter.
- Cycle « *Decode* » : le contrôleur génère à partir de l'instruction une séquence de micro-instructions nécessaires à l'exécution de cette instruction. Cette séquence commande les opérations dans le chemin de données ainsi que les chargements de registres, lectures/écritures en mémoire, opérations sur les entrées sorties, etc.
- Cycle « *Execute* » : l'instruction considérée est traitée dans le chemin de donnée.
- Cycle « *Write* » : le résultat du calcul est stocké dans la mémoire.

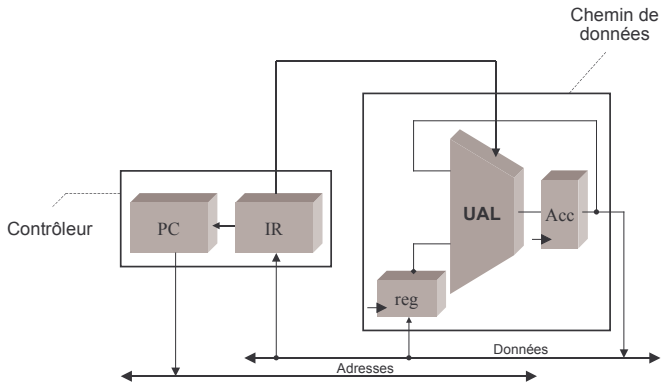


FIG. 13 - *Exemple d'architecture minimale de processeur programmable. Elle est composée d'un contrôleur et d'un chemin de données.*

b. Architecture mémoire

Le contrôleur et le chemin de données constituent les organes de base d'un microprocesseur. Un aspect important de la conception des processeurs est l'interfaçage avec la mémoire. Nous verrons dans ce paragraphe qu'il existe principalement deux approches qui sont l'architecture de Von Neumann et l'architecture de Harvard.

- Architecture de Von Neumann

Par abus de langage, on qualifie souvent les processeurs à usage général (GPP : General Purpose Processors) de processeurs de type Von Neumann. En fait, ce n'est pas tellement la structure interne du processeur qui justifie cela, mais plutôt l'architecture mémoire. En effet, ce modèle suppose une mémoire unifiée dans laquelle données et instructions partagent le même support matériel (figure 14). Même si aujourd'hui, les microprocesseurs à usage général possèdent des caches d'instruction et de données, l'architecture mémoire du système est toujours basée sur ce modèle. C'est ensuite la hiérarchie mémoire qui permet d'accélérer les transferts entre les différents niveaux.

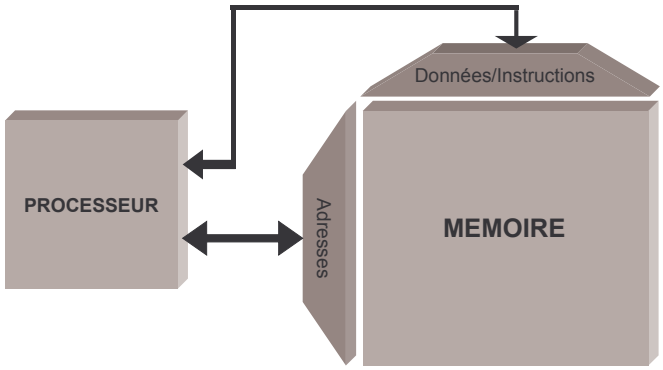


FIG. 14 - *Architecture de Von Neumann appliquée aux microprocesseurs. Le programme et les données manipulées par le programme se trouvent dans la même mémoire.*

- Architecture de Harvard

Il est bien évident que le modèle de Von Neumann est limitatif dans le sens où données et instructions se partagent le même bus de communication. Il en résulte un goulet d'étranglement entre processeur et mémoire. Ce phénomène est accentué par le fait que la mémoire est souvent plus lente que le processeur. Pour remédier à ces inconvénients, une solution consiste à utiliser une architecture mémoire de Harvard (figure 15), dans laquelle la mémoire de données et la mémoire programme sont physiquement disjointes. Ce type d'architecture est couramment employé dans les DSP (Digital Signal Processors). Notons cependant qu'il est souvent nécessaire de faire trois accès mémoire simultanément, ce qui revient à aller chercher une instruction et ses deux opérandes. L'architecture de Harvard étant inapte à l'implantation de ce triple accès, l'idée consiste à utiliser une mémoire cache qui va stocker les instructions qui pourront être réutilisées , déchargeant ainsi les deux bus pouvant alors accéder aux deux opérandes directement dans la mémoire de données. Cette extension, cache ajouté à l'architecture de Harvard, est appelée Super Harvard ARChitecture (SHARC).

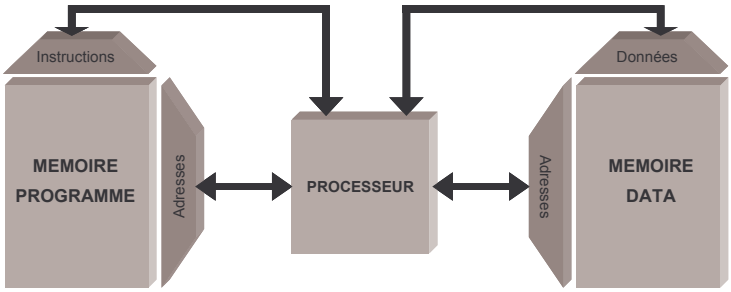


FIG. 15 - *Architecture de Harvard : la séparation physique des deux éléments et l'utilisation de Bus dédiés permet d'accéder simultanément aux données et aux instructions de programmation. Par ce biais, on double théoriquement les performances du processeur.*

- Hiérarchie mémoire

Afin de ne pas limiter les performances théoriques des processeurs, il est nécessaire de définir une hiérarchie mémoire adaptée. Le principe de cette hiérarchie consiste à tirer parti de la localité et des caractéristiques coût/performance des technologies mémoire. Cette localité peut-être spatiale, ce qui correspond à la proximité, dans la mémoire, des données, ou bien temporelle, ce qui correspond au fait qu'une donnée manipulée à un instant a de fortes chances d'être réutilisée dans un futur proche. Ce principe associé au fait que du matériel plus petit est plus rapide, conduisent à une hiérarchie basée sur des mémoires de tailles et de vitesses différentes. La figure 16 représente une hiérarchie mémoire à 3 niveaux.

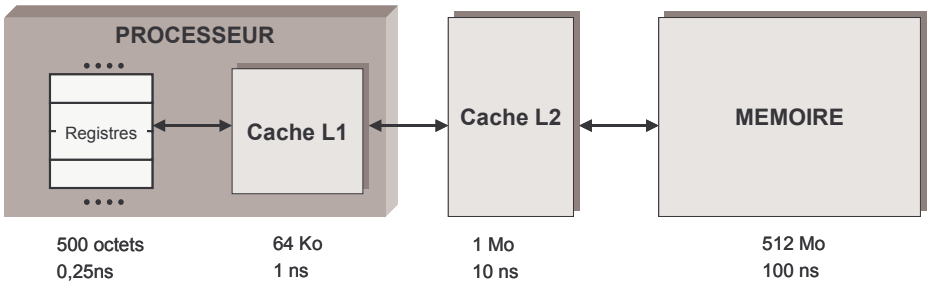


FIG. 16 - *Exemple d'une hiérarchie mémoire à 4 niveaux, avec exemples de tailles et de temps d'accès.*

Ces dernières années, de nombreux travaux de recherche ont été menés pour améliorer les performances des caches. Ces techniques permettent de réduire les taux d'échecs et la pénalité d'échec des caches qui conduisent à des périodes de suspension de traitement du processeur. Pour plus de détails, le lecteur pourra se référer à [Henn03] qui recense les techniques les plus efficaces pour hiérarchiser et dimensionner les caches.

c. Le pipeline dans les processeurs

Une des bases des évolutions architecturales des processeurs est le pipeline. Il s'agit d'une technique pour laquelle plusieurs instructions se recouvrent au cours de leur exécution. Il exploite le parallélisme temporel qui existe entre les différents phases d'exécution des instructions, et ce grâce à leurs recouvrements tel que cela est illustré dans le tableau 3. Aujourd'hui, c'est la première technique utilisée dans les processeurs, RISC notamment, pour améliorer les cadences de traitement : il permet en effet de réduire le temps d'exécution moyen des instructions (réduction du CPI ou Cycles par Instruction [Dub90]).

Afin de clarifier les principes de bases du pipeline, prenons l'exemple d'une architecture pour laquelle le nombre de cycles par instruction est de cinq. L'exécution de chaque instruction est décomposée de la manière suivante :

1. Cycle de lecture de l'instruction (F)
2. Cycle de décodage de l'instruction ou lecture de registre (Dec)
3. Cycle d'exécution ou calcul de l'adresse effective (Ex)
4. Cycle d'accès mémoire (Mem)
5. Cycle d'écriture du résultat (Wr)

Il est possible de pipeliner l'exécution d'une instruction décrite ci-dessus, en démarrant une nouvelle instruction à chaque cycle d'horloge. Le tableau 3 illustre cette exécution *pipelinée*.

Tab. 3. *Exemple de pipeline à cinq étages. A chaque cycle d'horloge, une nouvelle instruction est commencée. Si une instruction est démarrée à chaque cycle, le nombre de cycles de calcul sera jusqu'à cinq fois inférieur à celui d'un processeur non pipeliné.*

	Numéro de Cycle								
n° d'instruction	1	2	3	4	5	6	7	8	9
Instruction i	F	Dec	Ex	Mem	Wr				
Instruction i+1		F	Dec	Ex	Mem	Wr			
Instruction i+2			F	Dec	Ex	Mem	Wr		
Instruction i+3				F	Dec	Ex	Mem	Wr	
Instruction i+4					F	Dec	Ex	Mem	Wr

La figure 17 représente un processeur mettant en œuvre le pipeline. Lorsque deux instructions sont présentent parallèlement dans le pipeline et qu'elles n'ont aucune dépendance, elles peuvent s'exécuter simultanément sans provoquer de suspension. Par contre, si deux instructions sont dépendantes l'une de l'autre, elles doivent être exécutées dans l'ordre et ne peuvent être que partiellement recouvertes. Le chargement d'une instruction conditionnelle, dont la variable testée est en cours de modification par l'UAL, présente par exemple un risque en terme de suspension du pipeline.

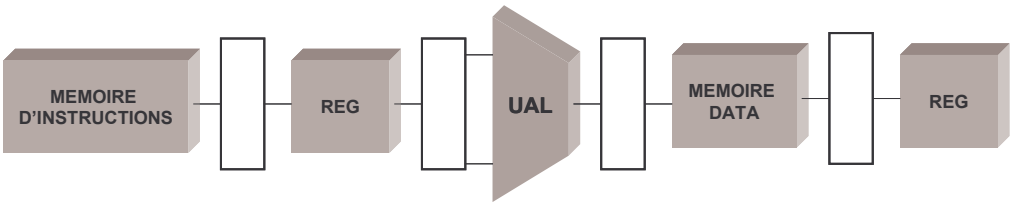


FIG. 17 - *Illustration de l'architecture d'un processeur permettant le pipeline d'instructions. Notez que le modèle présenté ici suppose une lecture simultanée des instructions et des données (architecture mémoire Harvard).*

1.1.2. Classifications

Dans ce paragraphe, nous présentons deux exemples de classifications des processeurs. Pour la première, il s'agit d'une classification suivant le jeu d'instruction et pour la deuxième la distinction se base sur le flot de donnée(s) et d'instruction(s).

a. Classification du jeu d'instruction

Le type de stockage interne des opérandes dans un processeur peut permettre de différencier les jeux d'instruction de processeurs programmables. C'est ce que proposent les auteurs de [Henn03] en nous présentant dans leur ouvrage quatre type d'architecture de jeu d'instruction. Les choix principaux d'opérandes sont les piles, les registres ou les accumulateurs. Ces opérandes peuvent être adressés implicitement ou explicitement. La figure 18 illustre le flot de positionnement pour les quatre architectures.

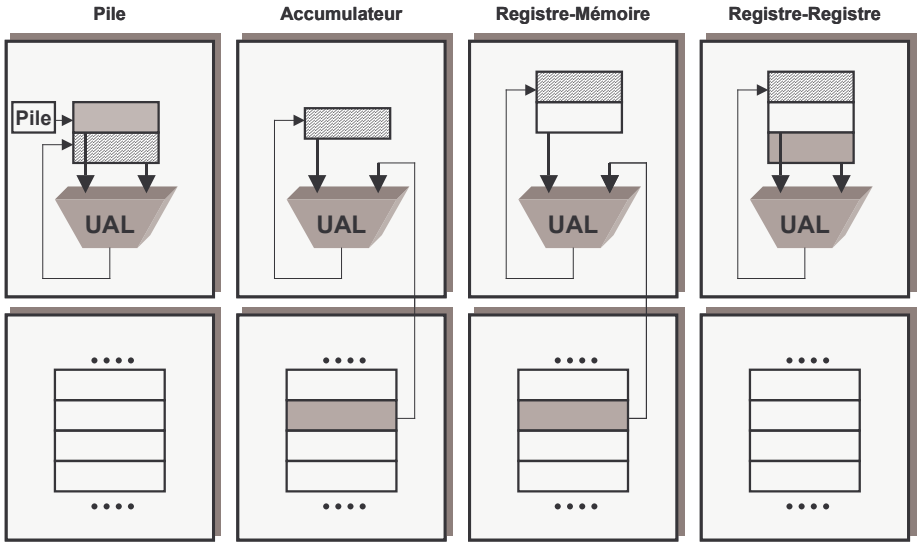


FIG. 18 - *Position des opérandes pour quatre types d'architecture de jeux d'instructions*

Historiquement, la plupart des anciens ordinateurs fonctionnaient à pile ou à accumulateur. Mais depuis 1980, les architectures conçues sont quasiment toutes basées sur

une architecture Load/Store ou Registre/Registre. Cette évolution s'est faite parallèlement à la mise en œuvre des outils de compilations.

A titre d'exemple, nous pouvons citer les processeurs DEC Alpha 21164[Gwen94-a], le PowerPC 601 [Moor93] ou le MIPS[Gwen94-b] pour les jeux d'instruction Registre/Registre, et les processeurs Intel 80x86 [Henn03] ou TI TMS320C54x [Texa00a] pour les jeux d'instruction Registre/Mémoire. Les premiers possèdent l'avantage de permettre un codage simple et de longueur fixe des instructions. La génération de code est elle aussi simplifiée et l'exécution des instructions possède un nombre de cycle fixe. En revanche, le nombre d'instructions est plus élevé et implique des programmes de taille conséquente. A l'opposé, les architectures Registre/Mémoire possèdent une meilleure densité de code d'où une taille des programmes réduite.

b. Classification de Flynn

Pour augmenter les performances des systèmes à base de processeur, plusieurs approches sont envisageables. Tout d'abord, on peut cibler les performances intrinsèques du composant, c'est, par exemple, ce que l'on cherche à produire avec le pipeline. Une autre approche consiste à démultiplier les unités, c'est le cas des multiprocesseurs. L'idée d'utiliser plusieurs processeurs pour améliorer les performances est apparue avant même la fabrication du tout premier microprocesseur. A cet effet, Flynn [Hwa84] propose en 1966 un modèle simple de classification des ordinateurs, qui reste encore utilisée aujourd'hui pour caractériser les architectures de processeurs parallèles. Son analyse se base sur le parallélisme dans les flots d'instructions et de données ce qui a donné naissance à quatre catégories qui sont répertoriées dans le tableau 4. Ce tableau résume les quatre possibilités d'architectures de processeurs parallèles. La première, SISD (Single Instruction Single Data), correspond à une seule instruction sur une seule donnée ce qui est équivalent à une architecture mono-processeur de type « Von Neumann ». La deuxième, SIMD (Single Instruction Multiple Data), correspond à une seule instruction appliquée à de multiples données : on parle aussi de traitement vectoriel [John78] des données. Les processeurs SIMD permettent au composant d'effectuer des traitements identiques sur des données de taille réduite (par exemple, sur un chemin de données de largeur 32 bits, au lieu d'effectuer 1 seule opération sur une donnée de 8 bits, on scinde l'opérateur en 4 ce qui permet d'effectuer le même traitement sur 4 données en même temps, permettant de réduire d'autant le nombre de cycles de calcul). La classe des architectures MISD (Multiple Instruction Single Data) peut-être apparenté avec précaution aux architectures *pipelinées*. Quant aux architectures MIMD (Multiple Instruction Multiple Data), il s'agit de processeurs parallèles exécutant des instructions différentes sur des données multiples. Ces processeurs présentent des propriétés très intéressantes pour l'exploitation de processus disjoints, comme le parallélisme au niveau tâche.

Tab. 4. *Classification de Flynn*

		Flot de données	
		Unique	*Multiple*
Flot d'instructions	*Unique*	SISD	SIMD
	Multiple	MISD	MIMD

1.1.3. Evolution des modèles

Il sera question dans ce paragraphe de décrire les principales évolutions marquantes de la conception des processeurs, en rappelant tout d'abord l'évolution CISC vers RISC[Shiv00], les deux méthodes les plus connues pour tirer parti du parallélisme d'instruction et enfin, le multi-flots[Tul95].

a. CISC et RISC

Le premier est l'acronyme de *Complex Instruction Set Computer* et le second de *Reduced Instruction Set Computer*. Ces deux architectures diffèrent principalement dans le codage des instructions.

Historiquement, les CISC ont concentré la plupart des efforts de développement durant de nombreuses années. Les 8086-8088 sont deux exemples de processeurs CISC. Les concepteurs rajoutaient à l'époque le plus d'instructions possible pour permettre à l'utilisateur d'améliorer ses programmes. Néanmoins, cela avait tendance à ralentir le fonctionnement global et certaines instructions complexes étaient finalement peu utilisées en pratique. L'avantage de ce type d'architecture résidait dans la taille réduite du code généré : une instruction « complexe » permettait une fois décodée de réaliser une succession de multiples micro-instructions. Une fois compilé, le programme était de taille relativement faible, ce qui permettait d'économiser de la mémoire. Par contre, le temps pris pour décoder les instructions complexes étant variables, cette technique limitait la mise en œuvre du parallélisme d'instruction comme le pipeline. De plus, le décodage de ces instructions nécessitait un support matériel adéquat non négligeable par rapport à la surface du processeur.

A l'opposé, les processeurs RISC disposent d'un jeu d'instruction réduit. Peu de primitives sont utilisées et c'est leur combinaison qui permet de réaliser des algorithmes complexes. De plus, le nombre de cycles par instruction est fixe et peu élevé. Cette caractéristique permet notamment de réaliser des pipelines d'instructions plus facilement. Ces processeurs, du fait de leur structure plus simple, permettent de stocker plus de données en interne à surface identique. Grâce à ces avantages et malgré des tailles de programmes plus élevées, les processeurs RISC semblent aujourd'hui s'être imposés devant les CISC.

b. Exploitation du parallélisme d'instruction

Pour réduire le temps d'exécution d'un programme, il est possible d'exploiter le parallélisme d'instruction. Comme nous l'avons vu précédemment, le pipeline d'instruction permet de réduire le nombre de cycles par instruction et donc, le temps d'exécution total. Cette technique a conduit au développement d'architectures dites *superpipeline* (20 étages dans le PIV).

Cependant, le pipeline n'est pas si simple à mettre en œuvre. L'une des difficultés majeures provient de la gestion des aléas. Un aléa se traduit par une rupture de pipeline, quand, par exemple, une instruction nécessite le résultat d'une autre instruction toujours présente dans le pipeline. Dans ce cas, l'instruction incidente doit être stoppée afin d'attendre que le résultat soit disponible, d'où une diminution de la cadence de traitement.

Les concepteurs de processeurs ont cependant mis au point des techniques qui permettent de remédier à cette diminution des performances dans les architectures pipeline. Ces techniques reposent sur une méthode d'exploitation du parallélisme spatial entre opérations. L'idée consiste à ajouter des unités fonctionnelles au processeur et à exécuter simultanément, par ce biais, plusieurs instructions. Les architectures superscalaires [John90]

et les architectures VLIW[1] [Fish84][Colw88] sont deux types d'architectures exploitant ce type de parallélisme. Pour les premières, le code machine initial n'est pas modifié. Il s'agit d'un programme compilé pour une machine purement séquentielle, et c'est un contrôleur complexe qui répartit les instructions sur les différentes unités fonctionnelles. L'utilisation du code exécutable originel permet d'assurer une compatibilité du logiciel entre les différentes générations de systèmes. Pour les architectures VLIW, cette compatibilité logicielle n'est pas assurée. Ceci n'est pas très gênant dans la mesure où ce type d'architecture est plutôt dédié à des applications spécifiques comme le traitement du signal et des images. Le programme est donc compilé de manière spécifique, d'où un contrôleur beaucoup plus simple puisque le parallélisme est extrait en amont. Le compilateur est chargé, en effet, de définir un ordonnancement des opérations permettant d'exploiter les unités parallèles mises à disposition par le matériel.

c. Le multi-flots

L'exploitation du parallélisme peut-être vu à un niveau plus élevé que l'ILP. Le parallélisme de flot correspond à cette approche où le flot peut être d'une granularité plus ou moins grosse : il peut s'agir de processus indépendants correspondant à l'exécution de plusieurs programmes chargés en mémoire ou encore de sous-processus ou tâches partageant les mêmes segments de code et de données. Typiquement, ce niveau de parallélisme, pour les processus, est intégré dans les systèmes d'exploitation modernes où chaque application dispose d'une tranche temporelle pour utiliser le processeur central. Pour exploiter des sous-processus, sont apparus depuis quelques années des mécanismes matériels intégrés aux processeurs permettant de prendre en charge directement l'exécution de multiples flots sur la même ressource[Mad00]. Nous en présenterons quelques aspects dans la section 3.2.

1.2. Architecture des processeurs superscalaires et SMT

Ces processeurs sont souvent perçus comme une extension des processeurs RISC. En effet, au traditionnel pipeline de traitement, le traitement superscalaire ajoute la capacité de lancer simultanément, durant la même période d'horloge, plusieurs instructions (figure 19). Cette propriété permet d'exploiter plus efficacement le parallélisme d'instruction en diminuant le nombre moyen de cycles par instruction. Pour cela, les unités de calcul (entières et flottantes) sont démultipliées et des structures matérielles récupèrent un flux d'instructions, permettant la détection des aléas et un ordonnancement statique ou dynamique des instructions pouvant s'exécuter en parallèle. Ces techniques permettent à la fois de limiter les dommages dus aux ruptures de pipeline et de profiter plus efficacement du parallélisme au niveau instruction.

Dans cette section, nous allons présenter ce type d'architecture de processeur, en rappelant tout d'abord les principes du modèle de programmation, puis nous présenterons une architecture générique de processeur superscalaire ainsi que quelques exemples de réalisation.

1.2.1. Modèle de programmation

a. Compatibilité binaire

Une des contraintes majeures des concepteurs de processeurs est la compatibilité du logiciel : on l'appelle la compatibilité binaire, i.e. la capacité d'une machine à exécuter un programme écrit pour une génération antérieure du processeur. C'est ce qui explique en partie pourquoi, aujourd'hui encore, la sémantique d'un processeur superscalaire est entièrement séquentielle. Pour ces processeurs, le modèle de programmation est similaire à ceux des

[1] VLIW : Very Long Instruction Word

premiers processeurs. A l'origine, en effet, l'idée était de découper l'algorithme en une succession d'instructions exécutées les unes après les autres. Une application est généralement décrite dans un langage de haut niveau. Elle est ensuite compilée dans le langage de la machine cible, correspondant au codage binaire de l'application. Le programme définit explicitement l'ordre d'exécution des instructions ce qui connote son caractère statique. Pour outrepasser ces contraintes de compatibilité binaire, les concepteurs ont dû rivaliser d'astuces, en partant de l'analyse du langage machine et de l'ordonnancement des instructions.

b. Dépendances de contrôle et de donnée

Dans le programme, on peut distinguer différents blocs : ceux pour lesquels il y a juste une incrémentation du compteur du programme, et ceux pour lesquels l'adresse est dépendante d'un « contrôle » (branchements conditionnels ou *jump*). En général, on dit que l'ensemble des instructions exécutées consécutivement définissent un flux d'instructions dynamiques ou encore bloc.

Une méthode pour augmenter le parallélisme d'instructions consiste à contourner les dépendances de contrôle. En effet, si l'on considère que dans un bloc de base, il y a un ensemble d'instructions contiguës, avec un seul point d'entrée et de sortie, on peut supposer qu'une fois entré dans un bloc, l'ensemble de ces instructions sera forcément exécuté. Par conséquent, chaque séquence d'instructions d'un bloc de base peut-être lancée en masse, dans ce qu'on appelle la fenêtre d'exécution. Cette fenêtre contient alors l'ensemble des instructions qui peuvent être considérées pour une exécution en parallèle. Ensuite, ces instructions ne sont plus sujettes qu'aux éventuelles dépendances de données qui peuvent éventuellement limiter le parallélisme d'instruction dans la fenêtre d'exécution. Pour obtenir plus de parallélisme, une technique complémentaire vise à prédire le résultat des instructions de branchement et de réaliser ce qu'on appelle une exécution spéculative d'un bloc. Si après coup, le branchement se révèle juste, l'exécution du processus suit son cours normal, sinon le chemin non considéré est exécuté.

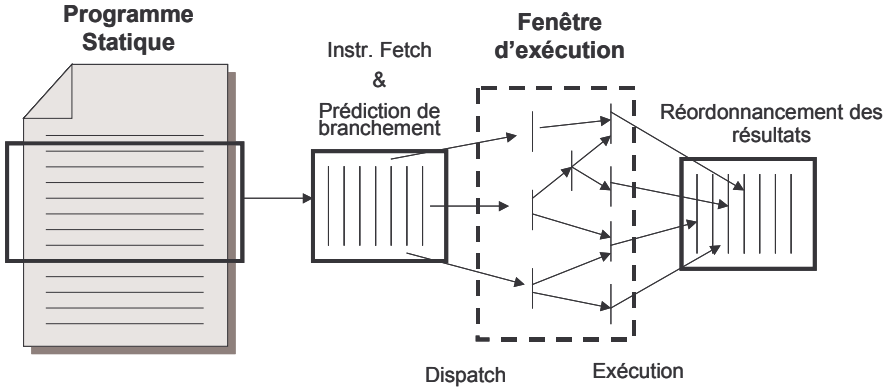


FIG. 19 - *Concept d'exécution superscalaire d'un programme séquentiel [Smit95]. Les instructions sont « bufferisées » et une prédiction de branchement permet de dispatcher, dans la fenêtre d'exécution, les différentes instructions pouvant éventuellement s'exécuter en parallèle. En fonction du parallélisme possible, des opérateurs disponibles et des dépendances de données, l'exécution est lancée éventuellement dans le désordre. En conséquence, un réordonnancement doit être effectué afin de revenir à l'ordre initial des instructions.*

Les instructions d'un bloc chargé dans la fenêtre d'exécution peuvent cependant être soumises aux dépendances de données. Ces dépendances s'observent lorsque par exemple deux instructions tentent d'accéder à un même registre : on parle dans ce cas d'aléa qui peut

perturber le bon déroulement du processus (de la même manière que dans les architectures pipeline simple). Pour passer outre ces aléas, il est donc nécessaire que le matériel puisse définir un ordonnancement pour lequel l'exécution parallèle des instructions ne perturbe pas le processus. Cet ordonnancement matériel suppose que l'ordonnancement défini statiquement par le compilateur soit modifié dynamiquement par le matériel pour une exécution dans le désordre.

1.2.2. Architecture superscalaire générique

La figure 20 illustre l'architecture générique d'un processeur superscalaire. Les éléments majeurs de ce processeur sont les unités de *« instruction fetch »*, *« prédiction de branchement »*, « *décodage et analyse des dépendances »*, *« ordonnancement et exécution »*, et *« réordonnancement des résultats»*. Ces phases constituent plus ou moins le flot d'exécution. A noter qu'il s'agit là aussi d'une architecture pipelinée.

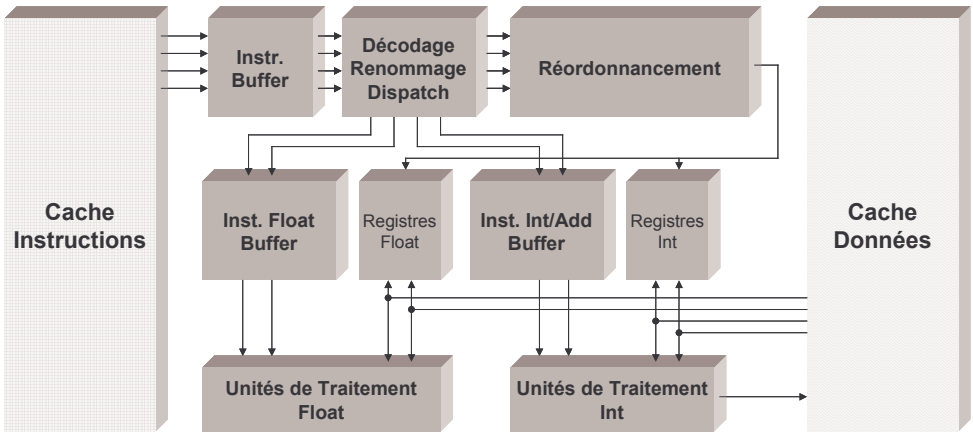


FIG. 20 - *Architecture d'un processeur superscalaire (modélisation simplifiée du MIPS R10000[Gwen94]). Sur cet exemple, quatre instructions peuvent être exécutées en parallèle, soit 2 instructions entières/adresses et 2 instructions en virgule flottante.*

a. Instruction *Fetch* et prédiction de branchement

La phase d'instruction *fetch* alimente tout le pipeline de traitement en instructions. On utilise un cache d'instructions dans lequel tout ou partie d'un processus est chargé pour accélérer l'accès aux instructions. Il se peut que parfois, suite à des branchements, l'instruction suivante ne se trouve pas dans le cache : on parle alors d'échec de cache, dans le cas contraire c'est un succès de cache.

Dans le cas d'une exécution superscalaire, la phase de *fetch* doit être capable de récupérer plusieurs instructions du cache en un seul cycle. Les bandes passantes requises pour ces chargements multiples ont motivé la séparation des caches de données et d'instructions. Le nombre d'instruction chargé à chaque cycle est au moins égal au nombre maximal d'instructions que l'unité est capable de décoder.

Les méthodes de chargements multiples s'appuient soit sur une incrémentation simple du compteur du programme pour des sections de code contiguës, soit par un système de prédiction de branchement pour les sauts conditionnels. Le traitement des instructions de branchement est généralement composé des étapes suivantes :

1. Détection d'une instruction de branchement conditionnel
2. Détermination de l'issue du branchement : on utilise des méthodes de prédiction de branchement. Celles-ci peuvent être statiques (issues d'information du compilateur) ou dynamiques, *i.e.* réalisées lors de l'exécution, grâce notamment à l'utilisation d'un historique des branchements.
3. Exécution du branchement : ce qui va permettre de calculer l'adresse de la portion suivante de code à lire
4. Rediriger le fetch : le compteur du programme est positionné sur l'adresse du bloc d'instructions à suivre si le branchement a été choisi.

b. Décodage des instructions, renommage, et dispatch

Durant cette phase, les instructions sont récupérées à partir du *buffer* d'instructions, examinées, et les dépendances de contrôle et de données sont analysées. Ce bloc est également chargé de distribuer ou dispatcher les instructions à exécuter dans les *buffers* des unités de traitement entières et flottantes. Afin d'éviter les aléas et d'augmenter le parallélisme durant l'exécution dynamique, les éléments physiques de stockage tels que les registres sont renommés si les dépendances le permettent. A l'issue de cette étape, une micro-instruction d'exécution est générée (*i.e.* le code opération et la localisation des registres des opérandes).

c. Choix des instructions et exécution parallèle

Une fois que la micro-instruction d'exécution a été créée, elle est *bufferisée* dans l'unité vers laquelle elle est destinée, suivant son code opération. A ce niveau, les instructions vont être choisies pour l'exécution suivant des critères tels que la disponibilité des ressources physiques de calcul et la disponibilité des opérandes.

d. Phase de recouvrement

Cette phase que l'on peut aussi nommer phase de réordonnancement, permet d'attendre suivant les contraintes d'exécution le moment opportun pour modifier l'état du processus. L'objectif de cette phase est donc de simuler un modèle d'exécution séquentiel bien que la phase de traitement ait été essentiellement parallèle, grâce notamment à l'exécution spéculative de portions de codes suite à des prédictions de branchement ou encore l'exécution dans le désordre des opérations.

1.2.3. Implémentation matérielle du multi-flots

Les processeurs multi-flots s'appuient sur une architecture superscalaire. L'idée consiste à greffer sur l'architecture plusieurs contextes physiques ce qui permet d'exécuter en parallèles des instructions provenant de différentes tâches[1], contrairement aux processeurs superscalaires classiques qui ne traitent simultanément que des instructions d'un même processus. L'avantage de cette approche réside dans sa capacité à mieux remplir les unités fonctionnelles et, surtout, dans son aptitude à gérer plus efficacement la bande passante entre la mémoire, l'antémémoire et le processeur. Par exemple, un processus qui doit aller chercher des instructions en mémoire centrale lors d'un échec de cache, peut exécuter des instructions d'un autre processus présent dans l'antémémoire.

[1] ces tâches peuvent correspondre à des processus disjoints (multi-flots hétérogène), c'est-à-dire issus d'applications différents, ou des sous-processus d'une même application (multi-flots homogène)

a. Notion de contextes

Afin de passer rapidement d'un flot à un autre, le matériel doit fournir les ressources adéquates. Un contexte d'exécution est constitué de l'ensemble des registres utilisés par la tâche pendant son exécution (entiers, flottants, compteur du programme, registres d'état, configuration du MMU[1]). Le contexte physique correspond lui à l'ensemble des registres nécessaires pour stocker dans le processeur un contexte d'exécution.

Dans un processeur superscalaire classique, il n'y a qu'un seul contexte physique, mais il peut y avoir plusieurs contextes d'exécution, un par processus. Dans un tel processeur, à un instant donné, un seul processus s'exécute alors que pour un processeur superscalaire multi-flots, plusieurs processus peuvent être exécutés en même temps.

Le changement de contexte peut s'effectuer suivant deux approches différentes du mutli-flots. Le multi-flots à grain fin commute entre les flots à chaque instruction, ce qui permet d'exécuter de manière entrelacée plusieurs flots, en sautant évidemment ceux qui sont suspendus. Ce type de multi-flots permet de masquer les pertes de débit mais retarde l'exécution individuelle de chaque flot. Comme alternative, le multi-flots à grain épais permet des changements de flot uniquement sur des suspensions coûteuses comme des défauts de cache secondaire.

b. Multi-Flots simultané

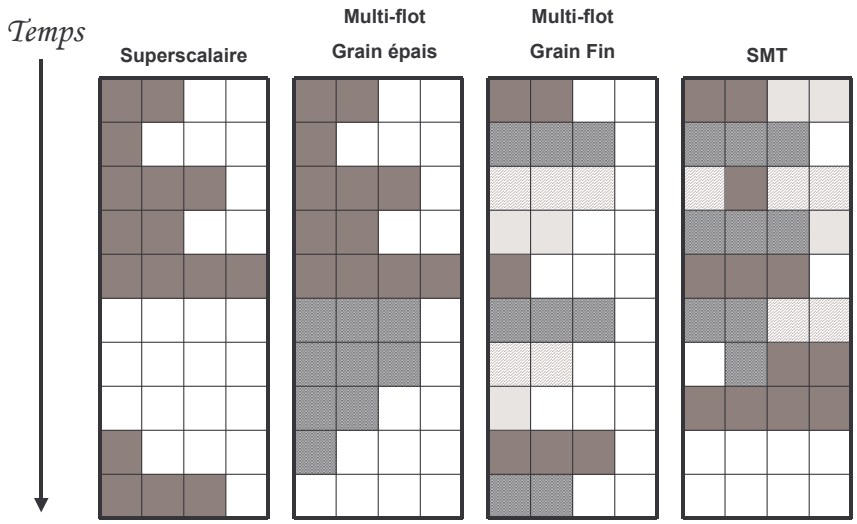


FIG. 21 - *Quatre exemples de traitements : superscalaire, Multi-Flots grain épais et grain fin, SMT. Pour l'approche superscalaire, on observe que le processeur est suspendu pendant 3 cycles avant de continuer le processus commencé. Dans le multi-flots à grain épais, une deuxième tâche est exécutée lors de la suspension du premier processus. Pour le multi-flots à grain fin, 4 tâches sont entrelacées, ce qui globalement allonge le temps d'exécution de chaque tâche, mais augmente en contrepartie le débit. Pour la dernière approche, on constate non seulement un meilleur débit mais en plus une minimisation du nombre de cycles par rapport à l'approche grain fin.*

L'objectif du multi-flots simultané ou SMT est de convertir le parallélisme de flot en parallélisme d'instructions. Il utilise les ressources d'un processeur superscalaire ordonnancé dynamiquement à lancement multiple. La motivation sous-jacente est qu'un processeur superscalaire dispose la plupart du temps de plus d'unités fonctionnelles qu'un seul flot n'a besoin. De plus, grâce au *renommage* de registres et l'ordonnancement dynamique, plusieurs

[1] Memory Management Unit: cette unité permet l'accès paginé à la mémoire

instructions de flots indépendants peuvent être lancées simultanément sans se préoccuper des problèmes de dépendance.

La figure 21 illustre le fonctionnement du multi-flots[Henn03] par quatre exemples basés sur un même support matériel (4 unités de traitement), et pour chacun des aptitudes au multi-flots variables, allant du superscalaire simple au SMT.

1.3. Architecture des processeurs DSP et VLIW

Pour certaines classes d'applications, le matériel peut nécessiter certaines ressources particulières dont les processeurs d'usage général ne disposent pas ou encore, ne pas avoir besoin de certaines coûteuses en performances. C'est dans cet esprit que les premiers DSP ont été développés. Il s'agit en fait de microprocesseurs programmables, basés sur le même modèle d'exécution que leur cousin, mais dédiés à une classe d'applications particulières : le traitement du signal et des images. Ces composants ont connu une très forte croissance ces dernières années, car on les retrouve notamment dans les téléphones portables.

Dans cette partie, nous allons tout d'abord revenir sur les principes de base des architectures DSP puis nous présenterons plus dans le détail une architecture VLIW et ses modes de programmation.

1.3.1. Architecture des DSP

a. Modèle générique

L'architecture générique d'un DSP est présentée sur la figure 22. Elle est basée en tout premier lieu sur l'utilisation d'une architecture mémoire de Harvard, qui permet un accès simultané aux données et aux instructions. Ensuite, afin de permettre la prise en charge des applications TSI, un multiplieur est disponible, et il est possible d'effectuer des opérations MAC (Multiplication-Accumulation) en un seul cycle (ces opérations sont très courantes en TSI, par exemple pour l'implantation de filtres). Un DSP peut également intégrer des périphériques (convertisseurs, contrôleur mémoire…) si bien que le la frontière entre microcontrôleur/DSP est parfois floue.

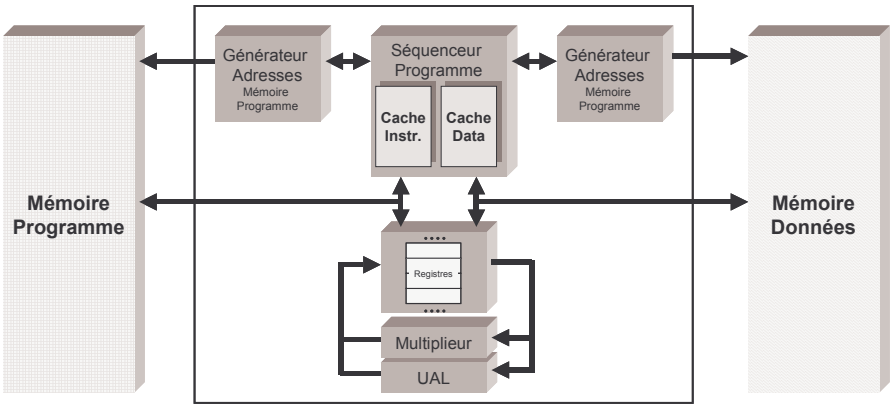


FIG. 22 - *Architecture générique d'un DSP. Le modèle de Harvard est la principale caractéristique de cette architecture, ainsi que la présence d'un bloc UAL/Multiplieur pour la prise en charge des opérations MAC.*

Des fabricants tels que TI (Texas Instrument) [Texa04], Analog Devices [Anal04], Motorola [Moto04], ST [Stmi04] proposent leurs propres structures de DSP. Par exemple, parmi les différents types de DSP du leader TI, on peut distinguer les DSP à virgule fixe,

virgule flottante, orientés contrôle (TI, famille C2x), orientés basse puissance (TI C54, C55) ou encore ceux orientés hautes performances (TI, famille C6x).

Aujourd'hui, les architectures de ces processeurs intègrent des mécanismes architecturaux similaires aux processeurs généraux. En effet, les dernières générations de DSP intègrent des pipelines profonds (jusqu'à une dizaine d'étages), des jeux d'instructions particuliers (VLIW et/ou SIMD), des mémoires caches (d'instructions et/ou de données) ainsi que des périphériques pour faciliter le transfert de données (DMA). De plus, les fréquences de fonctionnement suivent la même pente croissante, mais moins élevée, que celles des processeurs généraux (quelques centaines de méga Hertz).

b. Modes d'adressage spécifiques

Quelque soit l'architecture de processeur, il est nécessaire d'accéder aux données d'application en spécifiant leur adresse mémoire. Il peut s'agir d'un adressage immédiat ou bien d'un adressage résultant d'un calcul (souvent auto-incrément ou auto-décrément). Pour les DSP, cela est d'autant plus important qu'ils traitent en général des applications pour lesquelles le nombre de données est très important. Afin de pouvoir alimenter de manière continue les unités de calcul, l'adressage de la mémoire doit être automatisé par l'utilisation d'instructions et de matériel d'adressage spécifiques.

Les calculs s'appuient régulièrement sur des tampons circulaires : tout DSP récent a un mode d'adressage circulaire ou modulo pour gérer ces tampons automatiquement. Pour des applications telles que la FFT, les DSP possèdent un mode d'adressage appelé « bits inversés », qui permet de réaliser les inversions de bits sur les adresses lors des étapes de calcul de la FFT

c. Types d'opérandes

En TSI, les données à manipuler peuvent aussi bien être des complexes, des réels ou des entiers. Le type de codage utilisé peut donc être aussi bien entier que flottant.

En traitement d'image par exemple, les pixels permettent de traiter les éléments visibles d'une image et sont codés, en couleur, sur 32 bits (3 composantes R, G, B et A qui indique la transparence de la surface) ou en 256 niveaux de gris, sur 8 bits. Lorsqu'on travaille sur des images en couleur, les traitements étant identiques sur chaque composante, les calculs SIMD permettent à un chemin de largeur 32 bits de traiter simultanément les 4 composantes du pixel (par exemple, extensions MMX d'INTEL [Inte01]), ce qui permet d'accroître l'exploitation du parallélisme.

Actuellement, il existe des DSP qui utilisent le codage en virgule fixe, et d'autres en virgule flottante. En virgule fixe, les données sont le plus souvent codées sur 16 ou 24 bits. C'est le programmeur qui choisit la manière dont il va utiliser ce codage, par exemple, pour le placement de la virgule. Pour cela, des opérations supplémentaires de recadrage de données sont à prendre en compte ce qui complexifie le programme initial. Le codage en virgule flottante offre en général une meilleure précision et une plus grande dynamique de représentation des nombres réels. Cependant, les DSP en virgule fixe étant plus rapides, plus petits et moins consommateurs d'énergie, 95% des DSP actuels sont en virgule fixe 16 bits. Afin d'assurer une meilleure précision des calculs, le codage en interne est souvent plus élevé (24 ou 32 bits).

1.3.2. Architecture VLIW

Après avoir abordé les concepts architecturaux de base des processeurs DSP, nous allons maintenant nous focaliser sur un aspect fondamental du traitement : l'exploitation du parallélisme d'instruction.

a. Ordonnancement statique et VLIW

Les DSP, comme les processeurs classiques, sont des architectures programmables basées sur le même principe d'exécution. La multiplication des unités de traitement suppose cependant une exploitation du parallélisme au niveau instruction, comme pour les processeurs superscalaires. En revanche, la spécificité du domaine d'application impose un autre mode de pensée. En effet, la régularité des applications ciblées suppose finalement peu de contrôle et beaucoup plus de déterminisme que pour d'autres types d'application : cette hypothèse fonde la base même des architectures VLIW. Le déterminisme implique que la révélation du parallélisme peut-être effectuée par avance, grâce à des compilateurs adaptés qui se chargent de déployer le parallélisme sur les différentes ressources de calcul mises à disposition par les DSP : le processus d'ordonnancement est donc ici statique. L'outil de compilation estime les probabilités d'exécution des branches de calcul résultant d'instructions conditionnelles et identifie les instructions éventuellement soumises aux aléas. A partir de cette spéculation, il génère un ordonnancement qui permet d'exploiter au mieux l'ensemble des ressources. Il en résulte une sémantique parallèle de langage machine, contrairement aux processeurs superscalaires qui, du point de vue du compilateur, sont vus comme complètement séquentiels, grâce aux éléments matériels qui se chargent de déployer le parallélisme. La révélation de ce parallélisme nécessite par conséquent la construction d'instructions longues, constituées de la concaténation des instructions destinées à chacune des unités de traitement parallèles.

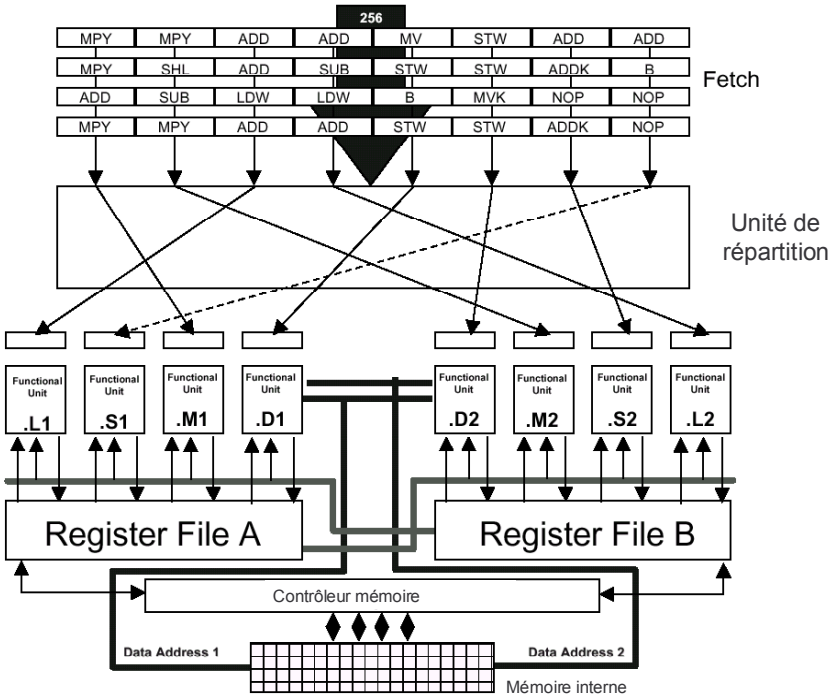


FIG. 23 - *Architecture du cœur du C62 de Texas Instrument.*

C'est suivant ce principe qu'a été développée l'architecture du C62, présentée sur la figure 23. Il possède 8 unités de traitement fonctionnant en parallèle, commandées par une instruction de 256 bits de largeur. Deux banques de 16 registres sont présentes, chacune

communiquant de manière quasi-exclusive avec 4 unités de traitement. Sa puissance crête est de 2400 MIPS à 300 MHz.

b. Perspectives de performances

L'avantage des DSP VLIW par rapport aux processeurs superscalaires réside tout d'abord dans la simplification du matériel nécessaire pour exploiter le parallélisme d'instruction. L'architecture est certes *pipelinée*, mais c'est le compilateur qui résout de manière statique les problèmes d'aléas. Ce statisme confère aux DSP un certain potentiel de faible consommation par rapport aux processeurs superscalaires. En revanche, il en résulte le plus souvent des performances en deçà de celles escomptées. Les outils de génération automatique de code, les compilateurs pour DSP VLIW qui se chargent de révéler le parallélisme et de traduire une description algorithmique de haut niveau en langage machine, ne permettent pas d'atteindre des performances aussi bonnes, que lorsque la programmation est réalisée en assembleur [Demi01].

Il en résulte donc que pour atteindre des performances satisfaisantes, l'effort de conception sera plus grand, car il nécessitera le passage par une représentation moins lisible du programme, le langage de niveau assembleur. L'utilisation d'assembleur linéaire permet cependant un compromis effort de développement / performances. De plus, même si les codes générés par le compilateur sont souvent moins efficaces qu'une description niveau assembleur, la présence d'un environnement de développement complet permettant d'effectuer un « profil » de l'application, facilite grandement le portage d'applications. Il est en effet possible à partir d'une première compilation d'identifier les parties critiques dans le code objet, et ensuite de les optimiser manuellement jusqu'à obtention d'un résultat satisfaisant.

Un second avantage de l'approche VLIW résulte directement de l'hypothèse de base. L'approche déterministe et l'exécution pré-déterminée permettent de mieux garantir des contraintes de date limite de traitement, comme par exemple dans le cadre d'applications en temps-réel [Laca98]. Par contre, le déterminisme tel qu'il est utilisé dans l'approche VLIW ne permet pas d'entrevoir des possibilités autres que l'exploitation de parallélisme au niveau instruction. En effet, le multi-flots résulte du lancement a priori imprévisible de plusieurs programmes. Il semble difficile d'envisager dans ces conditions un entrelacement des instructions appartenant à différents contextes d'exécution, au sein d'un processeur de type VLIW.

2. ARCHITECTURES RECONFIGURABLES

Dans cette partie, nous allons passer en revue les principes généraux des architectures reconfigurables en définissant précisément la terminologie qui sera utilisée dans ce mémoire. Nous verrons ensuite les deux approches principales du domaine des architectures reconfigurables, i.e. celles basées sur des opérateurs à grain fin, et les secondes, basées sur des opérateurs à grain épais.

2.1. Principes généraux

Le terme « configuration » signifie *forme*, *aspect*, et « reconfiguration » peut être interprété dans notre domaine du traitement par *changement de la disposition relative des éléments de calcul*. Autrement dit, une architecture reconfigurable va disposer d'un certain nombre d'éléments interconnectés structurellement d'une certaine manière, la configuration permettant de modifier la disposition relative de ces éléments. C'est ainsi que la fonctionnalité de ce type de processeur est adaptée au traitement à réaliser.

Le schéma de principe permettant la mise en œuvre d'architectures reconfigurables est basé sur une structure bi-couche (figure 24) :

- Une couche de configuration, qui est une mémoire contenant la configuration de l'ensemble de l'architecture reconfigurable.
- Une couche opérative qui contient les éléments fonctionnels de l'architecture, i.e. éléments de calcul, le système d'interconnexion et les blocs d'entrées / sorties.

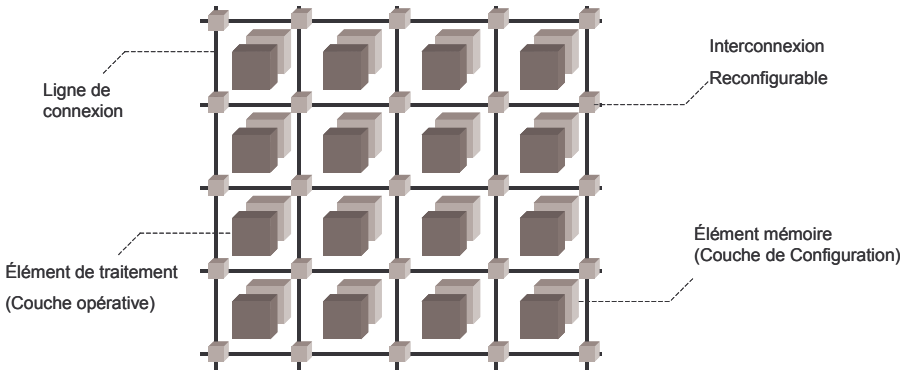


FIG. 24 - *Architecture bi-couche des FPGA*

Plusieurs technologies de stockage de la configuration sont possibles [Brow96][Xili00]. Nous pouvons citer les technologies EPLD (*Erasable Programmable Logic Device*) réalisés à partir de transistors à grille flottante, les technologies *antifusibles*, programmables une seule fois par claquage, et les technologies SRAM basées sur des mémoires statiques, autorisant un nombre de reconfiguration illimité. On se limitera dans la suite aux FPGA-SRAM qui ont pour avantage d'utiliser un procédé CMOS standard. Ces architectures peuvent être classées en différentes catégories suivant, notamment, la granularité du chemin de données, le système d'interconnexion ou encore la méthode de reconfiguration. C'est ce que nous allons analyser dans les prochains paragraphes.

2.1.1. Granularité du chemin de données

L'étude de la granularité du chemin de données est primordiale car elle permet de comprendre les performances potentielles et la flexibilité d'un circuit reconfigurable. Par granularité, on entend « taille en bits de la donnée élémentaire manipulée ». Dans les deux

prochains paragraphes, nous ferons un état de l'art des architectures reconfigurables en nous basant sur ce critère de comparaison, à savoir architectures à grain fin et architectures à grain épais.

Les FPGA (Field Programmable Gate Array) sont des architectures reconfigurables à grain fin et cette propriété leur confère une très grande flexibilité. Actuellement, ce sont les architectures reconfigurables les plus utilisées : leurs capacités leur permettent l'intégration de systèmes de plusieurs millions de portes équivalentes, et leur flexibilité de configuration autorise une latitude de conception presque aussi importante que les ASIC au niveau numérique.

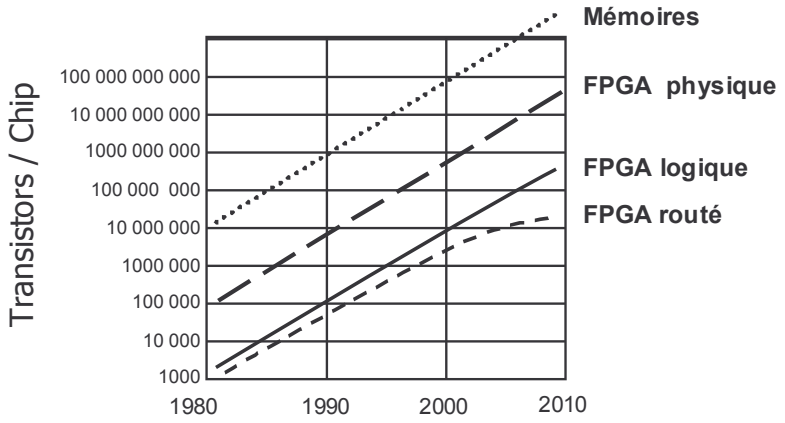


FIG. 25 - *Densités d'intégration sur composants reconfigurables : La structure régulière autorise une augmentation des densités d'intégration obtenues proportionnelles aux technologies silicium (FPGA physique). Cependant, le FPGA logique, i.e. en nombre de portes équivalentes, est d'un ordre 100 inférieur au nombre de portes réellement présentes sur le FPGA physique, ce qui qualifie le surcoût lié à l'utilisation de d'éléments à granularité fine.*

Cependant, l'utilisation d'une structure à grain fin induit un surcoût relativement important tel que cela est illustré dans la figure 25. D'une part, la difficulté d'implantation des fonctions dans un FPGA provient de la non-régularité des interconnexions de blocs logiques. D'autre part, lorsqu'on cherche à implanter un algorithme sur une architecture donnée, il est nécessaire de prendre en compte la largeur des données manipulées par l'application. Ainsi, pour une architecture spécifique, la largeur du chemin de données sera dimensionnée suivant le format des variables, donc pas ou peu de surcoût matériel mais aussi très peu de flexibilité. Une architecture flexible telle qu'une architecture FPGA a pour vocation de s'adapter au traitement et implicitement à différentes tailles de données. Les architectures à grain fin possèdent une flexibilité telle, qu'elles ont a priori la possibilité de s'adapter à n'importe quelle taille de variable. Mais cette flexibilité se paie par un surcoût matériel non négligeable, et qui peut avoir des conséquences également sur la fréquence de fonctionnement et la puissance consommée. Les fonctions arithmétiques manipulant des données multiples de l'octet sont donc sujettes à ces surcoûts matériels. Par exemple, les multiplieurs qui sont eux-mêmes composés d'une série d'additionneurs sont fortement pénalisé par l'utilisation de logique à granularité fine.

Dans [Sass02], la synthèse d'un multiplieur 16 bits illustre cette dégradation engendrée par la granularité fine sur la fréquence de fonctionnement, ainsi que le taux d'occupation qui laisse présager des capacités de réalisation intégrant une grande quantité de multiplieurs. Les travaux de R.J.Petersen et de B.L.Hutchings [Pete95] qui ont comparé les performances entre des implémentations de multiplieur sur FPGA et sur ASIC montrent que la perte est d'un facteur 4 à 10 selon la complexité de la logique à implémenter.

Tab. 5. *Synthèse d'un multiplieur 16 bits x 16 bits [Sass02]*

	VIRTEX 50K	*VIRTEX 1000K*	*ASIC 0.18µ / 12mm²*
SURFACE	18 %	1.2 %	0.02mm² / 0.16 %
FREQUENCE	10 MHz	33 MHz	180 MHz

Dans les domaines d'application où les données à manipuler ne sont pas des bits, mais des multiples de l'octet, on peut donc préjuger d'un surcoût lié à l'utilisation de granularité fine. C'est le cas notamment en TSI, où même si certains traitements manipulent le bit (suivi de contours, fermeture, chaînage, transformées, …), la majorité d'entre eux s'effectuent au niveau mot (détection de contours, DCT, FFT, FIR, IIR, transformée en ondelettes, estimation de mouvement …). Dans notre domaine d'application, il est donc nécessaire de noter que choisir une granularité fine aura un impact certain sur les performances, ce qui peut devenir une énorme contrainte dans le cadre d'applications en temps réel.

Pour éviter les désagréments liés à l'utilisation systématique de la granularité fine, il est possible d'utiliser un réseau disposant d'une granularité adaptée aux traitements du domaine d'applications ciblé. En l'occurrence, en TSI, il s'agit d'une granularité épaisse où la largeur du chemin de données est généralement un multiple de l'octet. Ce type d'architecture dispose habituellement d'opérateurs arithmétiques câblés optimisés pour améliorer les performances du circuit. Certains ont même pris le parti d'abandonner carrément la logique à grain fin, afin de ne cibler qu'un certain domaine d'applications. Il existe cependant sur le marché des réalisations hybrides de FPGA [Virt04] mettant à disposition des unités de traitement arithmétiques dispersées dans de la logique à grain fin.

2.1.2. Système d'interconnexion

C'est par association des éléments de traitement suivant un motif déterminé par la configuration, que l'architecture peut remplir des fonctionnalités différentes. Le choix du type d'interconnexion et la topologie ont donc une importance capitale dans les aptitudes du composant, qu'il s'agisse de performances ou de flexibilité. Nous allons dans cette partie référencer les différentes approches possibles pour le système d'interconnexion.

a. Type d'interconnexion

- Connexions globales : Ce type de connexion permet à n'importe quelle unité connectée sur le bus d'accéder à n'importe quelle autre située sur le même bus par le biais, par exemple, d'un *crossbar*[1]. Le motif d'interconnexion et le sens du flot de données sont déterminés par la configuration qui réalise les points de connexions entre les différentes unités.
- Connexions locales : Pour ce type de connexion, seul des éléments de traitement voisins peuvent se transmettre des données. Dans ce cas, des bus locaux dédiés ou bien des *switch* permettent d'acheminer avec plus ou moins de flexibilité des données d'un élément à un proche voisin grâce à un arbre de multiplexeurs.

b. Topologie d'interconnexion

Les architectures reconfigurables modernes, grâce aux technologies d'intégration actuelles, utilisent le plus souvent une hiérarchie dans le système d'interconnexion et ne se limitent que très rarement à un type d'interconnexion.

[1] Crossbar : terme anglais qui signifie commutateur

D'autre part, les topologies d'interconnexion sont extrêmement diversifiées. Celles-ci peuvent aller de structures relativement simples, linéaires 1D, à des hypercubes en 3D. La figure 26 donne trois exemples classiques de topologies 2D, avec plusieurs procédés d'interconnexions locales.

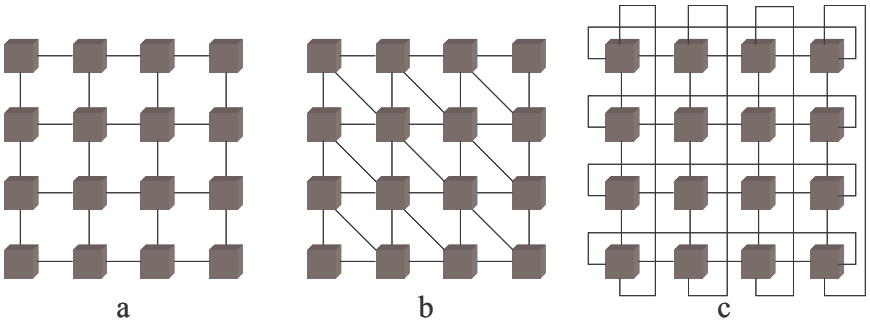


FIG. 26 - *Illustration schématique de trois topologies basées sur un maillage local d'interconnexion à 2 dimensions. (a) Structure Orthogonale (b) Structure Hexagonale (c) Structure Torique*

2.1.3. Gestion des reconfigurations

La gestion des reconfigurations est primordiale dans l'efficacité de la prise en charge des traitements. En effet, celle-ci peut induire une réactivité plus ou moins grande d'un système sur son environnement. Dans le cas de systèmes de sécurité par exemple, des traitements spécifiques doivent être configurés relativement vite afin de répondre en temps réel aux besoins.

Pour les architectures reconfigurables, on distingue deux niveaux : l'étendue de la reconfiguration et la rapidité de configuration. Ces deux niveaux sont par ailleurs souvent liés.

a. Statique / Dynamique

On peut distinguer deux niveaux de configuration :

- Configuration statique : les circuits reconfigurables de cette catégorie conservent la même configuration durant toute la phase de fonctionnement, *i.e.* le contenu du plan mémoire reste figé après chargement de la configuration.
- Configuration dynamique : tout ou partie de la configuration de l'architecture est modifiée en cours même de fonctionnement, sans nécessiter d'interruption des traitements en cours.

b. Partielle / Globale

La reconfiguration dynamique consiste à permettre un multiplexage temporel des contextes. Une autre approche consiste à permettre de ne reconfigurer qu'une partie du circuit à la fois. Les premières générations de FPGA possédaient une configuration globale. Ceci impliquait que l'ensemble du circuit devait être reconfiguré pour chaque changement de contexte. Xilinx, depuis le VIRTEX et ATMEL avec son AT6000 [AtmE04], autorise de reconfigurer leurs composants de manière partielle. La configuration est alors plus rapide car moins volumineuse et permet au reste du circuit de continuer à fonctionner normalement.

2.2. Architectures à grain fin

Dans ce paragraphe, il sera question d'architecture reconfigurable à grain fin. Les circuits basés sur de la logique à granularité fine sont majoritairement appelés FPGA. Ces circuits sont basés sur le schéma de principe de la figure 24. Leurs unités de calcul sont

souvent appelées Configurable Logic Block ou CLB, et ces derniers sont interconnectés au moyen de liaisons locales et/ou globales, avec des topologies variant d'une approche à l'autre. Les dernières évolutions techniques consistent majoritairement dans la gestion de la configuration ou encore l'insertion d'opérateurs spécialisés. Nous ne nous attarderons que sur des exemples qui nous paraissaient significatifs, et pour un état de l'art complet des architectures de base, le lecteur pourra se référer à [Brow96] et pour des machines basées sur des FPGA, à [Gucc94].

2.2.1. Eléments de calcul

a. Principe général

La brique de base des éléments de calcul d'une architecture à grain fin est la Look-Up Table ou LUT. Il s'agit d'un bloc entièrement combinatoire programmable composé de n entrées $E_0 \ldots E_{n-1}$, d'une sortie et d'un tableau vertical de 2^n cellules mémoires connectées à un multiplexeur 2^n vers 1 (Figure 27). La LUT est capable de réaliser toutes les fonctions possibles à n entrées. Pour cela, chacune de ses cellules mémoires est programmée selon la table de vérité de la fonction donnée, par exemple la fonction ET dans la figure 27.

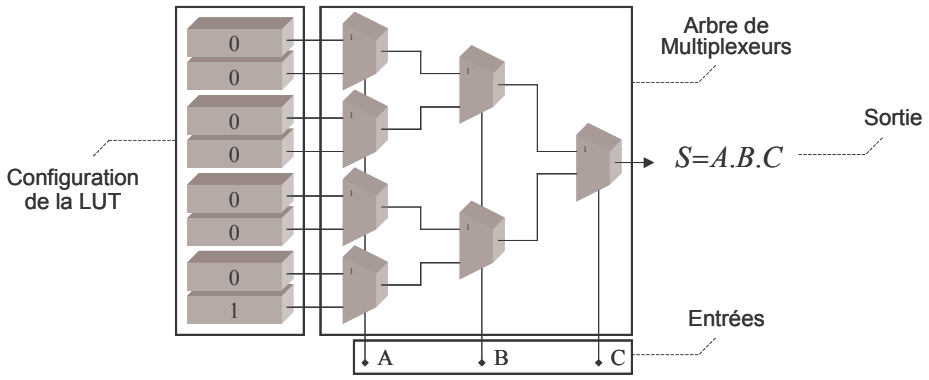


FIG. 27 - *Schéma de principe d'une LUT : Les entrées logiques de la LUT sont reliées aux entrées de sélection des multiplexeurs. Suivant le contenu de la mémoire de configuration (table de vérité) connectée au premier étage de multiplexeurs, la LUT génère une sortie logique fonction des entrées A, B et C. Dans l'exemple proposé, la table de vérité correspond à la réalisation d'une fonction logique ET à trois entrées.*

L'utilisation de LUT comporte de nombreux avantages. Outre la flexibilité, la complexité de l'équation booléenne implantée ne dépend pas du nombre de portes logiques à implanter, mais du nombre d'entrées de cette équation et le délai de propagation ne dépend pas de la complexité de l'équation booléenne.

Les éléments de calcul sont donc constitués en tout premier de LUT. Le nombre de LUT par élément de calcul peut varier d'un fabricant à un autre, mais l'association de LUT au sein d'un même élément logique permet de réaliser des fonctions logiques avec de multiples entrées à moindre coût matériel. De plus, afin de permettre la mise en œuvre de fonctions séquentielles, des bascules D *Flip Flop* (D*ff*) sont intégrées aux éléments de calcul et un multiplexeur situé en sortie de l'élément permet de choisir ou non cette option. La figure 28 représente un élément de calcul générique pour une architecture à grain fin. On notera également qu'un certain nombre de mécanismes supplémentaires, comme par exemple l'implantation d'une chaîne de retenues, peuvent également apparaître dans les éléments de calcul de base.

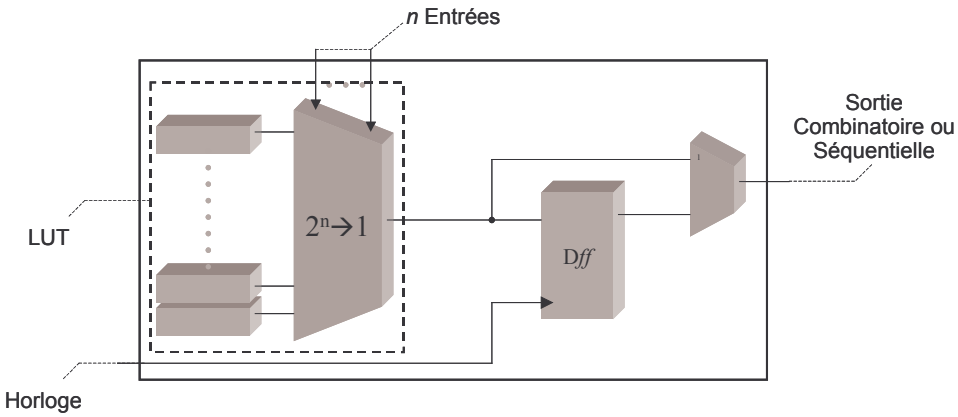


FIG. 28 - *Architecture simplifiée d'un élément de calcul d'une architecture à grain fin. Une ou plusieurs LUT sont associées à une ou plusieurs bascules. Un multiplexeur de sortie permet de choisir la sortie séquentielle ou combinatoire.*

b. Exemple

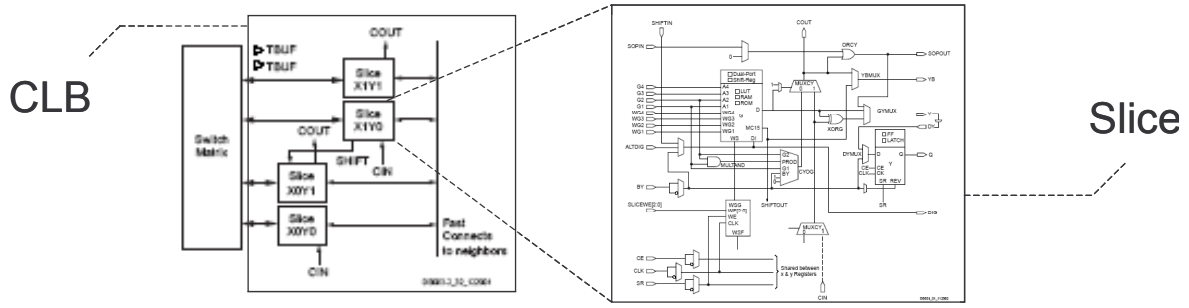


FIG. 29 - *Description détaillée d'un CLB et d'un Slice Xilinx [Xili00]. Le CLB est constitué de 4 Slices organisés en colonnes qui peuvent être localement connectés pour implémenter des chaînes de retenues ou encore des registres à décalage. Les slices permettent de générer des fonctions logiques combinatoires ou séquentielles ou encore des éléments de mémorisation*

Les deux leaders du marché des FPGA, Xilinx et Altera, proposent des éléments de calcul de base relativement similaires. A titre d'exemple, nous proposons d'exposer l'architecture d'un CLB (Configurable Logic Block) d'un FPGA Xilinx. Celui-ci est constitué de 4 slices identiques organisés en 2 colonnes qui possèdent chacune leur propre chaîne de retenue et un registre à décalage commun. Chaque Slice possède 2 générateurs de fonctions logiques à 4 entrées, une logique dédiée à la chaîne de retenue, des bascules et des multiplexeurs de sélection. De plus, deux éléments de stockage permettent d'utiliser le slice comme un élément mémoire distribué.

2.2.2. Interconnexion des éléments de calcul

Nous présentons ici des exemples de type d'interconnexion et de topologies utilisées dans différents FPGA. Le type d'interconnexion et la topologie sont deux éléments fortement liés car suivant la localité des éléments, on préfèrera certains types de liaisons à d'autres. Avec l'augmentation des densités d'intégration et par voie de conséquence, du nombre d'éléments de calcul logique, la hiérarchisation du système d'interconnexion s'est généralisé à la plupart des FPGA commerciaux. A titre d'exemple, l'APEX II d'Altera [Apex01] utilise trois niveaux de hiérarchie (les LE (Logic Element), les LAB qui regroupent les LE ensemble et les MegaLAB qui regroupent les LAB entre eux) ou bien encore les CLB

des Virtex avec 4 slices (voir Figure 29). L'approche hiérarchique permet d'économiser des ressources en interconnexion puisque chaque niveau est assigné à un traitement local.

a. Type d'interconnexion

Dans les FPGA récents, on distingue donc plusieurs niveaux hiérarchiques de connexions des différents blocs de calcul. Ainsi, des bus locaux permettent souvent d'interconnecter un faible nombre d'unités de traitement logique, (CLB du Virtex II Pro regroupent 4 cellules de bases, les LAB d'Altera intègrent de 8 à 10 LE), et des *crossbar* (figure 30) permettent d'une manière globale, de connecter les différents clusters sur des distances plus ou moins longues suivant les besoins. Des lignes générales sont en effet couplées avec des lignes double distance, ce qui limite le nombre de passages par des commutateurs.

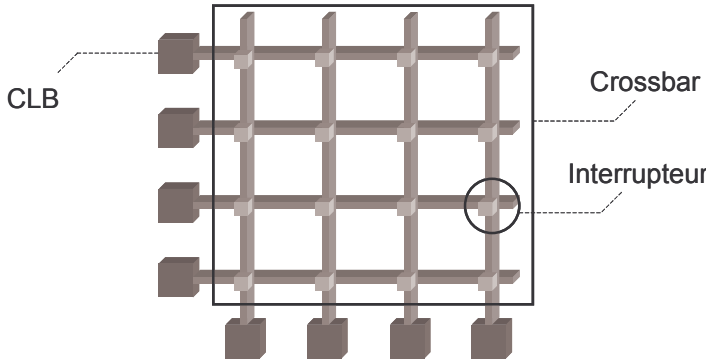


FIG. 30 - *Exemple de crossbar : n'importe quel CLB peut transmettre ou recevoir des données de n'importe quel autre suivant la configuration des interrupteurs de la grille.*

b. Topologie d'interconnexion

La topologie permet par le biais de différents types de connexions de connecter les générateurs de fonctions logiques avec plus ou moins de flexibilité et plus ou moins de matériel. Nous allons tout d'abord exposer des exemples de structures de maillage global puis celles des exemples de structures locales.

- Maillage global

La première méthode consiste à ne connecter directement que des éléments de la même ligne (colonne). Les cellules sont organisées sous forme de lignes (ou de colonnes) et possèdent des connexions horizontales (verticales) pour amener les informations d'une colonne à l'autre (figure 31). Le réseau de connexions est fixe et permet de passer des informations d'une ligne à l'autre, ce qui est plus particulièrement adapté à des traitements du type «flot de données». A titre d'exemple, nous pouvons citer la structure de routage des familles 40 MX et 42 MX d'ACTEL [Acte04].

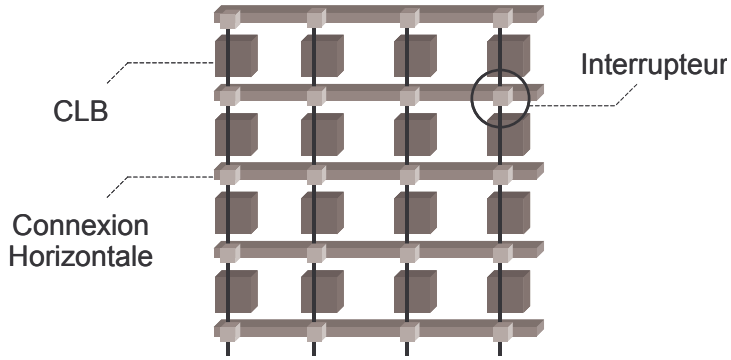


FIG. 31 - *Maillage global horizontal : les données sont passées d'une ligne à l'autre via des liaisons dédiées. Pour passer une donnée d'une colonne à une autre, des connexions horizontales peuvent être utilisées.*

Une deuxième méthode consiste à ajouter un degré de flexibilité à la première, en autorisant des connexions à la fois verticales et horizontales (figure 32). Ce type de modèle symétrique a comme principal avantage d'être facile à prendre en compte par l'outil de compilation. Cependant le nombre d'interconnexions nécessaires pour assurer la pleine connectivité croit avec le carré du nombre de CLB : cela justifie l'utilisation d'un second réseau de connexions locales qui cohabite avec le réseau symétrique afin d'assurer les connexions de voisins à voisins.

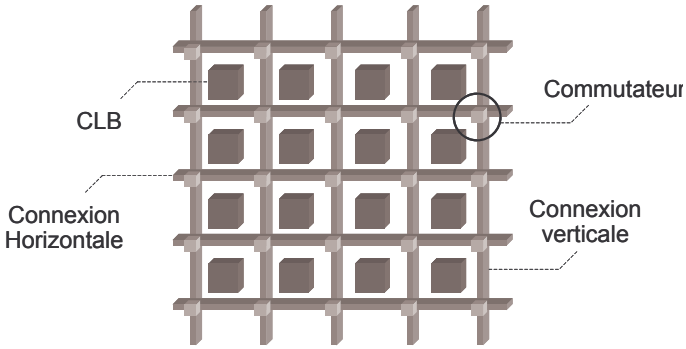


FIG. 32 - *Maillage global symétrique: les données d'un CLB peuvent être acheminées à n'importe quel autre CLB grâce aux différents commutateurs de routage.*

- Maillage local

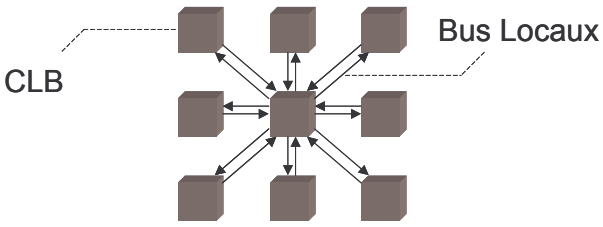


FIG. 33 - *Maillage local des cellules d'un AT40K d'ATMEL : chaque CLB du réseau est connecté à chacun de ses 8 voisins par l'intermédiaire de bus locaux dédiés*

A ce niveau d'interconnexion, chaque élément de traitement peut se connecter à un certain nombre d'éléments voisins. Ce type d'organisation est bien adapté au traitement local des données, comme par exemple, la réalisation de fonctions arithmétiques. La structure

du AT40K d'ATMEL [Atme04] est basée sur une structure de maillage local où chaque CLB peut être connecté à ses 8 voisins (figure 33).

2.2.3. Méthodes de reconfiguration

Plusieurs initiatives universitaires et industrielles peuvent être citées en ce qui concerne les méthodes de reconfiguration. Dans des approches de couplage processeur programmable FPGA, on peut citer le projet GARP à Berkeley [Haus00], et le projet NAPA proposé par National Semiconductor [Rupp98]. Ces deux architectures exploitent des blocs reconfigurables conçus au sein même des laboratoires. Il ne s'agit pas ici de configurer à l'initialisation mais de modifier la configuration à chaque fois que le processeur hôte rencontre un traitement critique. La reconfiguration du FPGA intervient dynamiquement au cours de l'exécution du programme. Les techniques utilisées sont basées sur la **reconfiguration partielle** et des composants multi-contextes. La **reconfiguration multi-contextes** consiste à mémoriser plusieurs configurations et le passage d'une configuration à une autre ne nécessite alors qu'une commutation entre les différents plans de configuration du système. Ceci permet de commuter rapidement d'une configuration à une autre mais se paie par des mémoires de configuration démultipliées. Dans le cadre de GARP, plusieurs plans de configuration sont stockées dans des mémoires caches ce qui permet de réduire les cycles d'accès à la mémoire principale. La reconfiguration partielle du bloc reconfigurable permet également de ne reconfigurer que certaines parties et donc d'entrelacer les phases de configuration et de traitement.

Une méthode originale est également mise en œuvre dans le cadre du projet ARDOISE [Demi99] (figure 34), dans lequel trois modules à base de FPGA (ATMEL AT40K) sont reconfigurées par un DSP (Sharc 21062 d'Analog devices). Les trois modules d'Ardoise communiquent à travers un bus de 64 bits. Le module central appelé UC (unité de calcul) permet d'effectuer les calculs requis par les algorithmes. Les autres modules appelés GTI (Gestionnaires de Tampon Image) jouent le rôle d'interface entre l'UC et le système d'acquisition. Les mémoires présentes dans les GTI servent de tampon de données. Le système peut faire d'une part l'acquisition d'une image n+1, traiter l'image n à partir d'un premier algorithme (flot de données de (1) vers (3)), et restituer l'image n-1. Après reconfiguration dynamique de l'UC par le DSP, le système applique un deuxième traitement à l'image après inversion du flot de des données (de (3) vers (1)). Ici, la **reconfiguration partielle** s'établit au **niveau** du **système**.

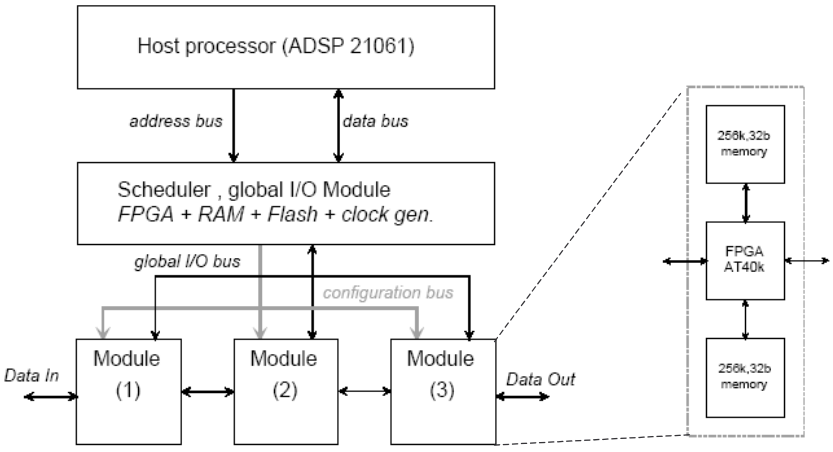


FIG. 34 - *L'architecture ARDOISE*

Une autre contribution dans les méthodes de reconfiguration est celle proposée dans le cadre du projet Symptôme [Blan02], notamment, un contrôleur de configuration implanté

dans de la logique à grain fin sous forme de réseaux de Pétri, reconfigure de manière conditionnelle une partie reconfigurable de l'architecture dédiée aux traitements, et ce suivant l'évolution de l'application.

2.2.4. Innovations architecturales récentes

Les limitations de la logique à grain fin pour l'implantation d'opérateurs arithmétiques notamment, ont motivé les concepteurs de FPGA à intégrer des éléments de calcul à grain épais au sein même de la logique à grain fin. De plus, la quantité de données à manipuler justifie également la mise en œuvre de véritables bancs mémoires entièrement configurables. En effet l'utilisation des LUT comme mémoires est limitée à des mémoires de faible capacité. Le constructeur Xilinx va même encore plus loin dans l'intégration des ressources : le VIRTEX II Pro [Virt04] peut embarquer jusqu'à 4 processeurs RISC au sein de son circuit, ce qui lui confère de véritables caractéristiques de système sur puce entièrement configurable.

Dans cette partie, nous allons exposer brièvement ces innovations importantes dans le domaine des FPGA, avec deux exemples qui nous semblaient significatifs des avancées technologiques : le STRATIX II [Strat04] de Altera et le VIRTEX II PRO de Xilinx.

a. VIRTEX II PRO

XILINX a introduit dans l'architecture du VIRTEX une cellule mémoire appelée SelectRAM. Cette cellule est basée sur une mémoire double port configurable de 18 Kbits. La mémoire peut être utilisée en simple port ou en double port avec des données de 1, 2, 4, 9, 18, 36 bits. Chacun des ports est synchrone mais leurs horloges peuvent être différentes. Pour chaque octet de données il est possible de stocker un bit de parité, ce bit doit être généré et vérifié dans de la logique externe à la SelectRAM. Chaque port peut travailler sur des données de tailles différentes.

En termes d'opérateurs à grain épais, l'architecture du VIRTEX intègre des cellules de multiplication signées 18x18 bits. Chacune de ces cellules peut être alimentée soit par les cellules de base soit par la mémoire SelectRAM. L'utilisation des mémoires pour alimenter les multiplieurs permet d'éviter d'éventuels goulots d'étranglement. De plus la présence d'additionneurs soustracteurs permet d'obtenir un éventail de possibilités pour des opérations typiques de TSI (somme ou différences de produits, produits de somme). Le VIRTEX II Pro possède de 14 à 328 multiplieurs 18*18bits suivant les versions du produit.

Enfin, la dernière évolution notoire est sans aucun doute l'intégration de cœurs de processeurs au sein du FPGA. Il s'agit de cœurs de PowerPC 405-D5 RISC avec un chemin de données sur 32 bits.

b. STRATIX II

Au niveau de la mémoire, ce circuit propose une hiérarchie mémoire nommée TriMatrix. Elle est basée sur trois blocs mémoires. Les fonctionnalités de ces cellules sont équivalentes à celle du VIRTEX avec en plus, une configuration FIFO des mémoires et la possibilité de réaliser des registres à décalage. La différence entre les trois mémoires repose principalement sur leurs capacités de stockage afin d'offrir une plus grande souplesse d'utilisation. Ainsi le STRATIX dispose de mémoires de 512 bits, 4K bits et de 512K bits. Les blocs mémoires peuvent être assemblés entre eux de façon à augmenter la taille de leurs bus de données ou leurs capacités.

En ce qui concerne les opérateurs câblés, le STRATIX II offre suivant la version, un nombre variable de blocs nommés DSP. Ils permettent de disposer de 48 à 384 multiplieurs 18*18 bits. Suivant la configuration ces blocs DSP peuvent fonctionnés en Multiplieur, MAC

(Multiplication ACcumulation), somme de 2 ou 4 produits. Afin de rendre possible l'implémentation de filtres (FIR, IIR,...) il est possible de configurer les entrées des multiplieurs pour réaliser un registre à décalage. Les cellules sont *cascadables* et permettent la mise en place de plusieurs niveaux de pipeline.

2.3. Architectures à grain épais

Après avoir exposé les principaux aspects des architectures reconfigurables à grain fin, nous allons maintenant nous focaliser sur celles à grain épais. Ces circuits sont également basés sur le schéma de principe de la figure 24. Nous décrirons dans un premier temps les éléments de calcul de base de ces composants puis nous donnerons quelques exemples de réalisations académiques ou industrielles. Les différentes topologies d'interconnexions seront ensuite évoquées. Enfin, les méthodes de reconfiguration que nous avons jugé intéressantes pour notre état de l'art, seront exposés dans le dernier paragraphe. Le lecteur pourra également se référer à [Hart01] qui recense la plupart des architectures reconfigurables à grain épais..

2.3.1. Eléments de calcul

a. Principe général

La brique de base des éléments de calcul d'une architecture à grain épais est le bloc fonctionnel configurable ou CFB[1]. Il est constitué au minimum d'une UAL et d'un registre de configuration. L'UAL a une granularité minimale de 8 bits. La figure 35 représente un CFB minimal.

Dans l'état de l'art, on peut référencer différentes évolutions de ce modèle. Ainsi, certaines approches intègrent au sein de leur CFB un registre à décalage permettant de réaliser des opérations en virgule flottante (Colt [Bitt96], PipeRench [Gold99]), un ou plusieurs Multiplieurs (MorphoSys [Sing98], Systolic Ring [Sass02], DART [Davi02]) , des systèmes de propagation de retenues, permettant de chaîner plusieurs CFB ou encore des bancs mémoires utilisés soit comme registres (Systolic Ring, MorphoSys) soit comme mémoire distribuée (DART, PACT [Baum01]).

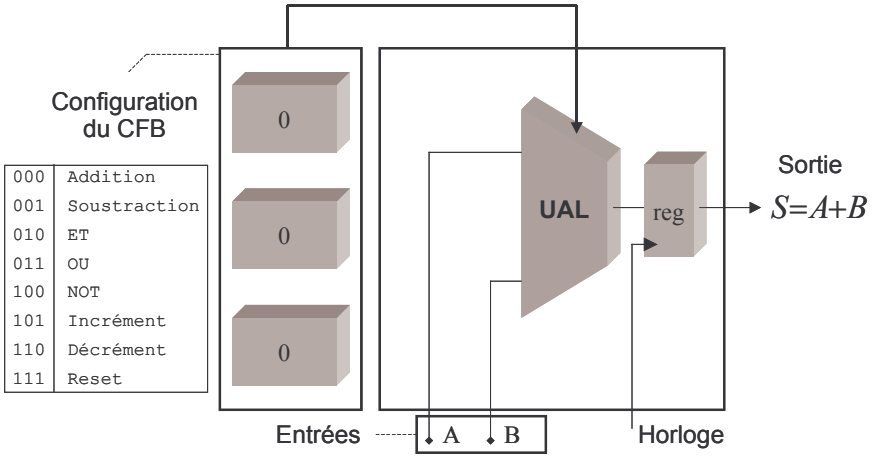


FIG. 35 - *Exemple de CFB minimal : un mot de configuration de 3 bits permet de sélectionner la nature de l'opération arithmétique à réaliser sur deux entrées. La sortie du CFB est synchronisée par une horloge.*

[1] CFB : Configurable Functional Block

La hiérarchie d'interconnexion peut être introduite à l'intérieur même du CFB. Ainsi, il est vu comme un opérateur complexe permettant de générer à partir de ses primitives matérielles (ALU, Multiplieurs, registres) des fonctions arithmétiques *pipelinées* complexes. C'est le cas, par exemple, des architectures Rapid [Cron98] ou DART. Cette caractéristique confère à ces CFB une grande flexibilité du matériel au prix d'une programmation et une configuration plus complexe.

Afin de simplifier la programmation et le volume de configuration de l'architecture, le CFB peut-être vu comme un simple macro-opérateur capable de générer des fonctions arithmétiques de base prédéfinies. Les architectures MorphoSys ou Systolic Ring sont des illustrations de ce principe. L'inconvénient majeur de cette approche réside dans une sous utilisation potentielle du matériel.

b. Exemples

Deux exemples significatifs des deux types de CFB pouvant être mis en œuvre pour des architectures à grain épais sont le DPR [Davi02] de DART et le RC [Sing98] de MorphoSys.

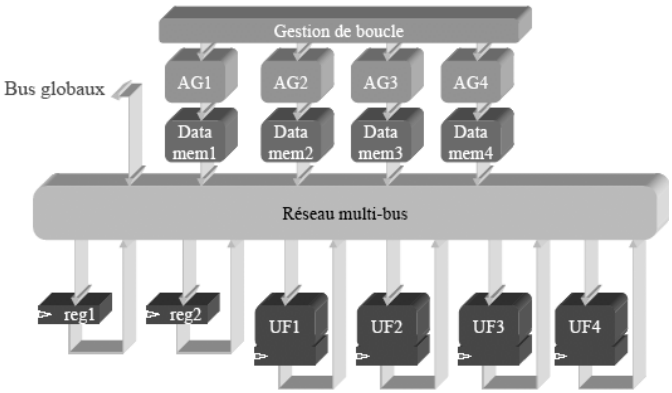


FIG. 36 - *Le Data-Path Reconfigurable de l'architecture DART. Les unités fonctionnelles consistent en 2 UAL et deux multiplieurs. Deux registres peuvent être également utilisés pour réaliser des retards. Des mémoires sont également directement associées au plus bas niveau des primitives de calcul. Le réseau multi-bus permet une flexibilité totale de connexion des différents éléments. Ainsi, des fonctions arithmétiques complexes peuvent être générées.*

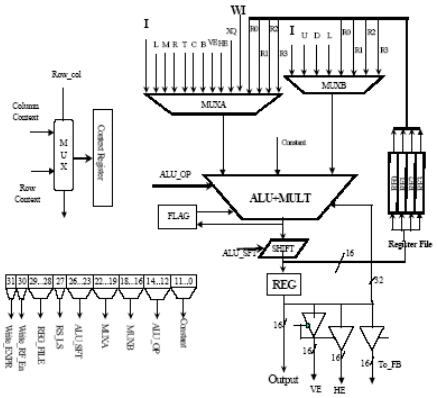


FIG. 37 - *Une cellule reconfigurable (RC) de l'architecture MorphoSys. Ce CFB de type macro-opérateur est composé d'un bloc, ALU+Multiplieur.*

2.3.2. Interconnexion des éléments de calcul

a. Type d'interconnexions

A l'instar des FPGA récents, les architectures reconfigurables à grain épais proposent pour la plupart des réseaux de connexions hiérarchiques. Elles utilisent soient des bus locaux (Matrix [Mirs96], MorphoSys) qui regroupent un sous ensemble de CFB, soient des réseaux multi-bus (DPR de DART, Rapid) pour des connexions locales entre opérateurs ou bien encore, des *switch* (Piperench, Systolic Ring) pour tout ce qui concerne les communications de proches voisins, la plupart du temps de manière unidirectionnelle. Pour les communications plus longues, les bus sont connectés à travers des *crossbars* (DART) ou encore des *switch*.

b. Topologies

Les topologies des architectures des architectures à grain épais peuvent être classées en trois catégories : les réseaux de type grille, les réseaux de type linéaire et les réseaux de type centralisé.

Dans la première catégorie, on a par exemple l'architecture MorphoSys avec 64 éléments de calculs répartis sur une grille homogène carrée de 64 CFB ou encore le Xputer [Xput01]. Ces architectures possèdent généralement une hiérarchie d'interconnexion permettant des communications locales rapides par l'utilisation de bus dédiés et des bus plus ou moins longs pour les liaisons longues distances.

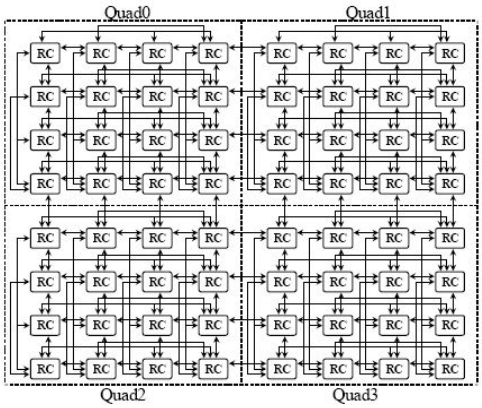


FIG. 38 - *Topologie d'interconnexion basée sur une grille hiérarchique. L'architecture MorphoSys propose ce schéma de connexion pour ses cellules reconfigurables (RC : Reconfigurable Cell)*

Dans la catégorie des architectures linéaires, nous pouvons recenser les architectures Rapid, Piperench et Systolic Ring. L'utilisation de plusieurs couches d'opérateurs permet de mettre en œuvre un pipeline de configuration et de traitement : quand on configure la couche *n+1*, on commence le traitement à la couche *n*, et ainsi de suite. De plus, la succession de plusieurs couches accroît la profondeur du pipeline de traitement, ce qui correspond à une utilisation soutenue de parallélisme temporel.

Enfin, on peut classer dans les topologies centralisées les architectures DART et PADDI. Un ensemble de CFB est relié à un *crossbar* central. Dans ce cas, le CFB est composé d'un nombre limité d'opérateurs de base et d'un réseau d'interconnexion complet qui autorise à peu de choses près, n'importe quel câblage entre les différentes unités (opérateurs, mémoires locales, registres etc.).

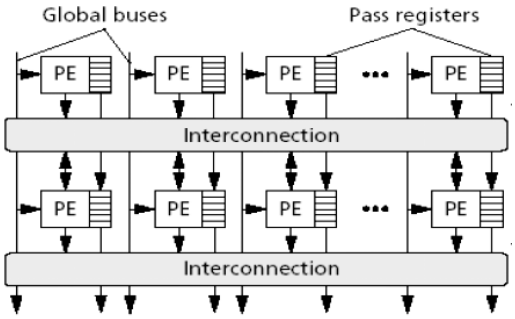


FIG. 39 - *Topologie d'interconnexion linéaire. L'architecture de Piperench illustre ce schéma de connexion avec la succession des différentes couches de CFB (PE : Processing Element) séparés par des switch d'interconnexion également configurables*

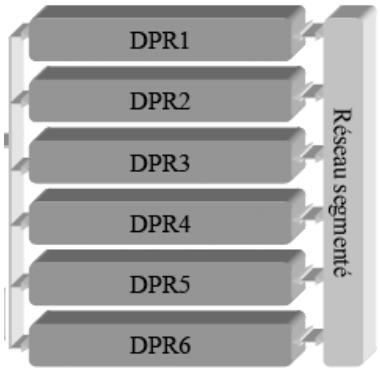


FIG. 40 - *Exemple d'architecture conçue autour d'un crossbar central : les différents chemins de données reconfigurables possèdent des bus pouvant être interconnectés via une matrice de commutateurs.*

Notons par ailleurs, que dans chacune des approches, la hiérarchie et la topologie confèrent à l'architecture un degré de flexibilité plus ou moins grand. Elles peuvent également être un facteur de limitation de l'exploitation des différents niveaux de parallélisme.

2.3.3. Méthodes de reconfiguration

Les architectures reconfigurables à grain épais nécessitent, du fait de leur granularité, moins de bits de configuration. De manière générale, les architectures à grain épais sont idéales pour la reconfiguration dynamique, leur procédé de configuration étant souvent très simple à mettre en œuvre. Ce dynamisme confère à ces composants un caractère réactif vis-à-vis de leur environnement. En d'autres termes, suivant les besoins de calcul, il est possible de moduler le matériel en peu de temps, voire en temps réel. L'étendue de la reconfiguration dynamique et les procédés de reconfiguration diffèrent d'une approche à une autre : c'est ce que nous allons analyser dans cette section.

Pour les architectures linéaires telles que PipeRench et le Systolic Ring, un étage d'opérateurs peut-être configurée à chaque cycle, pendant que le reste des CFB fonctionne et traite des données. Le pipeline présent dans le traitement des données se retrouve donc dans la méthode de configuration de ce type d'architecture, où la **reconfiguration dynamique partielle** est mise en œuvre.

Dans d'autres projets comme Rapid ou DART, certains concepteurs ont fait le choix de différencier configuration du chemin de données (interconnexions) et configuration des

opérateurs (CFB). La première nécessite un nombre relativement important de données de configuration (quelques centaines, voire milliers de bits) qui sont maintenues pendant toute la durée d'un traitement. La seconde est quant à elle caractérisée par un volume de données de configuration limité (quelques dizaines de bits) mais est amenée à évoluer très souvent. Elle permet par exemple de spécifier un reset ponctuel des accumulateurs ou d'initialiser certaines variables avant de rentrer dans une boucle. Dans cette optique, les concepteurs de DART proposent **deux flots de configuration distincts** : l'un qualifié de Hardware pour lequel le chemin de données est entièrement reconfiguré, l'autre Software qui ne permet de reconfigurer que la fonctionnalité des ALU. Suivant la méthode utilisée, le temps de configuration sera plus ou moins important.

Pour l'architecture Rapid, les concepts de **pipeline de configuration** et de **distinction des flux de configuration** sont mis en œuvre. Comme le traitement de RAPID est systolique, l'idée consiste à *pipeliner* les informations de configurations dynamiques en même temps que les données.

D'autres approches telles que Chameleon [Cham00] proposent l'utilisation de **plusieurs contextes de configuration** stockés en cache. Dans ce cas, le temps de configuration est extrêmement rapide car il consiste à seulement multiplexer le contenu d'une mémoire multi-contextes sur les registres de configuration du composant. Les différents plans de configuration autorisent une reconfiguration partielle et dynamique du bloc reconfigurable puisqu'un plan de configuration peut être modifié par le contrôleur pendant qu'un second spécifie la fonctionnalité du bloc reconfigurable.

Une autre approche intéressante est celle de l'architecture PACT XPP, où un bit d'état est associé à chaque opérateur. Ainsi, un gestionnaire de configuration qui connaît l'état de ses ressources peut charger plus rapidement des nouvelles configurations une fois les traitements achevés. Les concepteurs de PACT présentent la **méthode de configuration** comme **partielle et différentielle**, c'est à dire que la configuration traduit les changements qui interviennent lors du passage d'un contexte à l'autre. Ces mécanismes autorisent le système à configurer une partie de l'architecture sans perturber le fonctionnement du reste. Ainsi, plusieurs sous-traitements peuvent cohabiter dans la structure et gérer de façon indépendante leurs reconfigurations. La dernière innovation de PACT réside dans la possibilité qu'un traitement demande sa propre reconfiguration. Des mécanismes d'auto-reconfiguration similaires sont également évoqués dans le projet MATRIX.

Pour des architectures basées sur des grilles d'opérateurs telles que MorphoSys, on notera un **processus de configuration basé sur la géométrie de la topologie**. Il est par exemple possible de configurer successivement les lignes ou les colonnes d'opérateurs à chaque cycle d'horloge. Cette architecture permet également des configurations multi-contextes.

Plus récemment, au CEA, un projet nommé RAMPASS a pour objectif de fusionner l'approche d'auto-reconfiguration conditionnelle proposée dans le projet Symptôme (Reconfigurable grain fin Adapté au Contrôle) et les ressources grain épais de DART (Reconfigurable grain épais Adapté aux Opérations). Le *séquencement* des configurations est basé sur le bloc RAC de SYMPTOME qui projète sous forme de graphe de Pétri le flot de contrôle de l'application, et le flot de données est pris en charge par un cluster de DPR de DART.

Il est à noter également que la plupart de ces architectures proposent plusieurs modes de configuration, comme par exemple des **modes SIMD**, où l'ensemble des opérateurs est configuré de la même manière en un seul cycle. Par souci de concision, nous n'avons évoqué ici que ceux qui nous paraissaient intéressants en termes d'innovations techniques.

3. BILAN QUALITATIF

Nous avons jusqu'à présent exposé deux paradigmes et 4 types d'architectures pour le traitement des données numériques. Il est important de préciser que ces approches ne sont pas nécessairement concurrentes. Au contraire, elles sont souvent complémentaires : on utilisera vraisemblablement toujours un cœur de processeur à usage général afin de prendre en charge la gestion du système.

Suite à cet état de l'art, nous allons discuter les aspects principaux pour le domaine des applications TSI dans un contexte SOC. Ainsi nous analyserons les différentes approches sous l'angle du parallélisme et de ses conséquences sur les performances temporelles et la consommation, et nous terminerons par les aspects flexibilité et coût.

3.1. Exploitation des différents niveaux de parallélisme

Nous pensons que l'exploitation du parallélisme est un des points clé parmi les concepts architecturaux présentés dans ce chapitre ; cela a évidemment des conséquences immédiates sur les temps de traitement, mais aussi sur la consommation. Le tableau 6 dresse le bilan de l'exploitation du parallélisme par les différentes approches présentées dans notre état de l'art.

Tab. 6. *Bilan des aptitudes au parallélisme de chaque architecture. Ce tableau fait apparaître quatre niveaux de parallélisme : flot, boucle, spatial et temporel.*

		Flot (tâche)	*Boucle*	*Spatial*	*Temporel*
Processeurs Programmables	Superscalaire SMT	Oui (multi-thread)	Oui	Oui (<8)	Oui (faible)
	DSP VLIW	Non	Oui (automatique et/ou manuel)	Oui (<8)	Oui (faible)
Processeurs Reconfigurables	Grain Fin mixte	Oui	Oui (manuel)	Oui (variable)	Oui (variable)
	Grain épais	Théorique	Oui (manuel)	Oui (variable)	Oui (variable)

Précisons pour commencer que nous aurions pu également prendre en compte le parallélisme au niveau système, mais il concerne plus particulièrement le CPU et le système d'exploitation, mais aussi le parallélisme vectoriel qui traduit les possibilités de traitements SIMD, mais qui n'est pas d'un intérêt primordial dans le cadre de cette étude.

A la première lecture de ce tableau, on constate tout d'abord que la plupart des architectures sont capables d'exploiter les différents degrés de parallélisme. Mais si l'on retrouve dans la même catégorie de parallélisme certaines architectures, cela ne veut pas dire pour autant que la mise en œuvre en est identique. Ainsi par exemple, le déroulage de boucle est réalisé manuellement dans le cadre des architectures reconfigurables, où la duplication du motif de calcul est faite par l'utilisateur, suivant la quantité de ressources libres. Pour les DSP VLIW, des algorithmes intégrés aux outils de compilation peuvent se substituer au programmeur afin de produire des déroulages automatiques, mais prédéfinis. Une optimisation manuelle est cependant souvent la bienvenue pour améliorer la solution initiale produite par l'outil. Pour les architectures superscalaires, la distinction entre parallélisme de boucle et parallélisme d'instruction est mince puisque c'est le matériel qui reçoit des instructions séquentielles et qui déploie le parallélisme. Les cœurs de circuits de processeurs actuels ne possèdent pas plus de 8 unités de traitement ce qui limite à la fois le parallélisme spatial et temporel. Pour accroître le nombre d'unités de traitement, la tendance consiste à dupliquer le nombre de cœurs plutôt qu'augmenter le nombre d'unités à l'intérieur du cœur.

L'approche est autre pour les architectures reconfigurables qui ont pour vocation d'intégrer un nombre croissant d'unités de base, afin d'exploiter un maximum de parallélisme.

3.2. Compromis de performances

3.2.1. Performances temporelles

Il parait assez difficile de conclure sur les temps de traitement tant cela peut être fonction de l'architecture et de l'application. Ce que nous pouvons signaler de manière générale, c'est que chacune de ces architectures met à disposition des éléments susceptibles d'accroître les performances temporelles : fréquences de fonctionnement élevées, duplication des unités de calcul, outils logiciels pour l'exploitation du parallélisme etc. Cependant, sur la classe d'applications ciblées (TSI) *i.e.* des algorithmes ayant une forte connotation arithmétique et intrinsèquement parallèle, DSP et architectures reconfigurables semblent disposer d'un potentiel supérieur, en particulier pour les architectures à grain épais [Cal98]: en effet, la mise en œuvre de mécanismes spécifiques (matériels et/ou logiciels) leur permettant d'envisager tous les niveaux de parallélisme (de flot notamment) paraît tout à fait réaliste. Cependant il faut garder à l'esprit que les DSP ont au moins le mérite d'exister, avec de nombreux outils de développement disponibles, alors que les architectures reconfigurables à grain épais en sont encore à un stade de recherche et développement relativement précoce.

3.2.2. Consommation

Par rapport aux contraintes de performances évoquées ci-dessus, l'expérience montre que l'augmentation de la fréquence de traitement affecte la consommation, malgré la diminution des tensions d'alimentation. Jusqu'à présent, les fabricants de processeurs ont mis l'accent sur l'augmentation des fréquences de fonctionnement comme vecteur d'accélération. On comprend alors l'inconvénient que cela représente, notamment pour tout ce qui concerne les systèmes portables. Si l'on prend comme référence de puissance de traitement le MOPS (Million d'Opérations Par Secondes) qui est le produit de la fréquence de fonctionnement par le nombre d'opérateurs matériels, alors on peut conclure qu'à puissance de traitement égale, il est préférable de privilégier le nombre d'opérateurs à la fréquence, si l'on limiter la puissance dynamique consommée. C'est une raison supplémentaire qui nous pousse à penser que les architectures reconfigurables pourraient avoir les qualités d'un bon coprocesseur de traitement, à la fois performant et efficace d'un point de vue énergétique.

La puissance dynamique consommée est également dépendante de la capacité équivalente et de l'activité du circuit. Une implantation d'un traitement sur une cible impliquant un important surcoût matériel a donc de fortes chances d'augmenter la consommation de puissance. Dans un domaine d'application tel que le TSI, l'utilisation d'une granularité épaisse peut représenter une alternative aux architectures à grain fin souvent peu efficaces dans l'implantation de chemins de données utilisant des opérateurs arithmétiques.

Par rapport aux architectures dites séquentielles, telles que les processeurs et DSP, le chargement/décodage d'instruction intervient de manière systématique. L'intérêt des architectures reconfigurables se manifeste encore de ce point de vue, dans la mesure où la configuration n'intervient pas systématiquement, ce qui pourrait impliquer une consommation de « contrôle » plus faible. Il faut cependant prendre en compte également le volume des données de configuration, ainsi que la fréquence des configurations.

3.2.3. Flexibilité

La granularité est un des critères de flexibilité. Pour un FPGA, la souplesse est maximale puisque avec le même support matériel, on peut réaliser des fonctions logiques 1 bit

ou des fonctions arithmétiques 1024 bits : autrement dit, aucune limitation, si ce n'est le nombre de cellules logiques. Pour les architectures se basant sur des multiples de l'octet, c'est plus compliqué. En effet, revenir à des traitements au niveau bit signifie qu'il faut extraire ce bit d'un mot et lui appliquer un masque avant de l'injecter dans la structure de calcul logique : donc un certain nombre de transformations lourdes et inefficaces en termes de performances. Même si de nombreuses architectures permettent des traitements SIMD vectoriels sur des données plus petites que la largeur de leur chemin de données, la flexibilité résultante n'est pas aussi grande que celle des FPGA.

La souplesse de programmation est aussi un critère de flexibilité. Sur cet aspect, les microprocesseurs dominent largement les autres architectures. Une simple description de l'application dans un langage de haut niveau suffit puisque c'est ensuite le compilateur qui va traduire dans le langage de la machine ce programme. Pour les performances du matériel, les architectures superscalaires disposent d'un matériel qui s'occupe seul de déployer le parallélisme. Le temps de développement d'une application est onc très réduit, puisqu'il se limite une fois le programme écrit, à quelques secondes de compilation. Pour les architectures DSP VLIW, c'est le compilateur qui se charge de déployer le parallélisme. La programmation est donc tout aussi aisée pour un DSP, à la nuance près qu'il est souvent nécessaire d'effectuer des optimisations manuelles pour obtenir de meilleures performances. En conséquence, cette approche implique un temps de développement plus long.

Les FPGA disposent d'outils pour leur programmation. Il est cependant plus long pour passer du programme en langage de haut niveau, à la phase de configuration du composant. Si la compilation du programme HDL se fait plutôt rapidement, ce n'est pas le cas de la phase de placement/ routage qui peut prendre parfois des heures sur les gros FPGA. Par conséquent, il est moins aisé de programmer un FPGA qu'un processeur en général.

Pour les architectures reconfigurables à grain épais, la plupart des approches de développement d'outils tendent à partir de langages de haut niveau type C, et donc à développer des compilateurs dans le même esprit que les processeurs parallèles[Const88]. L'expérience du VLIW laisse penser que cette méthodologie rencontrera de nombreuses difficultés de mise en œuvre. Dans la mesure où ces outils sont relativement peu aboutis, c'est souvent dans un langage de niveau assembleur qu'est réalisée la phase de portage de l'application. Ceci nécessite un certain nombre de pré-requis concernant la structure interne du composant (jeu d'instruction des opérateurs, richesse des connexions, hiérarchie mémoire etc.) et en plus, des temps de développement et de validation plus longs que pour une programmation en langage de haut niveau. Le développement d'une application peut donc nécessiter de nombreuses heures de travail, sans compter sur l'apprentissage préalable nécessaire.

3.2.4. Surface

Le grain fin apporte sans conteste une flexibilité supérieure dans l'utilisation que l'on peut faire du matériel. Mais cette flexibilité a un coût en mm² de silicium. En effet, implémenter des opérateurs arithmétiques par des LUT est sous-optimal en terme de surface. Autrement dit, pour réaliser la même fonction, l'utilisation de granularité épaisse sera *a priori* plus efficace, et donc moins coûteuse.

Pour les processeurs, l'utilisation de structures de contrôle complexes dans les processeurs superscalaires nécessite à l'évidence une surface plus importante que pour son homologue VLIW ; le déploiement du parallélisme étant laissé à la charge du compilateur.

3.3. Synthèse

Le tableau suivant résume les points que nous avons discutés dans cette section. Les éléments qui apparaissent reflètent une tendance générale dans un contexte ciblant essentiellement la mise en œuvre de systèmes embarqués pour des applications TSI.

Tab. 7. *Résumé du compromis de performances : ces résultats sont qualitatifs et permettent simplement d'estimer l'intérêt de chaque approche dans le domaine du TSI*

		Temps	*Consommation*	*Flexibilité*	*Surface*
Processeurs	Superscalaire SMT	☺☺	☹	☺☺☺	☹
	DSP VLIW	☺☺	☺	☺	☺
Architectures Reconfigurables	Grain Fin mixte	☺☺	☺	☺☺	☹
	Grain épais	☺☺☺	☺☺	☺	☺

Après une analyse de la bibliographie concernant les architectures de traitement flexibles, nous pensons que les architectures reconfigurables à grain épais sont une alternative intéressante aux autres architectures, en particulier en tant qu'accélérateurs. C'est à partir de ces premiers éléments qualitatifs, que nos travaux de recherche se sont orientés vers la conception d'une architecture répondant à ces critères. Le Systolic Ring, que nous présentons par la suite, servira de support pour les problématiques de caractérisation et de définition de méthode pour l'exploitation du parallélisme à travers la reconfiguration dynamique.

4. SUPPORT D'ETUDE ET DE VALIDATION : LE SYSTOLIC RING

Les Systolic Ring est une architecture reconfigurable dynamiquement destinée à décharger le processeur hôte d'un système des applications coûteuses en temps CPU. Elle dispose d'opérateurs à grains épais. Ce composant a été développé dans le cadre d'une thèse au LIRMM et validée sur un certain nombre d'applications de traitement du Signal [Beno2b] (DCT [Beno02a], FFT, transformée en ondelettes, estimation de mouvement, détection de contours …). Ces validations ont permis cependant de mettre en avant un certain de points faibles qu'il était nécessaire de corriger pour envisager des évolutions de cette architecture. Pour la version initiale du Systolic Ring, le lecteur pourra se référer à la thèse de Gilles Sassatelli [Sass02] qui présente dans le détail ce composant dans sa version initiale.

Cette architecture correspondant au profil des architectures qui nous semble être les mieux adaptées à la prise en charge d'applications consommant du temps CPU, nous l'utiliserons comme support d'étude et de validation dans la suite de ce mémoire. Pour cette raison, nous présentons ici cette architecture et notamment les modifications que nous avons apportées afin d'améliorer ses performances et sa flexibilité.

4.1. Présentation générale de l'architecture

L'architecture du Systolic Ring est basée sur un modèle mémoire de Harvard, et deux couches communiquant chacune avec une des deux mémoires. La couche de contrôle contient les registres de configuration et c'est elle qui est chargée d'aller lire le programme de configuration dans la mémoire programme, puis de décoder le motif de configuration à appliquer. La couche opérative est constituée d'opérateurs et de ressources d'interconnexion qui sont reliées à la mémoire de données dans laquelle se trouve les données à lire et les résultats à écrire après traitement.

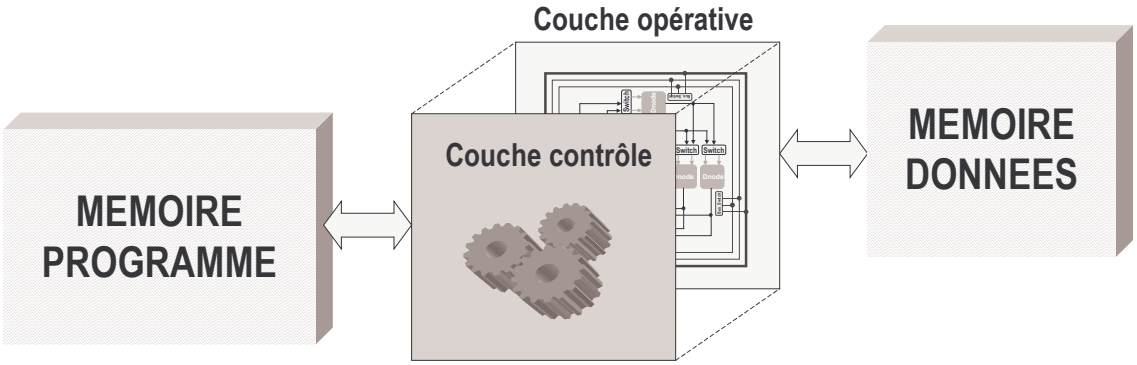


FIG. 41 - *Architecture conceptuelle du Systolic Ring*

4.1.1. Architecture des opérateurs

L'élément de base de l'architecture du Systolic Ring est un nœud de traitement utilisé pour implémenter les opérations arithmétiques des applications prises en charge. Chacun de ces éléments est identique et consiste en un macro-opérateur baptisé Dnode (Data Node). Il est constitué d'une base UAL – Multiplieur (16 bits) et les opérandes peuvent être des données issues de l'extérieur (autres Dnodes, mémoire), d'une banque de registres (16 registres) ou d'un bouclage de la sortie. Le choix des opérandes se fait à l'aide de micro-instructions qui permettent de router le signal à travers des multiplexeurs. Ces micro-instructions proviennent de la configuration du Dnode, qui est appliquée à partir de la couche

contrôle de l'architecture. Cette couche de contrôle est constituée de séquenceurs de configurations associés aux Dnodes. Ce séquenceur peut stocker jusqu'à huit micro-instructions et les exécuter en boucle.

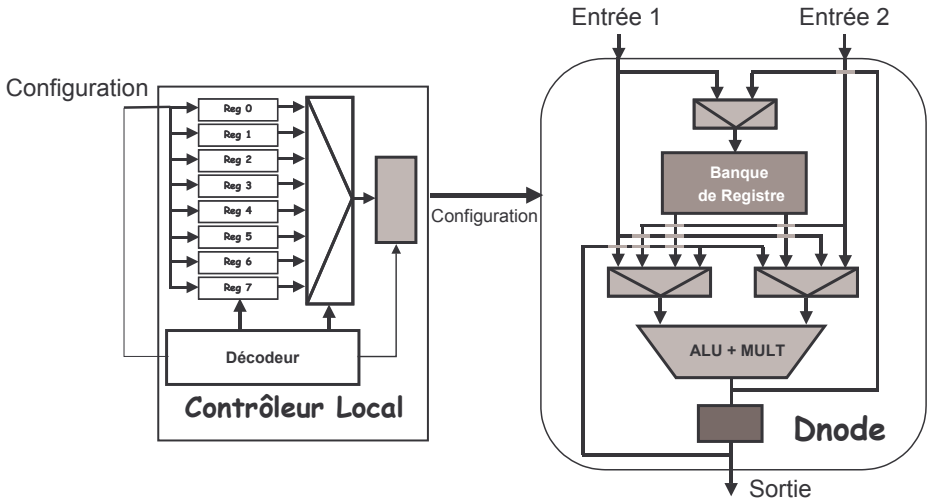


FIG. 42 - *Architecture du Dnode et de son contrôleur local*

4.1.2. Topologie et hiérarchie d'interconnexion

L'élément de connexion de base de l'architecture est un *switch* associé à chacun des Dnodes de l'architecture. Il permet de router vers chaque macro-opérateur, les signaux de chacun des Dnodes de la couche située en amont, les données issues des piles *FIFO* ou encore des réseaux de bus (que nous présenterons dans la suite). Un contrôleur local identique à celui des Dnodes est associé à chaque *switch* afin de *séquencer* les configurations de cet élément de routage.

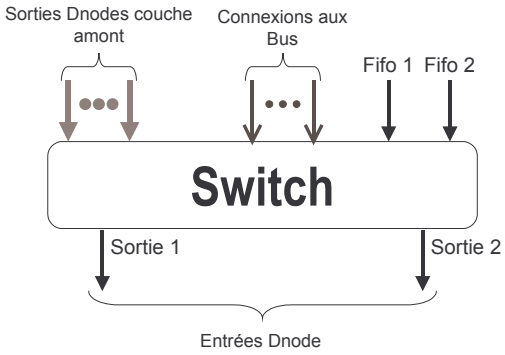


FIG. 43 - *Le switch, l'élément d'interconnexion*

L'organisation des Dnodes consiste en une organisation en couches d'opérateurs. Le nombre d'opérateurs par couche est un paramètre de l'utilisateur nommé N. Ces couches sont ensuite mises bout à bout et forment ainsi un premier chemin de données configurable (figure 44).

Ce premier niveau de connexion n'autorise des communications qu'entre couches adjacentes, et ce dans un seul sens (flot primaire). Or, il est parfois nécessaire de connecter des opérateurs de couches non-adjacentes soit pour contourner une couche d'opérateurs soit pour faire remonter des données à l'opposé du flot primaire. A l'origine, un réseau de rétro-propagation assurait ce rôle. Il a cependant été abandonné au profit d'une structure plus flexible et moins coûteuse en termes de ressources. Un réseau de bus configurable est utilisé à cet effet. Chaque Dnode de chaque couche peut se connecter à ce réseau par le biais d'un

switch configurable. La figure 45 illustre ce deuxième niveau d'interconnexion. Le nombre de bus est paramétrable (B).

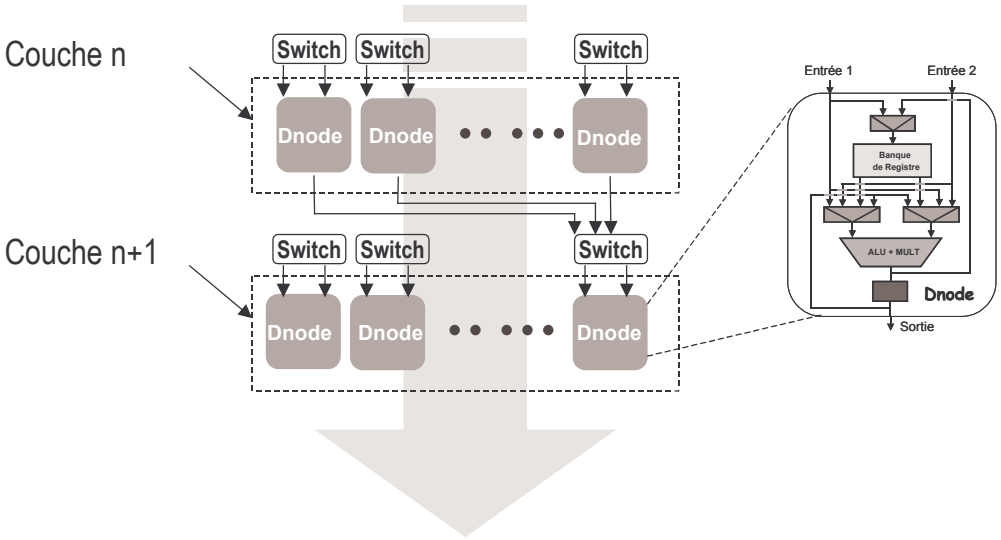


FIG. 44 - *Interconnexion des couches adjacentes de Dnodes et flot de données direct*

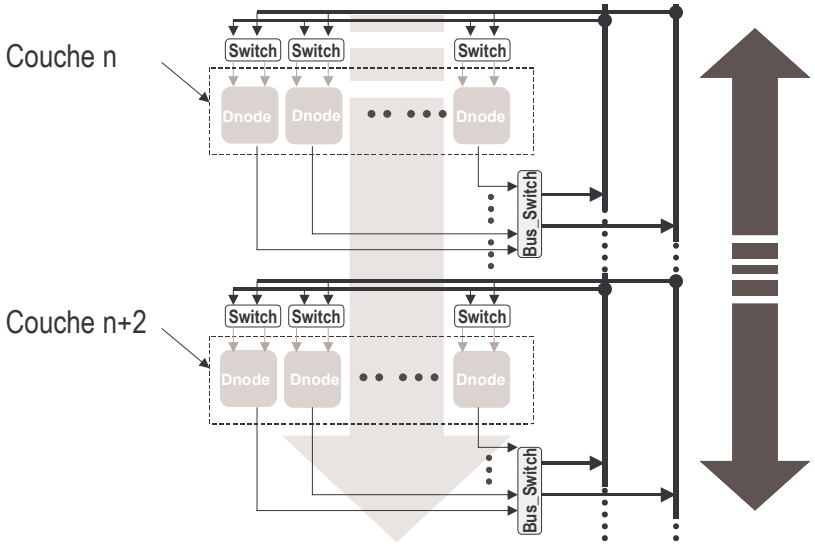


FIG. 45 - *Interconnexion des couches non-adjacentes et flot de données secondaire*

Les couches de Dnodes se succèdent dans l'architecture et la dernière est connectée à la première. Ainsi, le réseau forme un anneau, d'où le nom de Systolic Ring. Le nombre de couches est également paramétrable (C). La figure 46 illustre cet anneau pour des valeurs N=2, B=2, C=4.

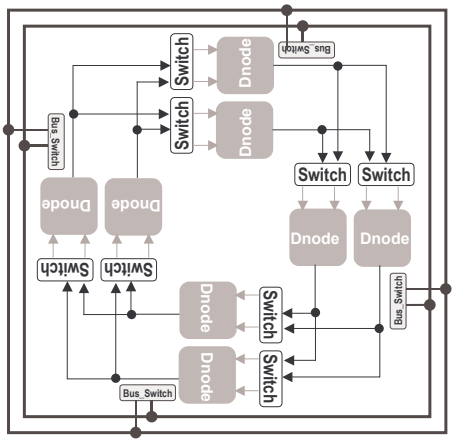


FIG. 46 - *Interconnexion en anneau des couches et des bus*

L'*interconnectivité* totale assurée par les switch pourrait s'avérer coûteuse en surface occupée si l'on augmente significativement le nombre de Dnodes par couche. Pour cela, nous introduisons un troisième niveau de connexion de hiérarchie basé sur un deuxième réseau de bus paramétrable, permettant de connecter plusieurs anneaux de processeurs. Le nombre d'anneaux utilisé est également un paramètre (S). La figure 47 représente une version à quatre anneaux de l'architecture.

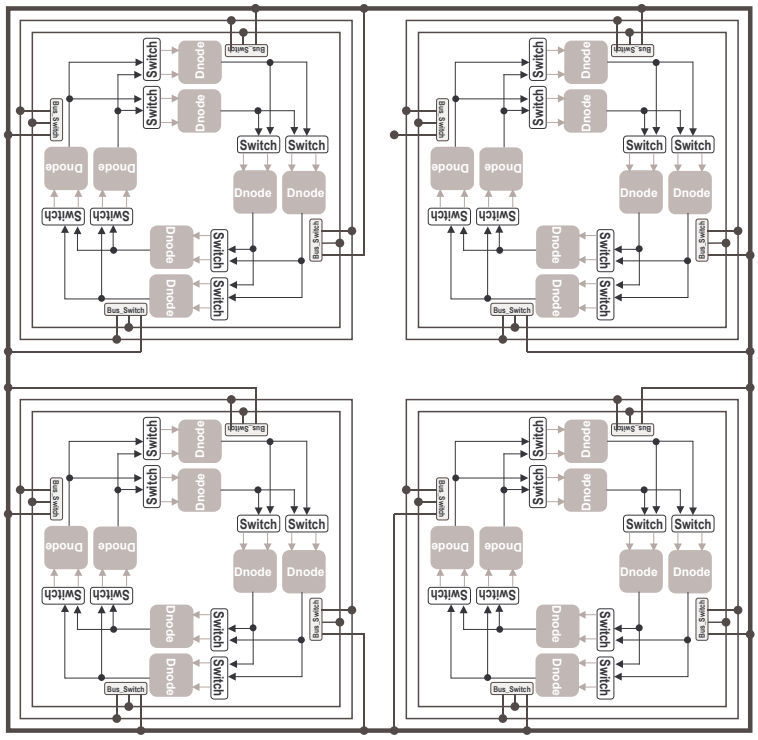


FIG. 47 - *Interconnexion des anneaux sur un deuxième réseau de bus*

4.2. Environnement de l'architecture

4.2.1. Hiérarchie mémoire

Comme nous l'avons évoqué au début de cette section, l'architecture mémoire est du type « Harvard ». Du côté de la mémoire programme, c'est un cœur de processeur RISC qui

assure la lecture des instructions de configuration dans la mémoire en calculant les adresses. Il stocke temporairement cette instruction qui est ensuite décodée par la partie contrôle du Systolic Ring. Chaque instruction de configuration adresse une couche d'opérateurs en mode normal ou bien tous les opérateurs en mode SIMD. Suivant la couche adressée, le contrôleur modifie le contenu des registres locaux de configuration. Sinon, la configuration reste inchangée.

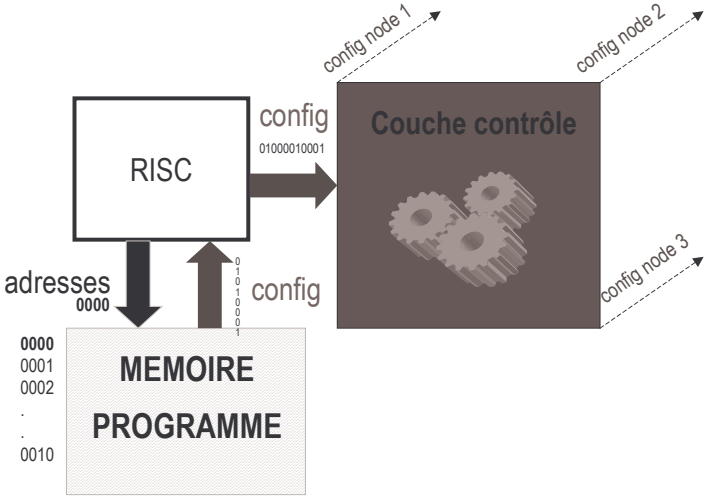


FIG. 48 - *Gestion des configurations par le processeur RISC*

Du côté de la mémoire de données, les motifs d'injection ont été, jusqu'à présent, généré de manière *ad hoc* dans les simulations, ou alors par le processeur Nios [Nios00] synthétisé dans le prototype FPGA (voir paragraphe suivant). Afin d'assurer une alimentation continue, une étude vise actuellement à proposer une solution dans cette optique (figure 49). Elle consiste à mettre en œuvre des blocs de piles FIFO connectées à un contrôleur qui assure un « remplissage » en adéquation avec le(s) motif(s) algorithmique(s) en cours de traitement.

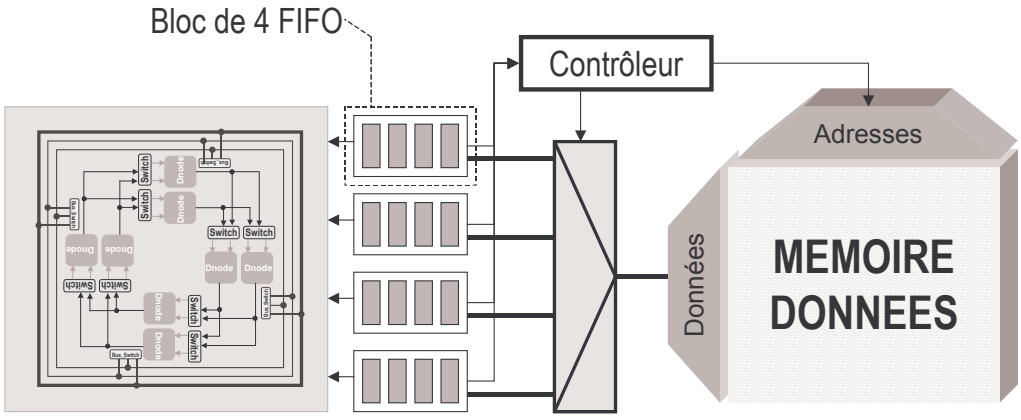


FIG. 49 - *Gestion de l'injection des données dans le Systolic Ring*

4.2.2. Outils

Cette architecture est proposée avec un ensemble d'outils. Tout d'abord, un outil de génération de code VHDL paramétrable permet de générer une version synthétisable du Systolic Ring pour n'importe quel jeu de paramètres. Ensuite, un outil de programmation de niveau « assembleur », également paramétrable, permet de micro-programmer n'importe quelle version du Systolic Ring. Le niveau d'abstraction autorisé permet de générer facilement un programme de configuration pour une application donnée, à partir du moment où l'utilisateur connaît l'architecture. Dans le cadre de l'équipe projet POMARD [Poma04], une étude vise à évaluer la faisabilité d'un compilateur C[Aho86] pour ce type d'architecture. Ces travaux sont basés sur l'outil Design Trotter [Lemo03], qui permet notamment de générer un graphe de flot de données hiérarchique (HCDFG) à partir d'une description en C de l'algorithme. L'idée consisterait alors à entrer le jeu d'instructions du Systolic Ring dans l'outil afin de générer, automatiquement, un fichier de configuration pour le coprocesseur.

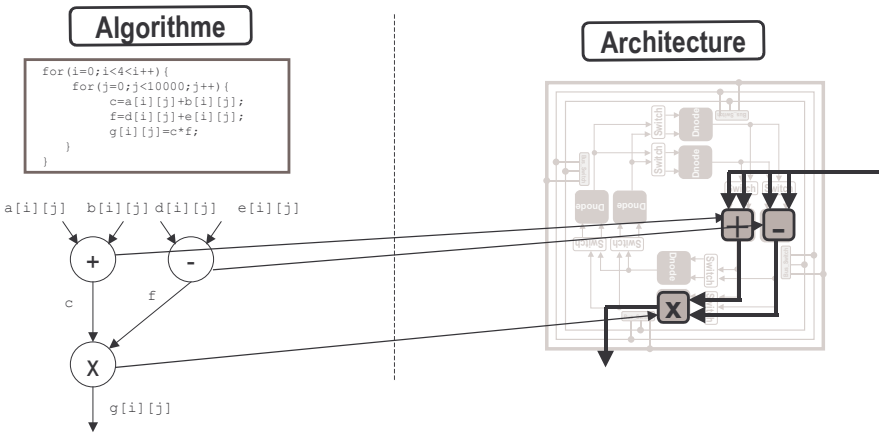


FIG. 50 - *Exemple d'adéquation algorithme / architecture. Une fois que les opérations et le chemin ont été identifiés, il ne reste plus qu'à entrer ces informations dans l'outil de génération de code paramétrable, qui se charge de créer le programme de configuration correspondant.*

4.2.3. Prototype

Cette architecture a été prototypée sur une plate-forme FPGA (Altera) avec un microprocesseur NIOS, des mémoires, un contrôleur VGA et un mécanisme pour alimenter l'architecture en données. La figure 51 illustre ce prototype. On peut noter qu'il y deux mémoires qui sont utilisées : une première stocke l'image initiale issue de l'*imager* CMOS par le biais du NIOS, et une deuxième reçoit les données traitées par le Systolic Ring (c'est un Dnode de ce dernier qui calcule les adresses d'écriture). Le deuxième port de la « Mémoire Image Traitée » est connecté au contrôleur VGA qui permet l'affichage de l'image traitée à l'écran. Le programme de configuration proposé dans cet exemple permet de calculer le gradient de l'image et donc de réaliser une détection de contours.

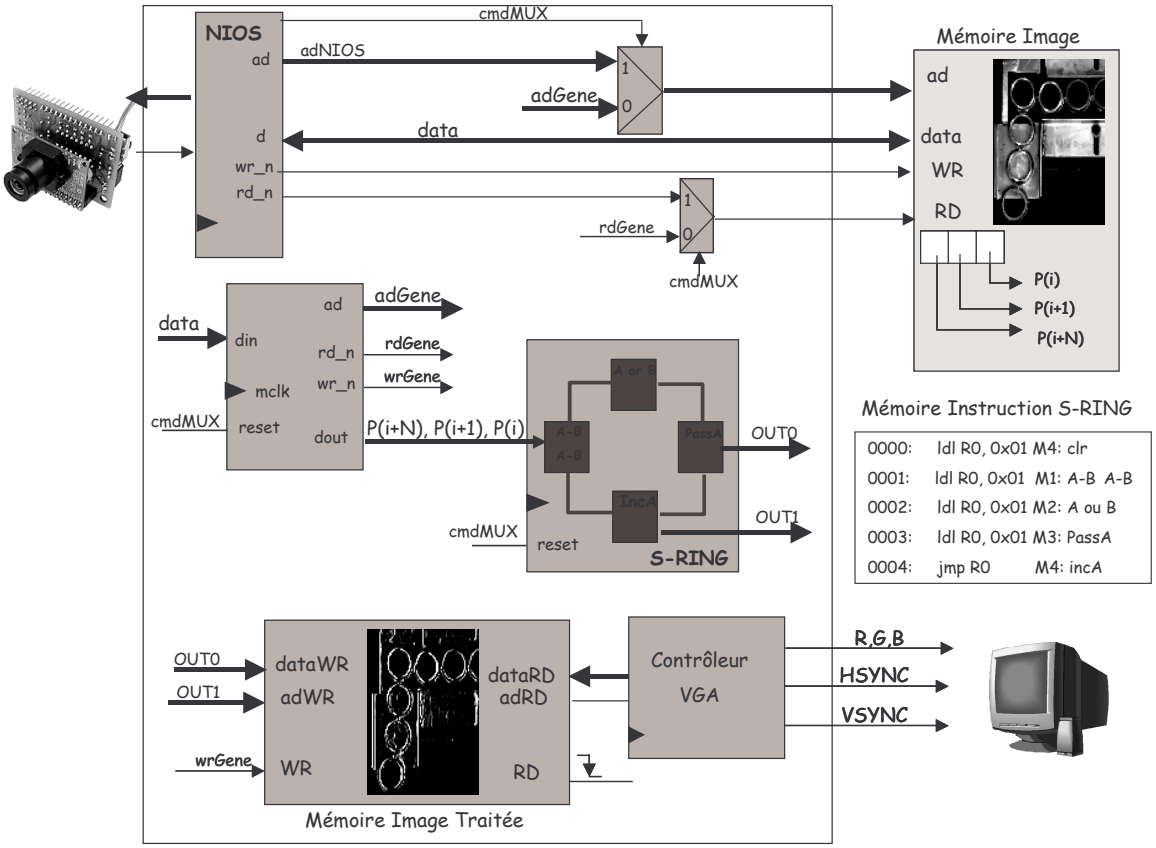


FIG. 51 - *Validation du Systolic Ring sur plate-forme FPGA.*

5. CONCLUSION

Nous avons réalisé un état de l'art des architectures de traitement flexibles dans un contexte SOC. La figure 52 résume notre état de l'art. Nous avons considéré deux familles d'architectures de traitement flexibles, les processeurs et les architectures reconfigurables. Pour chacune, nous avons insisté sur les caractéristiques architecturales qui permettent d'améliorer le compromis de performances, dans le domaine que nous nous sommes fixés, notamment les moyens matériels mis en œuvre pour exploiter le parallélisme, élément déterminant pour l'accélération des applications.

Après un premier bilan qualitatif, nous estimons que les architectures reconfigurables à grain épais disposent d'un compromis théorique très attractif pour les applications du TSI, notamment en tant que coprocesseur. La granularité des éléments de calcul des architectures reconfigurables à grain épais semble être en adéquation avec la majeure partie des applications ciblées. De plus, les aptitudes spatiales de ce type de composant permettraient a priori d'exploiter tous les grains de parallélisme possibles. De plus, leur flexibilité leur confère un avantage certain dans un domaine qui est continuellement en pleine évolution.

C'est pour ces raisons que des architectures telles que le Systolic Ring ont été proposées par la communauté scientifique. Ce type de composant très attractif n'en est encore qu'au stade d'élaboration. Nous avons d'ailleurs apporté un certain nombre de modifications nécessaires, et nous utiliserons ce composant comme cible technologique dans nos prochaines études et validations, dans la suite de ce mémoire.

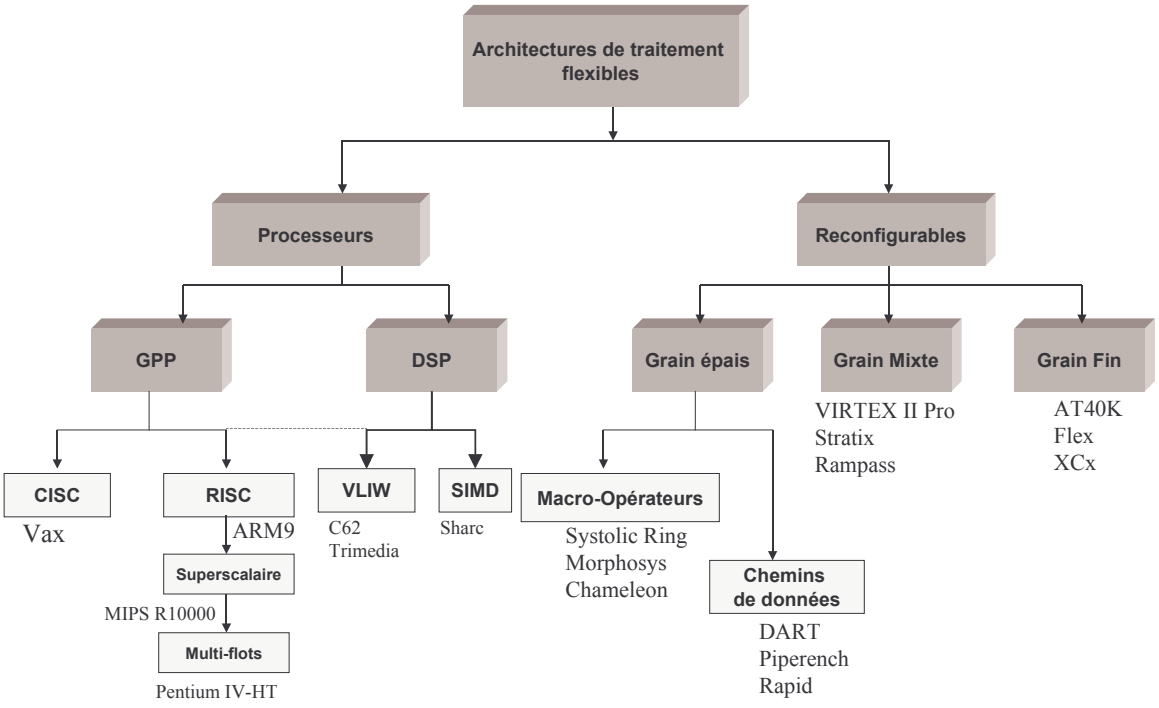


FIG. 52 - *Exemple de classification des architectures de traitement flexibles*

L'analyse comparative ne peut définitivement pas se limiter au bilan qualitatif de ce chapitre. C'est pour cette raison que nous poursuivrons ce mémoire par la définition d'un ensemble de métriques. Nous chercherons à répondre dans un premier temps, au problème de la caractérisation des accélérateurs de traitement flexibles. Les résultats obtenus nous permettront alors de comparer les différentes architectures existantes et de tirer des conclusions sur leurs performances potentielles. Ces métriques seront également utilisées

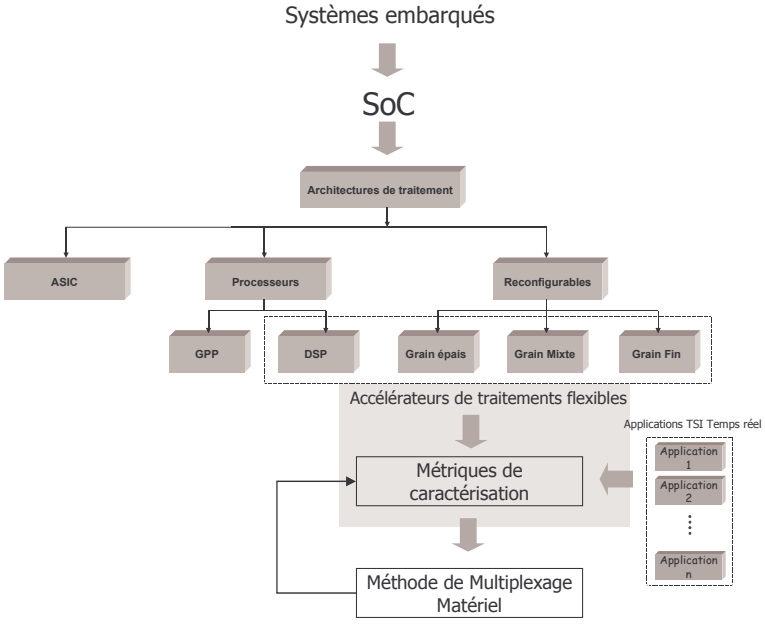
pour caractériser en fonction de l'application, l'architecture la plus adéquate. Dans le but de caractériser la *scalabilité* des architectures paramétrables tels que le Systolic Ring, nous proposerons également une méthode visant à étudier l'évolution des métriques en fonction de leurs paramètres.

Chapitre 3

METRIQUES POUR LA CARACTERISATION DES ACCELERATEURS

INTRODUCTION

De nombreux travaux de recherche tendent à démontrer les avantages des architectures reconfigurables à grain épais, qui semblent, *a priori*, disposer d'un potentiel en adéquation avec les spécifications imposées par les applications TSI temps réel pour SOC. Nous nous proposons, à travers ce chapitre, de comparer les caractéristiques des différentes familles d'architectures de traitement flexibles. Nous utiliserons pour cela un modèle d'architecture et un modèle de tâche pour définir un ensemble de métriques intrinsèques, afin d'intervenir très tôt dans le flot de conception. Nous exposerons les résultats obtenus de la comparaison des différentes architectures considérées, de la répartition des tâches, et enfin de la caractérisation de modèles d'architectures paramétrables.



SOMMAIRE DU CHAPITRE

1. ALGORITHME, ARCHITECTURE ET ADEQUATION

Pour commencer ce chapitre, nous présenterons les modèles et les méthodes que nous utiliserons par la suite. Il s'agira dans un premier temps de définir un modèle de tâche, relatif au domaine d'applications que nous avons fixé, à savoir le TSI. Ensuite, nous proposerons une architecture générique permettant la modélisation des accélérateurs de traitement flexible. Pour finir, nous discuterons les méthodes d'adéquation possibles et nous ferons à cette occasion, quelques hypothèses importantes concernant la caractérisation et la comparaison d'architectures.

1.1. Modèle de tâche

Par modèle de tâche, nous entendons les propriétés algorithmiques des applications ciblées dans notre contexte d'étude. Pour bien fixer le contour de ce travail, nous précisons d'abord la terminologie utilisée, puis nous exposons les dites caractéristiques.

1.1.1. Algorithme – Langage – Programme

La flexibilité est un critère important dans le cadre de notre étude. Celle-ci se situe dans la possibilité de modifier la fonctionnalité d'une architecture, après qu'elle ait été conçue. C'est généralement par un programme écrit par le concepteur d'une nouvelle application que s'opère cette « reconfiguration ».

Un programme est une traduction d'un algorithme dans un langage informatique donné. Il existe une large variété de langages de programmation (*e.g.* Pascal, C, ADA, Occam, VHDL, verilog…) qui, suivant la puissance sémantique du langage, permettent d'expliciter plus ou moins simplement le parallélisme inhérent aux applications à implanter. Suivant l'architecture ciblée et ses aptitudes au parallélisme, le choix du langage dépend de l'existence d'un compilateur en tout premier lieu, et si ce n'est pas le cas, de son adéquation avec le langage. En effet, le compilateur devra alors combler le fossé entre la description de l'application et l'architecture ciblée, ce qui signifie donc un développement plus complexe.

Le langage C est très largement répandu dans la communauté des programmeurs, de part ses possibilités et sa simplicité. C'est une des raisons majeures qui fait que les concepteurs de nouvelles architectures essaient de fournir des compilateurs C, afin que leur composant soit utilisé par une majorité de personnes. Pour notre modèle de calcul, nous avons donc supposé des descriptions d'algorithmes du TSI en langage C. Nous faisons également l'hypothèse que le niveau de couplage entre le processeur et le co-processeur leur permet d'exécuter des sous-parties de programmes de manière indépendante.

Les questions auxquelles nous allons tenté de répondre dans un premier temps sont les suivantes : quelles sont les caractéristiques de code que l'on peut significativement accélérer ? Autrement dit, quel est le profil des parties critiques d'un algorithme ? Il s'agit donc d'identifier les parties d'un programme consommant du temps CPU.

1.1.2. Propriétés du modèle

```
//Directives du pré-processeur
//Déclaration des constantes
main() {
        //Déclaration des variables
        …
        //bloc_instructions
        if(condition){
                //bloc_instructions_condition_vraie
        }
        else{
        //bloc_instructions_condition_fausse
        }
        switch (variable){
                case valeur1 :
                        //bloc_instructions_si_valeur1
                        break ;
                …
                default :
                        //bloc_instructions_par_défaut
        }
        for(init_compteur ;fin_compteur;in(de)crement_compteur){
                //bloc_instruction_boucle
        }
        while(condition_boucle){
                //bloc_instruction_while
                //modification_condition_boucle
        }
}
```

FIG. 53 - *Exemple de programme en C. Au niveau des données, on distingue les constantes et les variables. Les constantes correspondent à des valeurs fixes de données codées dans les instructions programme, les variables correspondent à des emplacements mémoires réservés statiquement ou dynamiquement, et modifiés durant l'exécution. Le programme principal est quant à lui composé d'appels de fonctions et d'instructions du C. L'exécution de certaines d'entre-elles dépendent du résultat d'un test sur une variable (if, while, switch case), d'autres sont déterministes (for).*

Nous cherchons à identifier les caractéristiques des portions de programmes qu'il est envisageable d'accélérer dans le cadre d'applications TSI. Il s'agit donc de localiser les parties critiques, *i.e.* celles qui consomment du temps CPU. Dans un système monoprocesseur, il est admis que le CPU passe la majeure partie de son temps sur des calculs dits *réguliers*. Si l'on analyse la structure d'un algorithme décrit en C (figure 53), on distingue trois types de blocs de calcul : ceux pour lesquels le traitement dépend d'un test (*if, switch*), les traitements répétitifs (*for*), et les traitements répétitifs conditionnels (*while, do while*). Ce sont les traitements répétitifs (et conditionnels parfois) qui consomment du temps CPU. Les résultats de [Codi02] sur les bancs de test *SPECfp95* et *Perfect Club* confirment cette tendance puisque 78 à 95% du temps d'exécution est passé sur des boucles. Cela est d'autant plus vrai dans un contexte TSI, où il est souvent nécessaire d'appliquer les mêmes opérations à un flot de données (*e.g.* traitement vidéo). Nous limiterons donc le contour de notre étude à des portions de code fortement répétitives telles que les boucles, correspondant à des traitements à dominante arithmétique sur des données de largeur multiple de l'octet. En résumé, les parties de programme prises en charge par l'accélérateur auront les propriétés suivantes:

- Les variables sont de type *mot* (entiers, ou réels), codées sur *k* octets. Dans la mesure où les architectures d'accélérateurs que nous étudions fonctionnent pour la grande majorité sur des données en virgule fixe, nous faisons donc l'hypothèse que les opérations supplémentaires de recadrage sont identiques pour chaque architecture.

- Pas de d'instructions conditionnelles. Les blocs correspondant représentant une charge CPU moins élevée, sont supposés s'exécuter sur ce dernier.
- Un cœur de boucle constitué de N opérations. On considère uniquement les opérations arithmétiques et logiques de base (+, -, *, &, ||, etc., pas de combinaisons d'opérations tels que « multiplication-accumulation »).
- Un nombre d'itérations (déterminé ou prédit) de N_{ITER}, proportionnel au volume de données à traiter.

Afin de simplifier le problème, nous considérons le produit $N.N_{ITER}$ comme le nombre total d'opérations *parallèles* à réaliser, ce qui revient à négliger les dépendances temporelles entre les différentes itérations de boucles. Cette hypothèse est certes peu vraisemblable, mais dans la mesure où DSP et architectures reconfigurables à grain épais sont aptes au parallélisme temporel, on assimile ce dernier au parallélisme spatial, compte tenu du niveau d'abstraction.

1.2. Modèle d'architecture

L'architecture de l'accélérateur peut être considérée avec une description plus ou moins fine. Compte-tenu du niveau d'abstraction et dans un souci de simplicité, nous nous limiterons à la structure interne du coprocesseur. Nous faisons l'hypothèse que les entrées/sorties et les mémoires offrent des débits permettant d'exploiter, sans contrainte, la totalité des ressources de calcul.

1.2.1. Description et propriétés du modèle

Le modèle général que nous avons choisi est illustré par la figure 54. Ce modèle est constitué de 4 éléments :

- **Unités de traitement (OP) :** Ces éléments constituent les briques de base de l'architecture, car c'est par eux que vont être réalisées les opérations arithmétiques et logiques de l'application implantée.
- **Ressources d'interconnexion :** ces ressources assurent les communications entre unités de traitement. Elles peuvent être de types différents : liaisons directes, *crossbars*, *switch*, bus. Pour les architectures séquentielles, on ne peut pas vraiment parler d'interconnexion, mais plutôt de stockage temporaire entre les partitions temporelles (bancs de registres). L'arrangement des opérateurs est variable d'une architecture à une autre : on peut distinguer les topologies linéaires (1D), topologies en grille (2D), topologies 3D…, ainsi que différents niveaux de hiérarchie.
- **Mémoire de configuration :** celle-ci est constituée de registres de mémorisation permettant de stocker les instructions (processeurs) ou configuration(s) (architectures reconfigurables) des unités de traitement et des ressources d'interconnexions.
- **Contrôleur (optionnel) :** quand celui-ci est disponible, il permet de reconfigurer certaines parties de l'architecture durant la phase de traitement (unités de traitement et éventuellement ressources d'interconnexions)
- **Hypothèse Mémoire et Entrées/Sorties :** on suppose qu'il n'y pas de cycles de suspension (tels que des échecs de cache par exemple) durant la phase de traitement d'une application, soit une alimentation continue de l'accélérateur en données d'application.

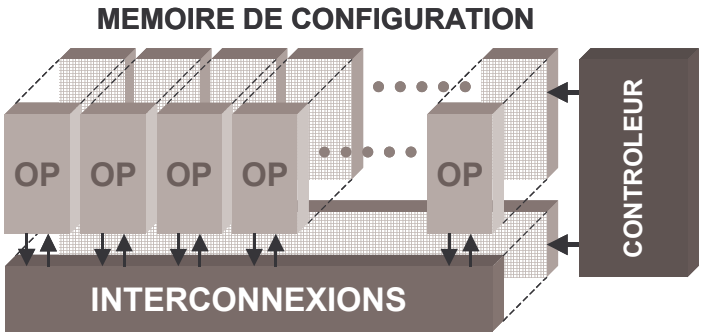


FIG. 54 - *Modèle général pour architectures de traitement flexibles*

1.2.2. Exemples

Le modèle d'architecture présenté ici permet de modéliser des architectures d'accélérateurs flexibles de traitement. Les figures 55, 56 et 57 donnent un exemple de trois types d'architecture.

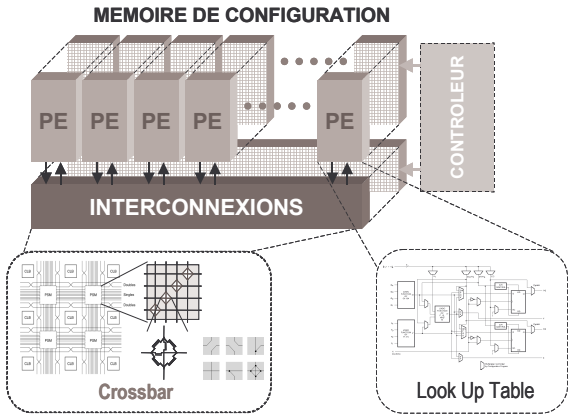


FIG. 55 - *Exemple de FPGA Xilinx [Xili00] représenté par ce modèle*

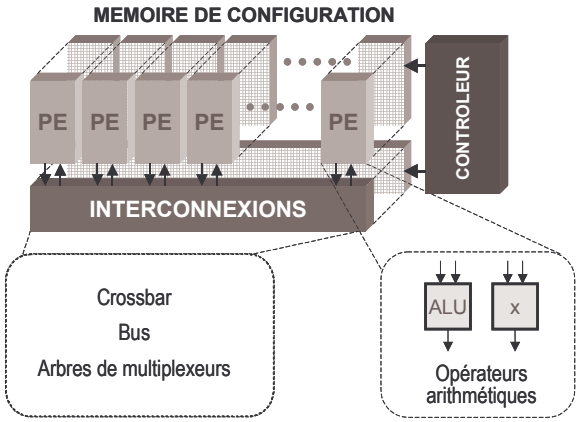


FIG. 56 - *Modèle pour Architecture Reconfigurable à grain épais*

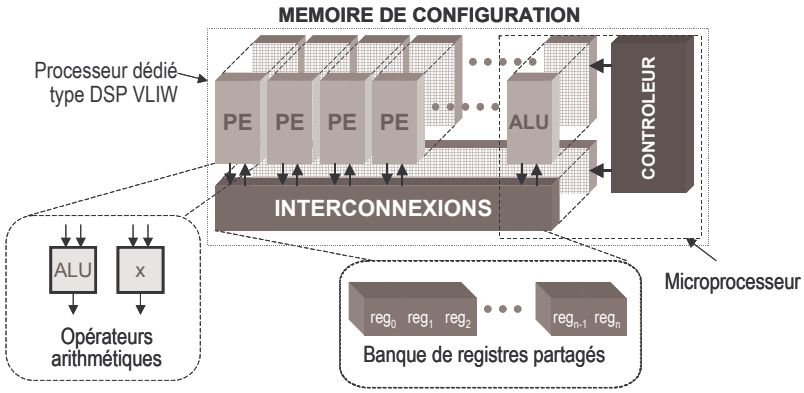


FIG. 57 - *Modèle général appliqué aux microprocesseurs DSP VLIW*

1.3. Méthodes d'adéquation

Lorsqu'on cherche à comparer plusieurs implantations d'une même application, la méthode d'adéquation peut être décisive dans les performances obtenues : dans [Henn03], les auteurs exposent des temps de traitement variant d'un facteur 10 entre une adéquation automatique (compilateur) et une adéquation manuelle (assembleur) pour un DSP VLIW. Dans ce paragraphe, nous recensons les différentes approches possibles en évaluant leur coût et leur efficacité.

Pour les architectures reconfigurables à grain épais, les outils automatiques n'en sont qu'au stade expérimental. Nous pouvons néanmoins citer [Mori98] et [Nade01] qui proposent des outils (quasi-)automatiques d'aide à la génération de configurations pour une application donnée. Au niveau des compilateurs C, le lecteur intéressé pourra se référer à [Cron98] et [Cala00]. Le plus souvent, la majeure partie de la phase de projection est cependant réalisée à la main. Cette approche est plus longue à mettre en œuvre et elle est source d'erreurs. Par contre, avec une bonne connaissance de l'architecture, les configurations générées sont généralement plus performantes que celles obtenues de manière automatique.

Dans la suite de cet article, nous désirons comparer le potentiel intrinsèque de chaque accélérateur. Nous supposons donc, quelle que soit la méthode d'adéquation, que la projection permet d'exploiter complètement le parallélisme de la description de l'algorithme. Autrement dit, d'après le modèle de calcul présenté précédemment, nous faisons l'hypothèse que chaque accélérateur est capable d'exploiter autant de parallélisme que ses ressources matérielles le permettent.

1.3.1. Projection automatique

Dans ce cas, le programmeur utilise un outil qui réalise automatiquement l'allocation des ressources matérielles. Suivant s'il s'agit d'un processeur ou bien d'un FPGA, les langages d'entrée diffèrent ainsi que les procédés de projection.

a. Compilateur

La compilation consiste en une traduction d'un programme en langage de haut niveau dans le langage de la machine cible, le niveau assembleur étant la partie lisible de cette traduction. Les performances des compilateurs diffèrent suivant les ressources matérielles mises en œuvre pour adapter l'ordonnancement statique réalisé lors de la phase de compilation. En effet, les processeurs superscalaires actuels permettent de réordonnancer dynamiquement plusieurs instructions afin de pouvoir exécuter en parallèle plusieurs opérations non dépendantes. Pour l'approche VLIW, le procédé de compilation est entièrement statique, *i.e.* qu'une fois compilées, les instructions sont appliquées dans l'ordre

défini lors de la compilation. Pour cela, différentes techniques de pipeline logiciel peuvent être utilisées (la plupart de ces techniques sont comparées [Codi02]) pour déployer automatiquement un maximum de parallélisme. L'optimisation manuelle du code généré par le compilateur permet d'augmenter nettement les performances comme nous l'avons évoqué dans l'introduction plus haut. En conclusion, le principal intérêt de l'automatisation du procédé de projection est la simplicité d'utilisation car elle ne permet malheureusement pas encore d'exploiter pleinement le potentiel de ces architectures.

b. Placement / Routage

Pour les FPGA, plutôt que de compilateur on parle d'outils de synthèse et de placement/routage. Dans ce cas, après la synthèse logique de la description matérielle, c'est l'outil qui génère une configuration matérielle en réalisant la projection de la description sur le FPGA ciblé. Ce processus peut être entièrement automatique ou introduire certains niveaux d'optimisations manuelles pour améliorer des solutions parfois insuffisantes.

Il existe d'autres approches où un langage de nature séquentielle est utilisé : c'est le cas du Handel-C [Celo04]. Il s'agit d'un langage de programmation conçu pour compiler des programmes en images matérielles pour FPGA (et ASIC également).

1.3.2. Projection Manuelle

Les utilisateurs d'outils automatiques ont souvent recours à des optimisations manuelles pour améliorer la solution initiale. Mais il est parfois nécessaire de réaliser entièrement le processus de projection à la main, lorsque aucune automatisation du processus n'a été proposée. Dans ce cas, le programmeur manipule un langage machine, de niveau assembleur. Cette approche est nécessairement plus longue à mettre en œuvre et source d'éventuelles d'erreurs. Par contre, avec une bonne maîtrise de l'architecture, les performances atteintes par ce biais, sont généralement satisfaisantes.

1.3.3. Hypothèse

Dans cette section, nous avons délimité le contour de notre étude aux accélérateurs de traitement dans le cadre d'applications TSI, en faisant un certain nombre d'hypothèses afin de comparer simplement et rapidement différentes approches existantes au plus tôt dans le flot de conception. Les architectures étudiées ici sont utilisées dans le cadre de la minimisation du nombre de cycles de calculs sur des boucles de tailles variables, afin de mettre en évidence leurs capacités accélératrices. De plus, nous supposons que pour chaque candidat, l'outil de projection de l'application (logiciel ou manuel) n'est pas limitatif dans l'exploitation des ressources. Cependant, par rapport au contexte des systèmes sur puce, la puissance de traitement ne peut être le seul critère de comparaison. Pour cela, nous chercherons à définir un ensemble de métriques allant dans le sens de l'évaluation d'un compromis de performances.

2. METRIQUES DE CARACTERISATION

Des trois axes de l'AAA dérivent trois axes de caractérisation, comme cela est illustré dans la figure 58. Un certain nombre de travaux dans ces domaines peuvent être référencés, notamment pour ce qui concerne la caractérisation des applications (métriques logicielles) ou la caractérisation de l'implémentation (métriques extrinsèques, *i.e.* post adéquation). Nous résumerons dans ce paragraphe les différentes métriques existantes et nous présenterons ensuite, plus dans le détail, nos outils et notre méthode de caractérisation intrinsèque des architectures de coprocesseur.

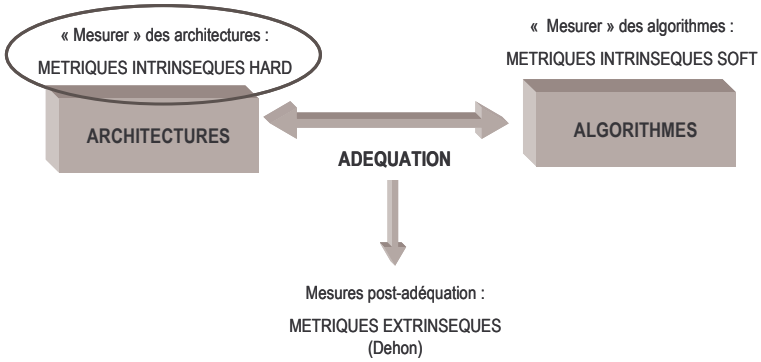


FIG. 58 - *Adéquation Algorithme Architecture et caractérisation. Suivant le niveau auquel on se place, il faut nécessairement utiliser des métriques spécifiques. Notre approche consiste à considérer les architectures de traitement flexibles et un modèle de calcul pour le TSI. Il est nécessaire, pour appréhender le problème de caractérisation dans sa globalité, d'avoir en tête chacun de ces aspects.*

2.1. Caractérisation logicielle

Les métriques logicielles visent à caractériser l'application à implanter. Dans la littérature, on trouve par exemple l'outil *MADEO* [Laga00] à partir duquel il est possible de générer une architecture reconfigurable disposant d'opérateurs les mieux adaptés à l'application. Un autre exemple est l'outil *Xplorer* qui permet de générer une configuration optimisée pour l'architecture à grain épais *Kress Array* [Hart00]. Afin de mieux illustrer l'utilisation de telles métriques, nous avons choisi d'exposer les travaux de [Lemo03].

2.1.1. Contexte d'utilisation

Le contexte d'utilisation de ces métriques est l'outil d'exploration architecturale, *Design Trotter*. Le point d'entrée de ce logiciel est une description en C de l'algorithme à implanter. A partir de celle-ci, le programme est traduit en HCDFG, c'est-à-dire en graphe de flot de données et de contrôle hiérarchique. Les figures 59 et 60 représentent le passage d'une description C en HCDFG et le flot de caractérisation logicielle automatisée. A noter également que cette description de l'application permet de jouer ensuite sur les compromis matériels nécessaires à l'implémentation de ces programmes.

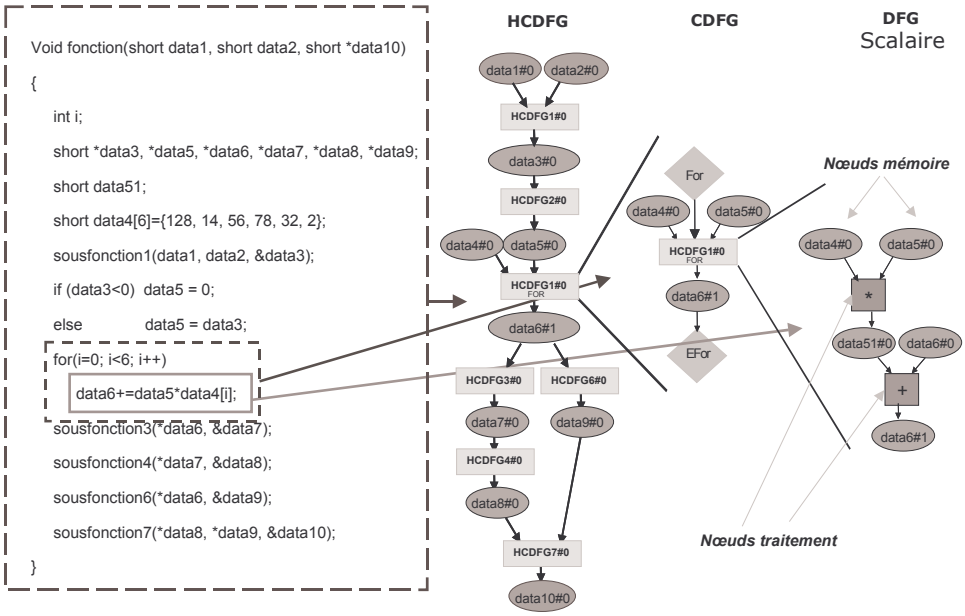


FIG. 59 - *Représentation sous forme de HCDFG de la description en C d'un algorithme. Comme nous pouvons le constater, les différents niveaux hiérarchiques du programme sont encapsulés : des graphes de flot de données (DFG) sont composés de nœuds de calcul et de nœuds mémoire, les CDFG encapsulent le DFG et des nœuds de contrôle sont ajoutés, idem pour le HCDG qui conserve la hiérarchie des CDFG.*

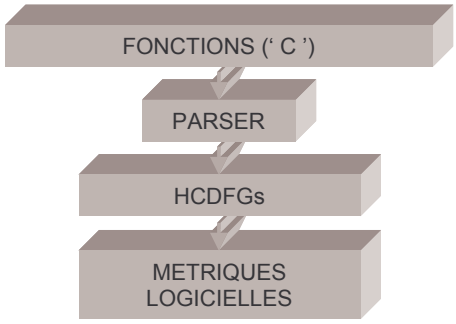


FIG. 60 - *Flot de caractérisation automatique. En partant d'une description d'un algorithme en langage C, le code est ensuite analysé puis un HCDFG est construit. Ensuite, un certain nombre de métriques logicielles sont calculées à partir des informations du graphe*

2.1.2. Métriques logicielles

L'ensemble des métriques présentées dans le cadre de la caractérisation logicielle dans [Lemo03] permet d'analyser l'orientation d'un algorithme décrit en langage C. On trouve, entre autres, trois outils dédiés à la quantification du parallélisme moyen, de l'orientation mémoire et de l'orientation contrôle.

a. Parallélisme

La métrique γ permet de calculer le parallélisme moyen d'une application donnée. N_{OP} correspond au nombre de cycles de calcul et C_P représente le chemin critique. Plus la valeur de cette métrique est élevée, plus le parallélisme est élevé et donc plus il est possible d'optimiser. La figure 61 illustre un exemple de calcul de γ pour un DFG simple.

$$\text{Métrique de parallélisme : } \gamma = \frac{N_{op}}{Cp}$$

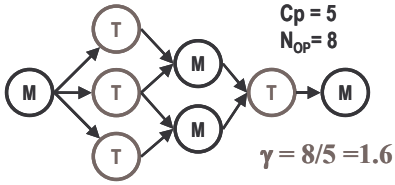


FIG. 61 - *Calcul du γ. Le N_{OP} est ici de 8 (8 nœuds de lecture mémoire ou de calcul) et le chemin critique, quelle-que soit la branche empruntée est de 5. Ceci donne un parallélisme moyen de 1,6.*

b. MOM

La métrique MOM (Memory Oriented Metric) permet de caractériser la fréquence des accès à la mémoire par un algorithme donné. La valeur du MOM est normalisée (comprise entre 0 et 1) et un MOM proche de 1 correspond à une dominante accès mémoire de l'application considérée. La figure 62 donne un exemple de MOM.

$$\text{Métrique orientée Mémoire : } MOM = \frac{Nm}{Nm+Np}$$

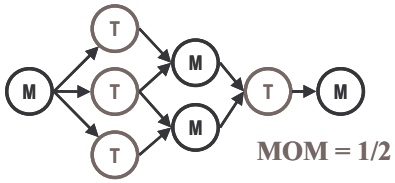


FIG. 62 - *Exemple de calcul du MOM. Le nombre de nœuds mémoire est de 4, tout comme le nombre de nœuds de traitement ce qui un MOM de 0,5.*

c. COM

Cette dernière métrique permet de caractériser les aspects contrôle d'un algorithme. Ainsi, le COM représente le nombre de nœuds de contrôle sur le nombre de nœuds total, sur un intervalle normalisé [0,1] où une valeur proche de '1' correspond à une application dominée par le contrôle. La figure 63 donne un exemple de COM.

$$\text{Métrique orientée Contrôle : } COM = \frac{Nc}{Nc+Np+Nm}$$

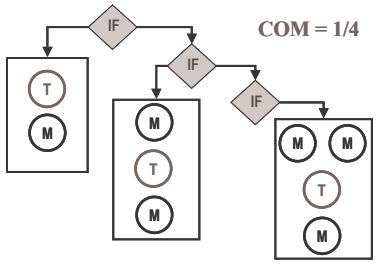


FIG. 63 - *Exemple de calcul du COM. Le nombre de nœuds de contrôle est de 3, et le nombre de nœuds total (contrôle, mémoire et calcul) est de 12 ce qui donne une valeur du COM à 0,25.*

2.1.3. Résultats

Ces métriques sont intégrées à l'outil Design Trotter. En entrée, un algorithme décrit en langage C est converti en HCDFG. C'est à partir de ce graphe que les différentes métriques présentées dans le paragraphe précédent sont calculées. La figure 64 illustre les résultats sur différentes applications TSI.

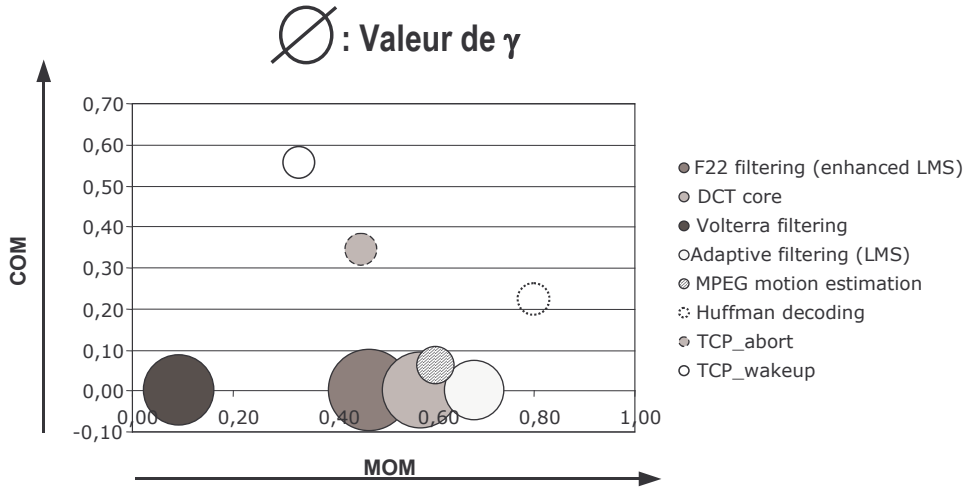


FIG. 64 - *Ce graphique représente les résultats du calcul des métriques logicielles sur différents algorithmes. Les deux axes permettent de représenter en abscisse le MOM, en ordonnée le COM et enfin, le parallélisme moyen st proportionnel au rayon des disques.*

Ces travaux de caractérisation sont très intéressants car ils permettent réellement de caractériser un programme. Cependant, les résultats sont à manipuler avec précaution, comme à chaque fois d'ailleurs qu'il s'agit de métriques. En effet, comme nous l'avons précisé au début de ce chapitre, un programme n'est qu'une description de l'algorithme dans un langage donné. Autrement dit, suivant le niveau du programmeur, les résultats peuvent varier. De plus, la sémantique séquentielle du langage C implique une « manière de pensée » sérielle : avec une description de l'algorithme dans un autre langage tel que le VHDL, quelles seraient les valeurs de ces métriques ? Enfin, concernant le parallélisme moyen, ne sont considérés ici que les opérations. Pour plus de représentativité, il faudrait prendre en compte également le parallélisme de boucle (déroulage) et le parallélisme de flot inter-fonctions. En conclusion, ces métriques donnent une indication fiable sur le parallélisme d'instruction, l'orientation mémoire et contrôle, uniquement de la description en C choisie.

2.2. Caractérisation extrinsèque

Pour comparer deux systèmes, il est possible aussi de se baser sur la caractérisation extrinsèque. Dans ce cas, il va s'agir de mesurer des caractéristiques telles que le temps, la consommation, la surface, sur une ou plusieurs applications implantées sur une architecture donnée, puis de comparer les résultats afin d'élire le système le plus adéquat. Nous avons choisi de présenter ici plusieurs métriques de caractérisation extrinsèque qui sont le temps d'exécution moyen, la loi de Amdahl et l'efficacité.

2.2.1. Temps d'exécution moyen

Le temps d'exécution total d'un processus correspond au temps qui s'écoule entre le lancement du programme et la fin du traitement. Pour un système donné, il est nécessaire de considérer l'ensemble des processus qui s'y exécutent, ainsi que la moyenne des temps d'exécution. La question à se poser est donc : comment comparer plusieurs systèmes ? Plusieurs solutions sont proposées dans la littérature. En effet, comparer des systèmes pour chaque programme est un procédé source de nombreuses confusions, et peut masquer certaines réalités. Pour éviter cela, il est possible de réaliser des moyennes de temps d'exécution.

Il peut s'agir tout d'abord de moyenne arithmétique pour *n* programmes :

$$\frac{1}{n}\sum_{i=1}^{n} T_i$$

Cette moyenne permet de pondérer de manière équivalente les résultats obtenus pour chaque programme. Cependant, chacun d'entre eux n'intervient pas de la même manière dans la charge de travail du système. En effet, certains processus sont exécutés beaucoup plus souvent que d'autres. Par conséquent, il peut apparaître comme nécessaire de multiplier chaque temps d'exécution par le poids de la charge (valeur comprise entre 0 et 1) qu'il représente, de la manière suivante :

$$\sum_{i=1}^{n} P_i \times T_i \text{ où } \sum_{i=1}^{n} P_i = 1$$

Ainsi, la moyenne totale pondérée représente le temps moyen passé par le système pour chaque processus ramené à la charge de travail relative.

2.2.2. La Loi de Amdahl

a. Principe

Cette loi s'intéresse à la performance globale du système [Henn03]. La philosophie de cette loi repose sur le fait qu'il faille fournir un effort d'accélération au cas fréquent. Cette loi indique en effet que le gain obtenu à partir de l'utilisation d'un mode d'exécution plus rapide est limité par la fraction de temps pendant laquelle ce mode est utilisé. Elle définit un critère d'accélération globale pouvant être obtenue par l'adjonction d'un co-processeur par exemple. Elle se calcule de la manière suivante :

$$A_{globale} = \frac{t_{old}}{t_{new}}$$

où t_{old} correspond au temps d'exécution avec seulement le processeur central, et t_{new} qui correspond au temps d'exécution en ajoutant le coprocesseur au système.

La loi de Amdahl dépend de deux facteurs qui sont :

- F : la fraction du temps de calcul du système originel qui peut-être utilisée pour tirer partie de l'amélioration (cette valeur est comprise entre 0 et 1).
- G : le gain apporté par le coprocesseur, c'est-à-dire l'accélération sur cette seule partie du programme.

Ainsi, on peut exprimer le temps d'exécution nouveau, t_{new} :

$$t_{new} = t_{old} \times [(1-F) + \frac{F}{G}]$$

d'où une accélération globale qui sera calculée de la manière suivante :

$$A_{globale} = \frac{1}{(1-F) + \frac{F}{G}}$$

b. Exemple

Supposons qu'on ajoute un coprocesseur graphique à un système. Celui-ci permet d'améliorer les performances sur les applications graphiques d'un facteur 20 par rapport au

CPU. Le CPU originel passait 30% de son temps sur ces applications. Au final, l'accélération est donc :

$$A_{globale} = \frac{1}{(1-0,3)+\frac{0,3}{20}} = 1,4$$

Cette loi permet non seulement de quantifier l'impact global de l'accélérateur sur les performances mais peut également servir de critère pour jauger le compromis coût / performances.

2.2.3. Efficacité d'un système

Certains travaux [Deho98][Wirth98] ont proposé une métrique intéressante permettant d'évaluer directement l'efficacité du compromis surface / performance. Celle-ci est calculée de la manière suivante:

$$D = \frac{1}{A_{CONS}.T} \quad [\lambda^{-2}.s^{-1}] \textbf{\textit{(16)}}$$

Elle permet, entre autres, de mesurer l'efficacité d'une architecture à implémenter un algorithme en prenant en compte la surface (en unité relative) du système dédiée à la prise en charge de l'algorithme, et le temps total (en unité absolue). L'utilisation d'une unité relative pour la surface permet de comparer des systèmes fabriqués dans des procédés technologiques proches. A titre d'exemple, nous exposons dans le tableau 8 quelques résultats comparatifs concernant l'implantation de la DCT (transformée en cosinus discrète) sur différents accélérateurs (utilisés comme coprocesseurs). Une hypothèse est faite sur l'accès aux données supposé ici direct. Dans tous les cas, il s'agit de la meilleure implémentation (optimisation manuelle) pour chaque accélérateur. On relèvera dans ce tableau l'efficacité des architectures reconfigurables à grain épais. A noter toutefois que certaines surfaces relèvent d'estimations de synthèse à prendre en compte donc avec précaution.

Tab. 8. *Comparaisons basées sur la métrique D. L'algorithme considéré est la DCT, appliqué à des blocs 8*8 d'images en 64*64 pixels en 256 niveaux de gris.*

Nom	*Techno.*	*Implement.*[1]	N_{OP}	*# cycles*[2]	*f (MHz)*	*T (µs)*	A_{CONS}[3] *(Mλ²)*	*D*
VirtexII [Xili02]	0.15µm	Signal Flow	N/A[4]	6144	140	43.88	8000[5]	$0,28.10^{-5}$
S. Ring [Beno03a]	0.18µm	Matricielle	12	6826	200	34.13	260	$11,2.10^{-5}$
DART [Davi02]	0.18µm	N/A[6]	24	9536	130	73.35	300	$4,54.10^{-5}$
MorphoSys [Sing00]	0.35µm	Signal Flow	64	1344	100	13.44	5500	$1,35.10^{-5}$
C62 [Texa00]	0.15µm	Matricielle	8	10240	300	34.13	750	$3,90.10^{-5}$

[1] Le format de données utilisé est *short,* entier court (16 bits).
[2] Les cycles de calcul ne tiennent pas compte, dans tous les cas, des stockages mémoire.
[3] Nous avons considérée dans chacun des cas que la A_{CONS} correspondait à la surface totale du cœur
[4] Nombre d'opérateurs arithmétiques synthétisés non disponible
[5] Le nombre de slices utilisés est de 2400 (2 slices / CLB avec A_{CLB} = 6,78 Mλ² [Ebel03])
[6] Type d'implémentation non disponible

2.3. Caractérisation matérielle

A l'utilisation, c'est la caractérisation extrinsèque qui permet réellement d'évaluer la meilleure solution pour une classe d'applications implantées sur plusieurs composants. Mais pour qu'un ingénieur système compare des architectures au plus tôt dans le flot de conception, l'utilisation de telles métriques semble difficile à mettre en œuvre. En effet, pour calculer les métriques extrinsèques, il faut disposer d'informations comme le temps de traitement, la surface, la puissance consommée qu'il est difficile d'obtenir à un haut niveau d'abstraction : cela nécessite des phases de synthèses et de simulations coûteuses en cycles de développement. D'autre part, les outils d'exploration s'intègrent très tôt dans le flot de conception et leur but est de générer l'architecture ou la configuration qui répond le mieux à une classe d'applications. Notre objectif est différent puisqu'il consiste à caractériser des architectures existantes afin de les comparer suivant un ensemble de critères intrinsèques.

Les métriques que nous proposons s'inspirent des quatre critères de performance définis dans le chapitre 1, à savoir le temps, la consommation, la flexibilité et les coûts. Parmi ces critères, plusieurs correspondent à des grandeurs physiques (secondes, Watts/Joules, euros, mm²). Or, pour avoir une valeur numérique fiable, le système doit être dans une phase avancée du processus de conception ou bien alors carrément fabriqué. Pour un ingénieur système qui cherche à comparer plusieurs approches à un stade précoce, il n'est pas raisonnablement acceptable d'aller aussi loin dans le processus de conception, dans le seul but de comparer plusieurs approches. Il faut donc trouver un moyen de comparaison qui permettrait de faire abstraction de l'implantation physique du circuit, mais qui aurait une signification réelle des performances, ce qui justifierait l'intérêt de cette approche.

Que va-t-on chercher à comparer ? Ce sont les propriétés mêmes des architectures, c'est-à-dire leurs opérateurs, leurs éléments d'interconnexion et la hiérarchie, les modes de contrôle et leur conséquence sur le fonctionnement, la surface relative, etc. Cependant, il serait présomptueux de vouloir être exhaustif : la diversité des points à considérer pourrait noyer l'information utile. Nous nous sommes restreints aux informations nous semblant les plus pertinentes. Pour cela, nous proposons un ensemble de métriques constitué de : la densité opérative qui caractérise le compromis puissance de traitement/surface, la rémanence qui caractérise le dynamisme de l'architecture, le taux d'interconnexion qui représente la flexibilité d'associativité des opérateurs, le taux d'accélération qui illustre le gain en vitesse par rapport à une implantation purement séquentielle et plusieurs fonctions de coûts permettant l'évaluation de l'efficacité énergétique des architectures de processeurs. Ces métriques s'appliquent à toute architecture de traitement flexible, que ce soit un DSP ou une architecture reconfigurable.

2.3.1. Définitions et notations

Avant de définir plus en détail les métriques mises en place, il est au préalable nécessaire de présenter les notations que nous allons utiliser. Pour cela, nous nous appuyons sur le modèle d'architecture, le modèle de calcul et la méthode d'adéquation proposés dans la première partie de ce chapitre.

- N : nombre d'opérations d'un cœur de boucle,
- N_{ITER} : nombre d'itérations d'un cœur de boucle,
- β : facteur de duplication du motif de calcul (*e.g.* déroulage de boucle),

- N_{OP} : nombre total d'opérateurs arithmétiques pouvant être utilisé simultanément (multiplieur, UAL, MAC etc.), et N_C le nombre d'opérateurs configurables à chaque cycle,
- G (bits) : il s'agit de la granularité des opérateurs, *i.e.* la donnée atomique manipulable par l'architecture,
- A_{COEUR} : surface totale du cœur. Elle correspond à la somme de la surface de tous ses éléments (A_{OP} correspond à la surface d'un opérateur, A_{INTER} à la surface de tous les éléments d'interconnexion, A_{CONFIG} à la surface occupée par les registres de configuration, $A_{CONTROLE}$ à la surface dédiée au contrôle). Cette surface est calculée en unité relative, λ^2, comme suit :

$$A(M\lambda^2)=\frac{A(\mu m^2)}{(L/2)^2} \qquad (1)$$

où L ($\mu m/\lambda$) représente la longueur de grille minimale dans une technologie donnée,

- F_e : fréquence de fonctionnement des opérateurs, F_c leur fréquence de configuration,

2.3.2. Densité Opérative [Beno02c]

On présente souvent la puissance de traitement d'une architecture sous la forme de MIPS[1] ou de MOPS[2] :

$$MOPS = N_{OP} * F_e \quad \textbf{(2)}$$

Cette métrique est fonction du chemin critique du circuit et de la technologie silicium utilisée (la fréquence de fonctionnement) : pour les raisons évoquées plus haut, nous préférons nous affranchir des facteurs technologiques.

Pour caractériser le compromis fait entre ce potentiel opératif (N_{OP}) et la surface relative ($A(\lambda^2)$), nous utilisons la « densité opérative », D_{OP}, qui se calcule de la manière suivante :

$$D_{OP}=\frac{N_{OP}}{A_{COEUR}} \text{ [Nombre d'opérateurs.}\lambda^{-2}\text{]} \quad \textbf{(3)}$$

$$\text{où } A_{COEUR} = N_{OP} * A_{OP} + A_{INTER} + A_{CONFIG} + A_{CONTROLE} \quad \textbf{(4)}$$

D_{OP} représente le potentiel « opératif » par rapport à la surface du cœur. Afin de pouvoir comparer des architectures de granularité différente, nous pondérerons la valeur de D_{OP} par la largeur du chemin de données.

Il est également intéressant de tracer l'évolution de D_{OP} en fonction de N_{OP}. La courbe obtenue offre ainsi un aperçu de la *scalabilité*[3] de l'architecture. Ainsi, un cœur générique intégré à une plate-forme de conception peut-être personnalisé suivant les besoins opératifs et les contraintes de surface. Ce tracé peut être également utilisé comme une référence permettant d'apporter des modifications à un modèle d'architecture lors de sa conception. Lorsque la valeur de D_{OP} devient trop faible, le concepteur peut alors décider d'introduire un niveau de hiérarchie supplémentaire. Nous verrons un exemple concret à la fin de ce chapitre.

[1] MIPS : Millions d'Instructions Par Seconde
[2] MOPS : Millions d'Opérations Par Seconde
[3] Cet anglicisme signifie « mis(e) à l'échelle »

2.3.3. Taux d'interconnexion

Il est tout à fait possible d'imaginer une architecture très puissante en termes de ressources de calcul et complètement inutilisable : par exemple, une architecture intégrant un grand nombre d'opérateurs, mais trop peu de ressources de connexion. Autrement dit, une architecture doit pouvoir mettre à disposition des ressources lui permettant d'implanter les applications et ainsi, de tirer parti de leur potentiel opératif.

Le taux d'interconnexion est destiné à représenter le pourcentage du nombre de connexions simultanées possibles par rapport à un réseau d'interconnexion complet (équivalent à un réseau point à point). En d'autres termes, il représente l'inverse de la contrainte d'implantation : en effet, pour un réseau d'interconnexion complet, le taux est de 100% et la contrainte nulle. Cette métrique ne tiendra pas compte des liaisons vers la mémoire et les entrées/sorties, mais seulement des ressources mises à disposition entre tous les opérateurs de l'architecture. Elle donne également une indication sur la capacité de routage de l'architecture.

Le taux d'interconnexion (τ_{INTER}) est calculé de la manière suivante :

$$\tau_{INTER}=100\times\frac{\sum_i C_i}{N_{OP}(N_{OP}-1)} \quad \textbf{(5)}$$

Dans le cas où chaque élément de traitement pourrait être connecté à tous les autres éléments par un bus non partagé, le nombre d'interconnexion total est alors donné par $N_{OP}(N_{OP}-1)$. La somme des Ci représente la totalité des connexions simultanées possibles. La figure 65 illustre le calcul de τ_{INTER} dans plusieurs cas.

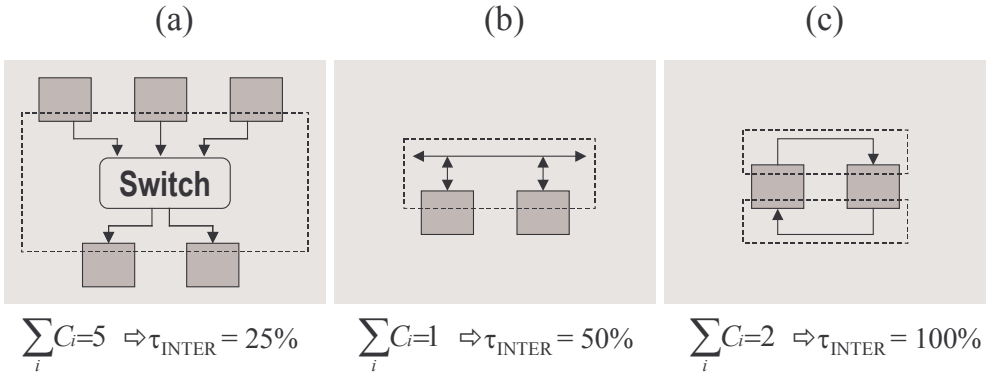


FIG. 65 - *Exemples de calcul de taux d'interconnexion pour : (a) une architecture flot de données avec un switch et 5 opérateurs, (b) deux opérateurs et un bus, (c) deux opérateurs et deux connexions point à point*

Une information supplémentaire sur la mesure du réseau concerne la représentation de la hiérarchie. Dans cette optique, nous pouvons fournir à chaque niveau de la hiérarchie du réseau une valeur représentative qui sera le flux d'entrée/sortie, calculé sur un opérateur.

2.3.4. Rémanence [Demi02]

Le dynamisme d'une architecture flexible peut être caractérisé par la mesure liée au temps nécessaire à la reconfiguration du composant. A cet effet, on utilise la rémanence qui se calcule de la manière suivante :

$$R=\frac{N_{OP}.F_e}{N_c.F_c} \quad \textbf{(6)}$$

où R est égal au nombre de cycles nécessaires pour reconfigurer la totalité des opérateurs. L'inverse de R caractérise le dynamisme de l'architecture (plus R tend vers 1, plus l'architecture est dynamique). Puisqu'il s'agit d'un rapport, remarquons que le calcul de la rémanence est à la fois indépendant de la granularité de l'architecture et de la technologie silicium.

Cette métrique indique qu'il faut R cycles pour reconfigurer l'ensemble de l'architecture. Cela signifie, et c'est en ça que cette métrique est discriminante, qu'elle donne indirectement une indication du volume de données minimum à traiter entre deux configurations pour que l'architecture soit efficace. En règle générale, le nombre de cycles de traitement doit être supérieur de 10 à 20 fois R pour que le coût de configuration soit faible : la rémanence indique donc un ordre de grandeur du volume de données à traiter entre deux configurations.

Ce critère possède au moins trois qualités :

- Il s'affranchit du grain de l'architecture. Un opérateur peut tout aussi bien être un CLB de FPGA, un multiplieur ou un additionneur d'un chemin de données, une UAL de microprocesseur. Le fait qu'un rapport d'opérateurs soit utilisé dans la définition de R neutralise la notion de grain.
- Même si, dans certaines architectures, on ne reconfigure que les chemins entre opérateurs, ceci revient à bloquer temporairement l'exécution liée à ces opérateurs : c'est donc fonctionnellement équivalent. Il se peut cependant que ce soit plus efficace que de reconfigurer directement les opérateurs. A puissance de calcul équivalente, la valeur de Nc peut être plus élevée (reconfiguration plus rapide) et/ou nécessiter moins de bits de configuration (débit de la mémoire de configuration plus faible).
- Peu importe la façon dont s'effectue la reconfiguration. Elle peut s'effectuer "en un coup", une fois tous les traitements liés à la configuration actuelle achevés, ou bien s'effectuer progressivement par vagues de configuration.

2.3.5. Facteur d'accélération

On suppose a priori qu'un système sur puce dispose d'un processeur généraliste. Ce qui intéresse alors l'ingénieur système, c'est d'estimer le gain qu'il peut obtenir sur le temps de traitement grâce à son accélérateur. Il est possible d'évaluer ce gain au niveau système par la loi de Amdahl. Au niveau du coprocesseur, on va mesurer le gain par rapport à une implémentation purement séquentielle.

D'après le modèle de calcul, le nombre de cycles minimum requis pour un traitement purement séquentiel est $N*N_{ITER}$ cycles, en considérant que le nombre de cycles par opération est de 1. Cette valeur est utilisée comme référence pour calculer F_{ACC}, le facteur d'accélération potentiel d'une architecture. Pour une architecture de traitement flexible, il faut non seulement considérer les cycles de calcul mais aussi les cycles de configuration.

La métrique F_{ACC} est obtenue de la manière suivante :

$$F_{ACC} = \frac{N.N_{ITER}}{(C_{COMP} + C_{CONF})} \quad (7)$$

où C_{COMP} est le nombre de cycles requis pour achever le traitement des $N.N_{ITER}$ opérations et C_{CONF} le nombre de cycles nécessaire pour configurer l'architecture à ce traitement.

Pour une architecture donnée, la borne supérieure du facteur d'accélération est son nombre d'opérateurs N_{OP}. Autrement dit, pour une boucle donnée, l'accélération sera d'autant meilleure que le nombre d'opérateurs utilisés sera grand.

Si les temps de configuration sont grands devant $N.N_{ITER}$, alors F_{ACC} tend vers 0. Dans ces cas, l'utilisation de l'architecture ne permet pas d'accélération du traitement. Pour un motif de traitement entièrement parallèle, $C_{COMP}=N_{ITER}.n_{conf}$, où n_{conf} est le nombre de configurations, et $C_{CONF}=R.\ n_{conf}$. On démontre alors que pour une utilisation maximale des ressources:

Si $R/N_{ITER} \rightarrow 0$, *alors* F_{ACC} *tend vers son maximum*(N_{OP})

D'un point de vue système, cette métrique peut être intégrée dans la loi de Amdahl en tant qu'accélération de la partie améliorée.

2.3.6. Aspects énergétiques

a. Efficacité énergétique

La métrique ξ suivante est généralement utilisée pour caractériser l'efficacité énergétique d'une architecture(on ne considère, en première approximation, que la puissance dynamique) :

$$\xi=\frac{MOPS}{mW}=\frac{N_{OP}F_e}{A_{COEUR}.\alpha.C.V_{DD}^2.F_e}=\frac{N_{OP}}{A_{COEUR}.\alpha.C.V_{DD}^2}=\frac{D_{OP}}{\alpha.C.V_{DD}^2} \quad \textbf{(8)}$$

où C est la capacité élémentaire, α l'activité du circuit, et V_{DD} la tension d'alimentation.

Cette métrique est proportionnelle à la densité opérative D_{OP}. Autrement dit, une architecture affichant une densité opérative élevée implique une efficacité énergétique potentiellement plus élevée. Cette efficacité est d'autant plus grande que l'activité du circuit α est réduite, *i.e.* que le nombre de commutations lors d'un traitement est faible.

b. Activité de configuration

L'activité d'un circuit correspond au nombre de commutations de tous ses transistors. L'activité est composée de quatre termes suivant notre modèle :

$$\alpha_{TOTAL}=\alpha_{CONFIG}+\alpha_{INTER}+\alpha_{OP}+\alpha_{CONTROLE} \quad \textbf{(9)}$$

Pour les parties « interconnexion » et « opérateurs », l'activité est fonction du nombre de transistors impliqués dans le traitement, donc de la « surface de l'application » et du nombre de données traitées. D'une manière globale, chacune de ces activités est pondérée par la surface que représente ses unités.

Pour un processeur, 50% de sa consommation peut être due à sa partie contrôle et configuration. Outre les aspects technologiques, le volume des données de configuration et la méthode de reconfiguration sont des éléments qui influencent l'activité totale et donc, la consommation. Si, durant une application, le contenu des registres de configuration reste inchangé, on minimise l'activité totale et par conséquent, on améliore l'efficacité énergétique.

Contrairement aux autres activités, l'activité de configuration peut être facilement évaluée à partir de la rémanence de l'architecture, la taille *i* des instructions de configuration, du nombre de configurations n_{conf} et du mode de configuration (θ). Elle représente le volume binaire total (sur toute la durée du traitement) impliqué dans la configuration de l'architecture. Celle-ci est calculée de la manière suivante :

$$\alpha_{CONFIG}=R.i.\theta.n_{conf} \quad \textbf{(10)}$$

$$\text{où } n_{conf} = \frac{\beta N}{N_{OP}} \quad \textbf{(11)}$$

On suppose, dans ce cas également, que le parallélisme est déployé au maximum. Le paramètre n_{conf} correspond au nombre de configurations chargées ; précisons que pour calculer sa valeur, β est choisi tel que n_{conf}=1si N est inférieur à N_{OP}, sinon β=1.

Le mode de configuration de l'architecture (θ) est un critère influençant largement l'activité de configuration. Suivant un modèle de calcul séquentiel, θ est proportionnel au volume de données (reconfiguration systématique). Suivant un modèle de calcul figé, θ=1.

2.3.7. Synthèse

Nous avons présenté dans cette partie un rapide état de l'art des contributions dans le domaine de la caractérisation. Nous avons en particulier souligné le manque de métriques permettant de caractériser le potentiel des architectures existantes, afin de pouvoir les comparer tôt dans le flot de conception des systèmes. Nous avons alors proposé un certain nombre d'outils de mesure pour caractériser rapidement le compromis des performances d'une architecture : son potentiel opératif par rapport à sa surface (D_{OP}), son taux de connectivité par rapport à un réseau point à point(τ_{INTER}), son dynamisme (R), ses capacités d'accélération et son activité de configuration suivant les caractéristiques de l'application (F_{ACC}, et α_{CONFIG}). Nous allons voir dans la suite de ce chapitre, deux applications des métriques intrinsèques : d'une part pour la caractérisation puis la comparaison de plusieurs architectures d'accélérateurs flexibles, et d'autre part pour la caractérisation de la *scalabilité* de modèles abstraits d'architectures génériques, afin d'explorer leur espace de conception.

3. RESULTATS

Lorsqu'on veut comparer des architectures tôt dans le flot de conception, il est difficilement possible de s'appuyer sur des grandeurs physiques. Nous proposons une approche alternative permettant de comparer les architectures suivant des critères intrinsèques que nous avons définis de manière théorique dans la partie précédente. Ici, nous prenons le cas pratique de 3 architectures que nous allons caractériser par les métriques proposées. Ensuite, nous étudierons la *scalabilité* de modèles génériques par le biais de ces mêmes outils, afin d'anticiper sur les performances des futures générations d'accélérateurs. Nous analyserons les résultats obtenus et essaierons de conclure sur la couverture de ces métriques.

3.1. Caractérisation et comparaison des accélérateurs flexibles

Nous présentons dans cette partie des exemples de caractérisation de trois architectures (une architecture à grain fin FPGA, une architecture reconfigurable à grain épais et un DSP VLIW). Une synthèse des résultats sera discutée afin de montrer l'intérêt de ces métriques. (Remarque : nous supposons que tous les opérateurs sont utilisés, *i.e.* que si le motif de calcul contient moins d'opérations que ce que l'architecture dispose d'opérateurs, le motif est alors dupliqué pour occuper toute la surface de l'accélérateur ; sinon, le motif est découpé de manière à générer un minimum de configurations)

3.1.1. Caractérisation

a. *FPGA AT40K*

L'AT40K est un FPGA à grain fin d'ATMEL qui est utilisé dans la plate-forme ARDOISE (cf. Chapitre 2) pour ses capacités de reconfiguration dynamique. Dans le tableau 9, nous présentons deux valeurs pour la densité opérative de ce FPGA. La première calculée pour un chemin de données sur 1 bit et la deuxième pour un chemin de données sur 24 bits. Compte-tenu de la granularité de ce circuit, la densité opérative est plus élevée pour des traitements logiques d'où une meilleure efficacité que pour des traitements sur des données plus larges. Notons qu'en normalisant la première valeur sur un chemin de données 24 bits, la densité opérative reste plus élevée (environ quatre fois plus) ce qui montre ici la surface additionnelle impliquée dans la synthèse des 26 UAL 24 bits.

Tab. 9. *Densité opérative des AT40K*

G (bits)	L (bits)	N_{OP}	A_{COEUR} ($M\lambda^2$)	D^1_{OP}	D_{OP}
1	1	2304	6000	$38{,}4 \cdot 10^{-2}$	$38{,}4 \cdot 10^{-2}$
1	24	26	6000	$9{,}6 \cdot 10^{-2}$	$0{,}4 \cdot 10^{-2}$

En considérant chaque position des cellules du FPGA (coin, arêtes, centre) les connexions point à point avec les huit voisins directs (en entrée et en sortie d'où $\varphi_{IN/OUT}$ =8/8), et les 5 plans de connexions par bus pour les connexions plus longues (connexions verticales et horizontales bidirectionnelles $\varphi_{IN/OUT}$ =10), on obtient le taux d'interconnexion et flux d'entrée/sortie du tableau 10. Vu la quantité d'opérateurs (2304 CLB), il n'était pas raisonnablement envisageable pour les concepteurs de ce FPGA de proposer un réseau complet : cela aurait entraîné une surface additionnelle beaucoup trop importante, et donc une densité opérative trop faible. La hiérarchie du système d'interconnexion autorise des connexions point à point au premier niveau (L1 : 8 opérateurs en entrée, 8 en sortie), et des

[1] La densité opérative de cette colonne est normalisée à la largeur du chemin de données (D_{OP}*L).

liaisons bus (10 bus) au deuxième niveau (L2). Les connexions locales sont donc fortement favorisées. Avec un taux d'interconnexion de 0,7%, les taux de remplissage ne seront pas élevés sur des configurations nécessitant de nombreuses connexions, pour les longues distances en particulier.

Tab. 10. *Taux et flux d'interconnexion des AT40K*

Ressources	*Topologie*	$\varphi_{IN/OUT}$	$\varphi_{IN/OUT}$ *(L2)*	$\varphi_{IN/OUT}$ *(L3)*	τ_{INTER}
bus, NN[1]	Grille	8/8	10	/	0.7

Le tableau 11 montre la valeur de la rémanence des AT40K. Ces composants permettant une reconfiguration dynamique partielle, nécessitent 0,5ms pour modifier la fonctionnalité du circuit à la fréquence de configuration de 33MHz

Tab. 11. *Rémanence des AT40K*

N_{OP}	N_{Cty}	*F(MHz)*	*R*
2304[2]	0.14	33	16457

Le tableau 12 représente les facteurs d'accélération des AT40K obtenus sur différentes tailles de motifs de calcul et différents volumes de données. La taille des motifs de calcul impose le nombre de configurations nécessaires à l'implantation de l'application. A la lecture de ces résultats, il est clair que ce composant ne sera pas efficace sur de faibles quantités de données. Les accélérations deviennent réellement intéressantes quand N_{ITER} est supérieur à 10^5 ($R/N_{ITER}>0,16$).

Tab. 12. *Facteurs d'accélération sur deux motifs de calcul*

N \ N_{ITER}	10^0	10^1	10^2	10^3	10^4	10^5	10^6
2	0,00	0,00	0,01	0,12	1,16	8,28	21,42
13	0,00	0,01	0,08	0,77	6,06	19,56	25,17

L'activité de configuration donne un indice relatif de la consommation due au contrôle et aux modes de reconfiguration. Le tableau 13 montre deux applications numériques de α_{CONFIG}. On considère un premier motif (a) de traitement, composé de 2 opérations et dupliqué β=13 fois, ce qui donne taux d'occupation à 100% des opérateurs du FPGA et un volume de données de N_{ITER} =10^6 échantillons (accélération quasi-maximale ≈ 26). Le deuxième motif (b) est quant à lui composé de 52 opérations et nécessite donc 1 reconfiguration de l'AT40K (n_{conf}=2). Dans les 2 cas, on suppose que le contenu des registres de configuration reste inchangé durant toute la phase de traitement (θ=1). Ces résultats seront commentés lors de la synthèse des résultats par rapport aux autres activités de configuration.

Tab. 13. *Activité de configuration pour deux applications*

i	*R*	n_{conf}		α_{CONFIG}	
		(a)	(b)	(a)	(b)
16	16457	1	2	$2,6.10^5$	$5,2.10^5$

[1] NN : « Nearest Neighboor », ou plus proche voisin

[2] C'est le nombre de cellules logiques reconfigurables : on ne distingue pas, dans ce cas uniquement, la granularité des opérateurs (puisque la rémanence est un rapport).

b. Systolic Ring

Chaque opérateur (Dnode) est composé d'une UAL et d'un multiplieur 16 bits. Une caractérisation de la surface du Systolic Ring en fonction de ses paramètres architecturaux a été effectuée. Nous utilisons ici ces résultats pour exprimer la densité opérative en fonction de trois jeux de paramètres. Lors de sa conception initiale, cette architecture disposait d'un seul niveau d'interconnexion dont la surface augmentait de manière quadratique. Pour améliorer la *scalabilité* de cet accélérateur, deux niveaux supplémentaires d'interconnexion ont été introduits dans la hiérarchie. Ainsi, on peut augmenter la quantité d'opérateurs en ne pénalisant que très peu le D_{OP} comme le montre le tableau 14 : la version (2) à 64 opérateurs possède deux niveaux de hiérarchie et un D_{OP} plus faible que la version (1), alors que la version (3), avec un troisième niveau, permet de revenir à la valeur initiale de D_{OP}.

Tab. 14. *Densité Opérative du Systolic Ring pour différents jeux de paramètres*

Valeurs des paramètres	N_{OP}	A_{COEUR} ($M\lambda^2$)	D^1_{OP}	D_{OP}
(1) : S=1, C=6, N=2, B=3	12	360	$52{,}8 \cdot 10^{-2}$	$3{,}3 \cdot 10^{-2}$
(2) : S=1, C=8, N=8, B=3	64	2020	$49{,}6 \cdot 10^{-2}$	$3{,}1 \cdot 10^{-2}$
(3) : S=4, C=8, N=2, B=3	64	1905	$52{,}8 \cdot 10^{-2}$	$3{,}3 \cdot 10^{-2}$

Le tableau 15 illustre des résultats de τ_{INTER} pour trois jeux de paramètres. La version (1) disposant du plus faible nombre d'opérateurs et seulement 2 niveaux de hiérarchie, affiche le meilleur taux d'interconnexion. Pour les deux versions à 64 opérateurs, la valeur de τ_{INTER} montre que la version à deux niveaux (2) dispose d'une plus grande capacité de communication que la version (3) à trois niveaux.

Tab. 15. *Taux et flux d'interconnexion du Systolic Ring*

Valeurs des paramètres	*Ressources*	*Topologie*	$\varphi_{IN/OUT}$	$\varphi_{IN/OUT}$ *(L2)*	$\varphi_{IN/OUT}$ *(L3)*	τ_{INTER}
(1): S=1, C=6, N=2, B=3	switch, bus	Anneau	2/2	3	-	21,2
(2): S=1, C=8, N=8, B=3	"	"	8/8	3	-	1,7
(3): S=4, C=8, N=2, B=3	"	"	2/2	3	1	0,1

Le Systolic Ring fonctionne suivant plusieurs modes de configuration qui impliquent des intervalles variables de rémanence. De plus, la valeur des paramètres génériques a une influence sur R. Ainsi, la rémanence (tableau 16) varie de 1 à 128 avec des valeurs typiques entre 6 et 32 pour les versions considérées, ce qui correspond, à 200MHz, à des temps de configuration inférieurs à 160 ns.

Tab. 16. *Calcul de la Rémanence du Systolic Ring*

Valeurs des paramètres	N_{OP}	$N_{C\,min}$	$N_{C\,ty}$	$N_{C\,max}$	R_{min}	R_{typ}	R_{max}
(1) : S=1, C=6, N=2, B=3	12	0.5	2	12	1	6	24
(2) : S=1, C=8, N=8, B=3	64	0.5	8	64	1	8	128
(3) : S=4, C=8, N=2, B=3	64	0.5	8	64	4	32	128

Cette vitesse potentielle de configuration confère à cette architecture, et plus généralement à ce type d'architecture, des capacités d'accélérations intéressantes sur les moyens et gros volumes de données. Comme nous pouvons le constater sur différentes tailles de motifs de calcul (tableau 17), le facteur d'accélération par rapport à une implémentation

[1] Densité opérative normalisée à largeur du chemin de données 16 bits

séquentielle est visible même pour des volumes de données intermédiaires. Compte-tenu du rapport R/N_{ITER}, le maximum d'accélération (version à 12 opérateurs) est atteint pour des volumes de l'ordre de 10^4 données, et l'accélération est intéressante même sur de faibles volumes (à partir de 10).

Tab. 17. *Facteurs d'accélération sur deux motifs de calcul*

N \ N_{ITER}	10^0	10^1	10^2	10^3	10^4	10^5	10^6
2	1	2,61	8,82	11,58	11,96	12,00	12,00
12	1,71	7,5	11,32	11,93	11,99	12,00	12,00

Tab. 18. *Activité de configuration pour deux applications*

Interconnexions	*i*	*R*	n_{conf}		α_{CONFIG}	
			(a)	(b)	(a)	(b)
(1) : S=1, C=6, N=2, B=3	96	6	1	5,3	576	3052,8
(2) : S=1, C=8, N=8, B=3	312	8	1	1	2496	2496
(3) : S=4, C=8, N=2, B=3	384	32	1	1	12288	12288

Concernant son activité de configuration, le tableau 18 montre 2 applications numériques pour les 3 jeux de paramètres considérés. Le premier motif de traitement (a) est composé de 4 opérations et dupliqué β fois afin de permettre un taux d'occupation à 100% du Systolic Ring, et un volume de données de $N_{ITER} = 10^5$ échantillons (accélération maximale pour toutes les versions). Le deuxième motif (b) est quant à lui composé de 64 opérations et nécessite donc plusieurs configurations (n_{conf}=5,3) pour le premier jeu de paramètres. Dans les 2 cas, on suppose que la configuration reste figée entre chaque reconfiguration (θ=1). Pour le premier motif, l'activité de configuration est minimale pour la version (1), mais c'est aussi celle qui apportera le moins d'accélération. Pour une accélération équivalente, la version (2) affiche un score plus intéressant. Cette tendance est d'ailleurs confirmée par le deuxième motif (b). Le deuxième jeu de paramètres permet même moins d'activité de configuration que le premier. En effet, pour celui-ci, puisqu'il est nécessaire de reconfigurer plusieurs fois, l'activité de configuration s'en trouve affectée.

c. C62 de Texas Instrument [Texa00b]

Ce DSP VLIW intègre 8 unités fonctionnelles dont 4 UAL et 2 multiplieurs (32 bits). Ces unités sont configurées à chaque cycle par une instruction de 256 bits. Les interconnexions entre opérateurs sont réalisées grâce à deux banques de registre multi-ports (16 registres chacune). La densité opérative du C62 est présentée dans le tableau 19, avec deux valeurs dont une est normalisée à 16 bits.

Tab. 19. *Densités opératives pour le DSP C62*

N_{OP}	N_{UAL}	N_{MULT}	A_{COEUR} ($M\lambda^2$)	D^1_{OP}	D_{OP}
8	4	2	750	$32{,}0.10^{-2}$	$1{,}0.10^{-2}$

Avec les deux bancs de registres, chaque opérateur dispose de 4 possibilités de connexions en entrée/sortie, avec un seul niveau de hiérarchie. La valeur de τ_{INTER} est donc de 29%.

[1] La densité opérative de cette colonne est normalisée à la largeur du chemin de données 32 bits.

Tab. 20. *Taux et flux d'interconnexion*

$\varphi_{IN/OUT}$ (L1)	$\varphi_{IN/OUT}$ (L2)	$\varphi_{IN/OUT}$ (L3)	τ_{INTER}
4/4	/	/	29

Le DSP est basé sur une implémentation séquentielle de l'application et donc, le contrôleur lit une instruction à chaque cycle ce qui correspond à une reconfiguration systématique. Les huit opérateurs sont configurés à chaque cycle d'où une rémanence minimale de 1. A une fréquence de 300MHz, seulement 27ns sont requises pour reconfigurer ce DSP.

Avec sa faible rémanence, le DSP peut rapidement atteindre son facteur d'accélération maximal. C'est ce que l'on peut observer sur le tableau 21 : pour des volumes de 10^2 données, le facteur du C62 est déjà très proche de 8.

Tab. 21. *Facteurs d'accélération sur deux motifs de calcul*

N \ N_{ITER}	10^0	10^1	10^2	10^3	10^4	10^5	10^6
2	1,60	5,71	7,69	7,97	8,00	8,00	8,00
12	4,00	7,27	7,92	7,99	8,00	8,00	8,00

Contrairement aux deux architectures précédentes, le DSP ne peut pas figer sa configuration. En conséquence, son activité de configuration est proportionnelle à la quantité de données traitées comme le montre le tableau 22. Le premier motif choisi (a) est constitué de 4 opérations, dupliqué β=2 fois (on suppose un parallélisme maximal et donc le DSP est utilisé à 100% à chaque cycle) et un volume de données de N_{ITER} =10^2 échantillons (accélération quasi-maximale ≈ 8). Le deuxième motif (b) est quant à lui composé de 8 opérations sur 10^5 données. L'utilisation d'une reconfiguration systématique implique des activités de configuration proportionnelles à la quantité de données traitées d'où un accroissement significatif entre les deux motifs de calcul.

Tab. 22. *Activité de configuration du C62 pour deux applications*

i	R	θ		α_{CONFIG}	
		(a)	(b)	(a)	(b)
256	1	50	10^5	12800	256.10^5

3.1.2. Synthèse des résultats et Comparaisons

L'utilisation des métriques intrinsèques permet de caractériser, avant la phase de conception d'un SoC, les performances potentielles de plusieurs cœurs d'accélérateurs flexibles. Nous allons ici comparer les différents résultats afin d'illustrer leur utilisation dans le cadre d'un flot de conception. Pour cela, nous considérons deux motifs de calculs répétitifs représentés par le nombre d'opérations N (2 et 12) et le volume de données N_{ITER} de (10 à 10^5), et les 3 accélérateurs de traitement précédemment caractérisés : les résultats sont résumés dans le tableau 23.

Tab. 23. *Comparaison des architectures suivant 2 motifs algorithmiques*

	N		2				12			
	N_{ITER}		10		10^5		10		10^5	
	D^1_{OP}	τ_{INTER}	F_{acc}	α	F_{acc}	α	F_{acc}	α	F_{acc}	α
AT40K	9,4	0,7	0	$2{,}6.10^5$	8,3	$2{,}6.10^5$	0	$2{,}4.10^5$	**18,4**	$2{,}4.10^5$
S. Ring	**52,8**	21,2	2,6	$0{,}6.10^3$	**12**	$\mathbf{0{,}6.10^3}$	**7,5**	$\mathbf{0{,}6.10^3}$	12	$\mathbf{0{,}6.10^3}$
C62	32,0	**29**	**5,7**	$\mathbf{0{,}2.10^3}$	8	13.10^6	7,2	4.10^3	8	4.10^8

Le nombre d'opérateurs du Systolic Ring (ici égal à 12) et sa rémanence lui permettent, pour les deux motifs, d'obtenir un bon compromis accélération / activité de configuration. La densité opérative plus élevée que les deux autres architectures lui confère également de bonnes prédispositions à l'efficacité énergétique. Les résultats pour le DSP montrent que son utilisation se justifie plutôt sur des petits motifs de calcul et des volumes de données restreints (zone (a), figure 4), par rapport aux deux autres candidats de ce panel. En effet, le coût de l'activité de configuration est largement supérieur aux deux autres architectures dès que la quantité de données devient importante. En revanche, par un taux de connexion supérieur aux autres, les communications entre partitions temporelles lui permettent théoriquement d'exploiter plus facilement le parallélisme.

Le FPGA, par sa forte rémanence, ne permet pas d'atteindre des accélérations élevées sur des faibles volumes de données (zones (a) et (b), figure 66). Cependant, dès que cette quantité devient importante, son facteur d'accélération devient le plus élevé (zone (c), figure 66) avec une activité de configuration faible par rapport au DSP (tableau 23). De plus, la flexibilité de la granularité fine lui permet de s'adapter facilement à toutes les largeurs de données. Cependant, la faible densité opérative caractérise le surcoût matériel du grain fin lorsqu'on l'utilise pour implémenter des opérateurs arithmétiques.

Ces résultats illustrent une utilisation des métriques intrinsèques lorsque l'on veut choisir, parmi plusieurs solutions existantes, l'accélérateur correspondant aux besoins des applications ciblées. L'évaluation des métriques ne s'effectue qu'à partir de considérations architecturales, et sont donc simples à mettre en œuvre : elles ne nécessitent pas par exemple de simuler les diverses applications. L'utilisation d'une surface relative permet également de s'affranchir d'une synthèse de l'architecture dans la technologie ciblée pour le système. Ces résultats montrent aussi que le compromis de performances est théoriquement dépendant des caractéristiques du code à accélérer. Ainsi chaque architecture peut avoir son intérêt suivant le motif et le volume de données. Avec les capacités d'intégration actuelles, il est tout à fait possible d'imaginer un système disposant à la fois de cœurs DSP, reconfigurables à grain fin et à grain épais. Les métriques intrinsèques peuvent alors être utilisées très tôt dans le flot de

[1] Valeurs normalisées de la densité opérative

conception, afin de donner une solution initiale de la répartition des différents motifs sur les divers accélérateurs du système.

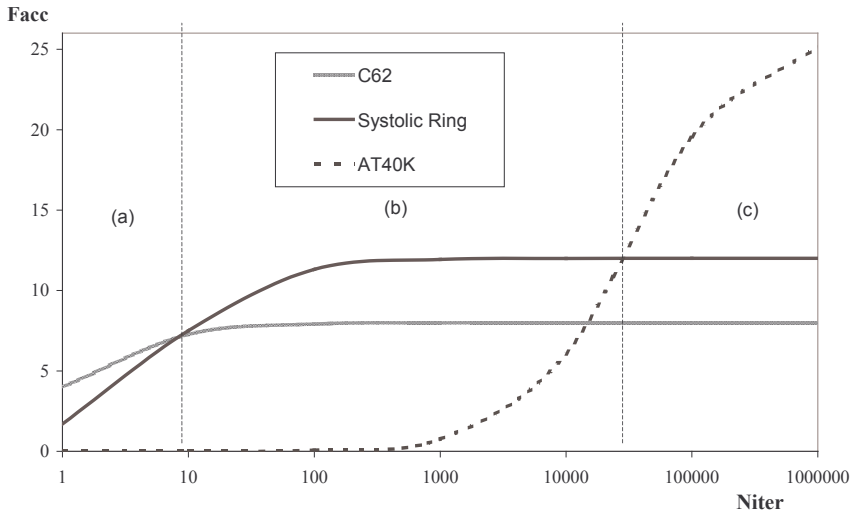


FIG. 66 - *Accroissement du facteur d'accélération pour les trois architectures, en fonction de* N_{ITER}

3.2. Analyse de la *scalabilité* de modèles abstraits

Les premiers résultats présentés ont permis de caractériser et de comparer différents types d'architectures de traitement existantes. Une autre application des métriques que nous proposons consiste à caractériser des modèles abstraits paramétrables. Dans cette partie, nous allons mettre en œuvre cette deuxième application de nos outils de mesure. Dans l'espace de conception des architectures reconfigurables à grain épais, les deux modèles choisis peuvent être considérés comme deux solutions extrêmes. Un certain nombre d'hypothèses seront tout d'abord énoncées. Ensuite, nous présenterons puis comparerons ces modèles sur la base des métriques intrinsèques. L'objectif de cette étude repose principalement sur une analyse de la *scalabilité* en fonction des paramètres architecturaux.

3.2.1. Hypothèses et exemple de modèles grain épais

Tab. 24. *Hypothèses des modèles d'architectures reconfigurables à grain épais*

Largeur du chemin de données	16 bits
Type des éléments de traitement	UAL + Multiplieur
Nombre d'entrées/opérateur	2
Nombre de sorties/opérateur	1
Nombre de bits de configuration/opérateur	20 bits
Nombre de bits de configuration/(interconnexion et opérateur)	Log_2 (#d'entrées)
Surface d'un élément de traitement	A_{OP} = 20 Mλ²
Surface d'un élément d'interconnexion	A_{INTER} = 0,2 Mλ²
Surface du contrôle/opérateur ou /interconnexion	A_{SEQ} = 1 Mλ²
Surface de configuration / bit	A_{CONFIG} = 0,2 Mλ²

Les hypothèses (tableau 24) concernent la granularité des opérateurs, le type et le nombre de ressources qui composent les deux modèles abstraits. Nous supposerons dans les deux cas, que le type d'opérateurs utilisé est basé sur l'utilisation à la fois d'une UAL et d'un multiplieur (semblable à ceux de MorphoSys [Sing00] et Systolic Ring [Sass01]). En ce qui concerne la topologie d'interconnexion, nous supposerons dans les deux cas un réseau systolique de type *grille 2D*. Une troisième hypothèse concerne le contrôle des configurations : nous supposerons de la logique (type machine d'état) associée à chaque élément d'interconnexion et chaque nœud de traitement. Enfin, concernant la mémoire de configuration, nous proposons de normaliser le nombre de bits par opérateur et élément d'interconnexion, ainsi qu'une surface unitaire par bit de configuration. Pour les surfaces fournies dans ce qui suit, une analyse des modèles Systolic Ring [Beno03c] et MorphoSys a permis d'extraire des valeurs vraisemblables de surface, proposées ici en Mλ² afin de s'affranchir de la dimension caractéristique de la technologie utilisée.

A partir de ces hypothèses, nous allons désormais définir deux modèles d'architecture abstraits, mais basé sur deux topologies d'interconnexion plus ou moins limitatives, ainsi qu'un nombre d'opérateurs reconfigurables par cycle différent.

a. Modèle A : réseau d'interconnexion complet

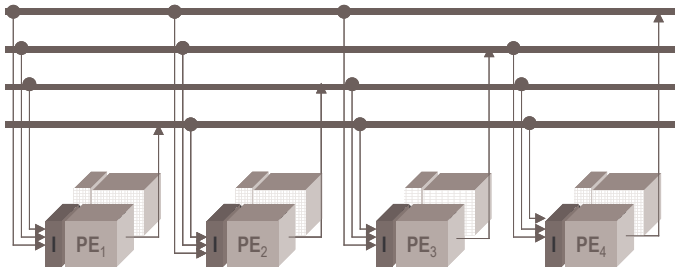


FIG. 67 - *Illustration du modèle A avec N_{OP}=4*

Le modèle « complet » suppose un réseau d'interconnexion permettant entre chaque opérateur de la grille une liaison directe avec un autre opérateur de la grille. L'avantage d'une telle inter-connectivité réside dans la faible contrainte de projection algorithmique (seulement une contrainte sur le nombre d'opérateurs disponibles) qu'elle implique lors de la phase de projection de l'application. Une illustration de ce modèle (seulement une ligne d'opérateurs et d'interconnexions) est proposée figure 67.

Tab. 25. *Formalisation du modèle A*

# d'opérateurs	N_{OP}
# d'op. configurables / cycle	1
# d'interconnexions	$N_{OP}*(N_{OP}-1)$
# d'éléments de contrôle	$N_{OP} + N_{OP}*(N_{OP}-1) = N_{OP}^2$
# de bits de configuration	$N_{OP} * 20 + N_{OP}*\log_2(N_{OP}-1)$
surface du cœur	$A_{COEUR}=N_{OP}A_{OP}+N_{OP}(N_{OP}-1)A_{INTER}+N_{OP}^2A_{SEQ}+(20N_{OP}+N_{OP}\log_2(N_{OP}-1))Abit$

Dans le tableau 25, nous avons résumé les propriétés concernant N_{OP}, N_C et les différentes surfaces formalisées que nous utiliserons dans la suite pour étudier la *scalabilité* de ce modèle.

b. Modèle B : réseau d'interconnexion limité en anneau

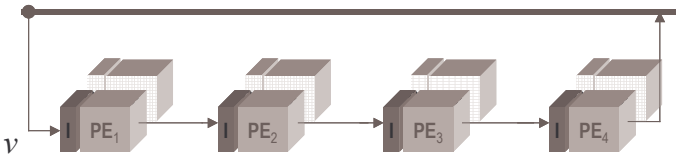


FIG. 68 - . *Illustration du modèle n°2 avec N_{OP} = 4*

Le modèle B est basé sur un réseau d'interconnexion en anneau (sortie du dernier opérateur bouclée sur l'entrée du premier). L'intérêt de ce type de topologie est d'accroître la profondeur du pipeline sans augmenter la quantité d'opérateurs. Dans notre cas une seule liaison directe avec un opérateur aval de la même ligne et un de la même colonne est possible. Les limitations d'interconnectivité du modèle B laissent supposer des contraintes assez fortes pour la projection des motifs de calcul sur l'architecture. Une illustration de ce modèle est proposée figure 68 (exemple de représentation d'une seule ligne d'opérateurs et d'interconnexions).

Comme pour le modèle A, nous résumons les différentes propriétés de ce modèle afin d'en étudier la *scalabilité* dans le tableau 26.

Tab. 26. *Formalisation du modèle B*

# d'opérateurs	N_{OP}
# d'op configurables / cycle	$N_{OP}^{1/2}$
# d'interconnexions	$2*(N_{OP}^{1/2}-1)*(N_{OP}^{1/2})$
# d'éléments de contrôle	$N_{OP} + 2*(N_{OP}^{1/2}-1)*(N_{OP}^{1/2})$
# de bits de configuration	$N_{OP}*20 + N_{OP} = 21* N_{OP}$
surface du cœur	$A_{COEUR}=N_{OP}A_{OP}+2(N_{OP}^{1/2}-1)(N_{OP}^{1/2})A_{inter}+[N_{OP}+ 2(N_{OP}^{1/2}-1)(N_{OP}^{1/2})]A_{SEQ}+21*N_{OP}*Abit$

3.2.2. Comparaison et analyse des deux modèles

Nous représenterons sous formes de courbes les différentes métriques en fonction du nombre d'opérateurs, ce qui permettra de comparer l'évolution des propriétés des deux modèles proposés et donc de conclure sur leur *scalabilité*.

a. Analyse de D_{OP}, R et de τ_{inter}

L'analyse de la densité opérative montre clairement que le modèle A n'est pas *scalable* suivant la surface, alors que le second permet de conserver une densité opérative quasi-constante (figure 69). Ceci montre que le coût de l'augmentation du nombre d'éléments de traitement sera moindre pour le modèle n°2. Si un concepteur fait le choix du modèle B, il peut néanmoins envisager d'introduire une hiérarchie dans le réseau d'interconnexion, afin de limiter le surcoût dû à la complétude du modèle choisi. D'ailleurs, en ce qui concerne la *scalabilité* du taux d'interconnexion, nous pouvons observer (figure 69) que celle-ci reste constante pour le réseau complet (100%) alors que pour le réseau en anneau la décroissance est relativement rapide. Cette caractéristique permet d'évaluer ici les conséquences néfastes que cela peut avoir sur les contraintes de projection des motifs de calcul, notamment dans le cas de divergences et/ou convergences dans le flot de données à implanter.

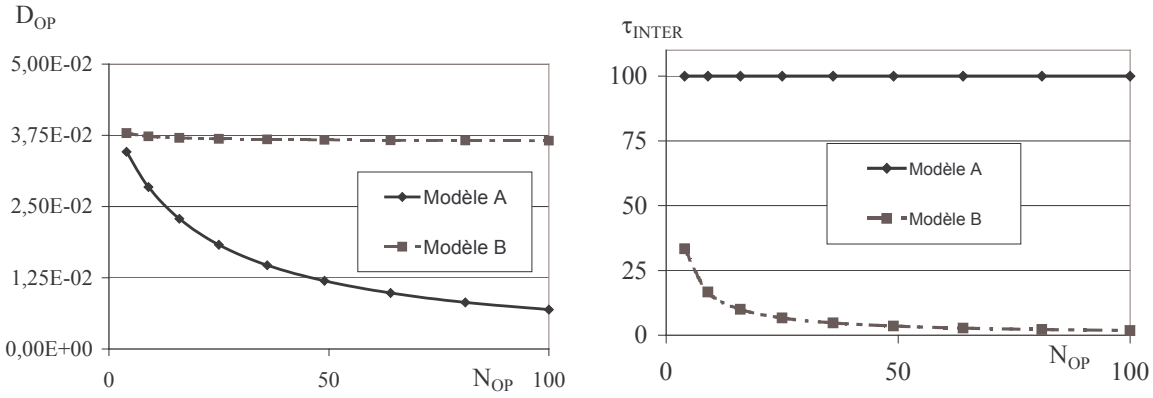


FIG. 69 - *Comparaison de D_{OP} et τ_{INTER} pour les modèles A et B, en fonction de N_{OP}*

Le modèle de configuration de l'architecture B implique une rémanence moins élevée et donc un dynamisme plus important (figure 70). Ceci peut avoir des conséquences sur le facteur d'accélération, comme nous le verrons dans le paragraphe suivant.

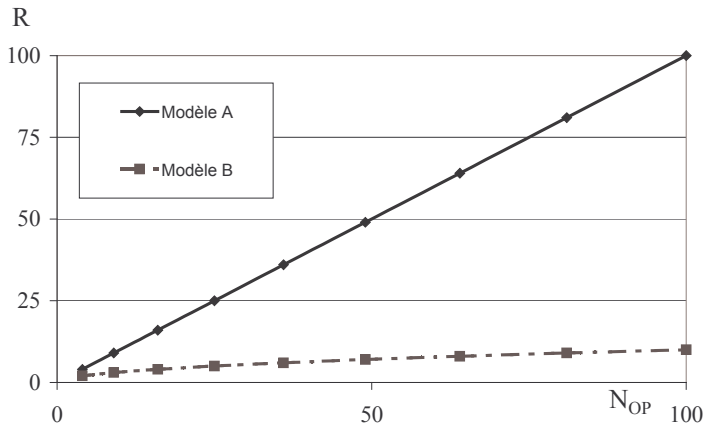


FIG. 70 - *Comparaison de R pour les 2 modèles, en fonction de N_{OP}*

b. Analyse de F_{acc} et aspects énergétiques

Nous avons choisi différents cœurs de boucles pour illustrer les facteurs d'accélération et les activités de configuration. Sur le graphique de la figure 71, nous avons tracé quatre fonctions correspondant aux facteurs d'accélération des modèles A et B, sur des motifs de calcul de 4 opérations, et des volumes de données de 100 et 10000 échantillons. En considérant une exploitation maximale du parallélisme, nous avons tracé l'évolution des taux d'accélération pour un nombre d'opérateurs croissant. La première constatation est l'observation d'une légère décroissance de ce taux d'accélération pour le modèle A basé sur une rémanence égale au nombre d'opérateurs (à partir de N_{OP}=16). En effet, compte tenu de cette caractéristique, les cycles de configurations sont suffisamment grands pour contraindre le facteur d'accélération. Ceci s'observe notamment sur les faibles volumes de traitement, comme dans le premier cas. En revanche, dès que ce volume devient important, la rémanence de l'architecture devient négligeable devant les cycles de calcul à effectuer. Ceci se manifeste clairement sur le graphique avec une allure de courbe quasi-identique à celle de l'architecture B, pourtant largement plus dynamique que le réseau complet.

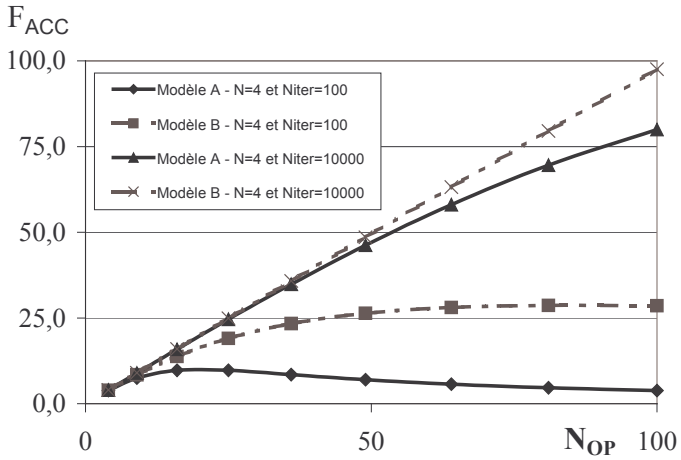


FIG. 71 - *Comparaison des F_{ACC} pour les 2 modèles, en fonction de N_{OP}*

Nous observons sur le graphique de la figure 72 que α_{CONFIG} croît de manière linéaire dans les deux cas, avec un taux d'accroissement légèrement inférieur pour l'architecture B. L'utilisation du deuxième modèle est a priori plus avantageuse si l'on veut consommer moins d'énergie, à puissance de traitement égale. De plus, si l'on s'en réfère aux résultats de D_{OP}, le modèle B est beaucoup plus *scalable* que le A. Donc à fréquence identique, il est préférable

de choisir un système d'interconnexion proche du modèle B si l'on veut préserver l'efficacité énergétique de l'accélérateur.

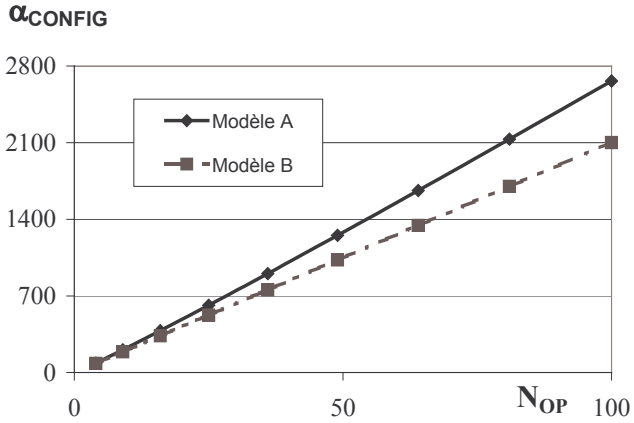


FIG. 72 - *Comparaison des α_{CONFIG} pour les 2 modèles, en fonction de N_{OP}*

3.2.3. Conclusion

Cette analyse nous a permis d'illustrer, sur deux exemples, l'évolution des propriétés d'un modèle d'architecture, *i.e.* sa *scalabilité* en fonction du nombre d'opérateurs. Cette analyse peut également être appliquée à un cas réel comme cela a été réalisé dans [Beno03] : cette étude a d'ailleurs menée au développement d'un outil d'exploration architecturale pour le Systolic Ring.

Ici, nous avons choisi d'illustrer cette étude par deux modèles génériques abstraits, calibrés sur des modèles d'architectures réelles à grain épais. Les résultats obtenus permettent d'évaluer l'évolution des compromis de performances en fonction du nombre d'opérateurs. Dans la mesure où les concepteurs cherchent à augmenter la puissance de traitement de leurs accélérateurs par l'intégration d'un nombre toujours plus élevé d'opérateurs, cette étude permet d'illustrer les bornes de l'espace des solutions possibles pour ce type d'architectures.

3.3. Bilan

3.3.1. Intérêts et limitations

Comme nous l'avons vu dans cette troisième partie, les métriques intrinsèques matérielles peuvent être utilisées pour caractériser des architectures de traitement (accélérateurs), pour comparer ces architectures, et pour paramétrer des modèles.

Les propriétés du modèle de calcul que nous avons choisies correspondent, moyennant un certain nombre d'hypothèses, au champ d'applications ciblées par les accélérateurs que nous avons comparés. Les résultats que nous avons obtenus montrent que les architectures reconfigurables à grain épais disposent des caractéristiques les mieux adaptées. Cependant, il ne faut pas oublier que certaines valeurs doivent être considérées avec une grande vigilance : par exemple, lorsqu'il s'agit de surfaces, les valeurs prises en compte peuvent être des estimations de synthèse ou des mesures post placement / routage. Or, on constate bien souvent que les estimations faites lors de la phase de synthèse sont bien en deçà des valeurs réelles (d'un facteur 2 parfois). Il faut donc avoir ces chiffres en tête pour tirer les conclusions qui s'imposent. De plus, les hypothèses réalisées impliquent un niveau d'abstraction relativement élevé et donc, ces métriques intrinsèques doivent être vues comme un ensemble d'outils permettant de délimiter l'espace des solutions architecturales, et donc de déterminer un sous-ensemble d'architectures en adéquation avec un domaine d'applications. Une exploration

architecturale plus fine devra alors être réalisée sur le sous-ensemble des solutions ainsi obtenu.

Dans la mesure où ces métriques fournissent un critère d'efficacité suivant le volume de données, on peut très bien envisager de les utiliser afin de répartir les tâches sur un système hybride, basé sur des éléments reconfigurables de granularités différentes.

Enfin, ces métriques peuvent servir à paramétrer des architectures génériques. Grâce à un espace de caractérisation basé sur les cinq axes correspondant à nos critères, on peut estimer la scalabilité suivant les paramètres génériques. Nous avons illustré cette méthode par un exemple basé sur deux modèles abstraits. Pour son illustration sur un cas réel, une étude a été réalisée sur l'architecture Systolic Ring dont nous présentons les résultats dans le paragraphe suivant.

3.3.2. Le flot de conception du Systolic Ring

Le Systolic Ring est un modèle d'architecture qui repose sur un certain nombre de principes que nous avons présentés dans le chapitre 2. Ce modèle d'architecture est paramétrable, et ce qui nous intéresse ici, c'est de pouvoir estimer les performances de l'architecture suivant n'importe quel jeu de paramètres, en somme de réaliser une exploration architecturale du modèle Systolic Ring.

FIG. 73 - *Layout de l'architecture ayant servi à calibrer les surfaces du modèle générique*

Afin de prévoir l'évolution de la surface notamment, nous avons caractérisé plusieurs versions de l'anneau grâce aux outils de synthèse et placement/routage de Cadence (Exemple du Systolic Ring placé routé figure 73). Nous avons pu ainsi formaliser la surface en fonction des paramètres de l'architecture. Nous avons ainsi constaté que l'interconnexion basée sur des *switch* et le réseau de rétro-propagation impliquait une croissance quadratique de la surface des *switch* [Beno02c], ce qui impliquait une diminution rapide de la densité opérative (figure 74). Cette observation nous a amené à revoir la hiérarchie d'interconnexion [Beno03d] de cette architecture, en introduisant tout d'abord un troisième niveau basé sur l'interconnexion de plusieurs anneaux (paramètre S), permettant d'accroître le nombre d'opérateurs sans trop pénaliser la surface, i.e. de maintenir un D_{OP} relativement élevé, comme nous pouvons l'observer sur la figure 74.

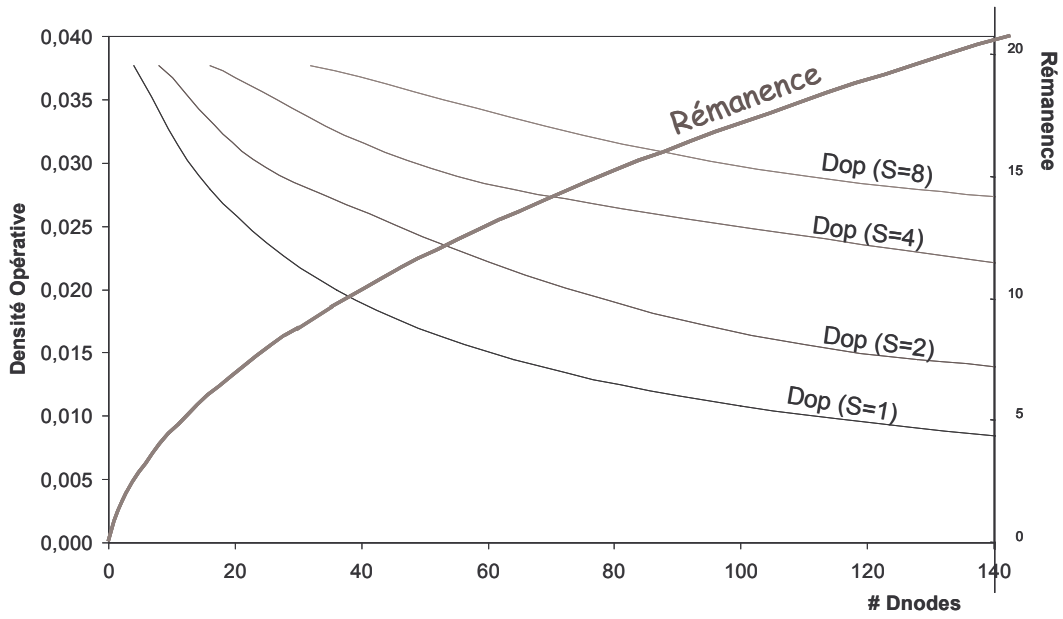


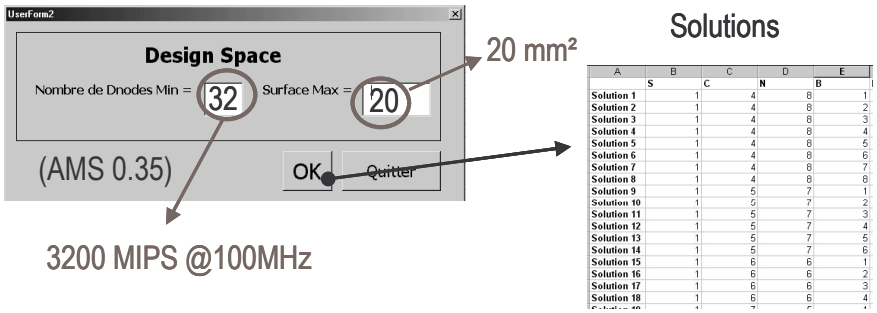
FIG. 74 - *Caractérisation de la densité opérative en fonction du nombre de Dnodes de la première version du Systolic Ring. La décroissance importante de D_{OP}, pour une version à un seul anneau, nous a motivés dans la définition d'un troisième niveau de hiérarchie d'interconnexion. L'introduction de ce niveau supplémentaire réduit le taux d'interconnectivité mais permet cependant d'augmenter la densité opérative.*

Pour des raisons de faible scalabilité et de manque de flexibilité, le réseau de rétro-propagation a été abandonné au profit d'un réseau de bus paramétrable plus flexible [Beno03e] (cf. chapitre 2). La surface du *Switch* ne croît plus de manière quadratique avec le nombre d'opérateurs, comme c'était le cas avec le réseau de rétro-propagation (puisque la taille de celui-ci dépendait du nombre de Dnodes). La taille du switch croît de manière linéaire suivant le nombre d'opérateurs ce qui a largement contribué à améliorer la *scalabilité* du réseau d'interconnexion, le nombre de bus étant un paramètre générique indépendant du nombre de Dnodes. Cependant, il faut ajouter que le nombre de bus doit logiquement être choisi de manière à accroître la richesse du réseau d'interconnexion, et ceci, donc, en fonction du nombre d'opérateurs (à la différence près que l'utilisateur est libre de choisir ce nombre).

Le modèle d'architecture est basé sur quatre paramètres qui permettent de moduler le matériel suivant les besoins de l'utilisateur : le nombre de Dnodes (S, C, N) et le nombre de bus (B). L'équilibrage entre les paramètres S, C, N et B permet de fixer la hiérarchie et le taux d'interconnexion.

L'intérêt d'une architecture paramétrable apporte un degré de flexibilité mais le nombre de solutions possibles augmente les solutions architecturales dans l'espace de conception. Si on suppose que chacun de ces paramètres peut prendre des valeurs de 1 à 8, cela donne un nombre de combinaisons de paramètres de 4096. Imaginons que l'on souhaite tester chacune des combinaisons sur 4 applications par exemple, d'extraire les temps de traitement, la surface et la puissance consommée. En faisant l'hypothèse qu'il faille 10 minutes (vision optimiste) pour générer le circuit, le synthétiser, réaliser une simulation post-synthèse et récupérer les résultats, on obtient un temps total de 114 jours. C'est à ce niveau que se justifie une fois de plus l'utilisation des métriques intrinsèques, qui vont aider à générer un ensemble de solutions qui a priori permettent de couvrir la spécification.

Pour automatiser le processus, nous avons développé un outil d'exploration architecturale dont l'interface est représentée sur la figure 75. Cet outil intègre les résultats de caractérisation de la dernière version du modèle d'architecture, et il permet d'extraire les 4 paramètres suivant les contraintes de l'utilisateur. De plus, pour chaque solution, les valeurs des métriques permettent de comparer les différents compromis obtenus.



	S	C	N	B
Solution 1	1	4	8	1
Solution 2	1	4	8	2
Solution 3	1	4	8	3
Solution 4	1	4	8	4
Solution 5	1	4	8	5
Solution 6	1	4	8	6
Solution 7	1	4	8	7
Solution 8	1	4	8	8
Solution 9	1	5	7	1
Solution 10	1	5	7	2
Solution 11	1	5	7	3
Solution 12	1	5	7	4
Solution 13	1	5	7	5
Solution 14	1	5	7	6
Solution 15	1	6	6	1
Solution 16	1	6	6	2
Solution 17	1	6	6	3
Solution 18	1	6	6	4

FIG. 75 - *Outil d'exploration architecturale du Systolic Ring. Cet outil permet de recevoir en entrée des contraintes telles que le nombre minimum de Dnodes et la surface maximale autorisée. D'après les caractérisations que nous avons réalisées en technologie CMOS AMS 0,35µ, nous pouvons extraire les paramètres de l'architecture qui vérifient les contraintes, ainsi que les valeurs des métriques afin de comparer les différents compromis.*

Afin de rendre exploitable ces résultats, nous avons développé toute une chaîne d'outils entièrement paramétrables. La figure 76 illustre le flot de conception et fait apparaître les outils dont nous disposons à l'heure actuelle. Après exploration de l'espace et extraction des paramètres initiaux du modèle, nous utilisons un outil de génération automatique de code VHDL. Pour programmer l'architecture ainsi obtenue, nous utilisons un assembleur paramétrable par lequel nous réalisons une projection manuelle des algorithmes à implanter.

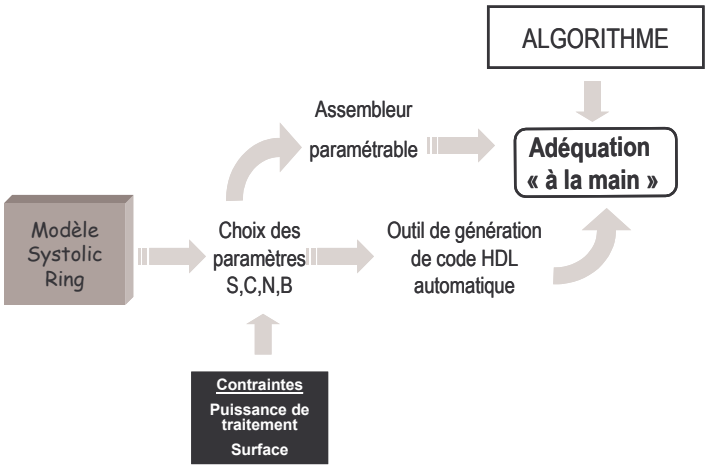


FIG. 76 - *Flot de conception du Systolic Ring. Les points d'entrée sont le modèle d'architecture, les contraintes de la spécification et les algorithmes à implanter. Par rapport aux spécifications du système, l'utilisateur va entrer par exemple la puissance de traitement souhaitée et la surface maximale autorisée pour le cœur à intégrer. Ensuite, un logiciel permet de générer automatiquement le code VHDL suivant les paramètres choisis. L'assembleur est lui aussi paramétré, et il permet de faire une projection manuelle des algorithmes.*

4. CONCLUSION

Face à la multiplication des accélérateurs matériels disponibles et dans l'objectif de comparer et caractériser ces différents cœurs de coprocesseur, nous avons proposé, à partir d'un certain nombre d'hypothèses, un modèle général formalisant les différentes spécificités de chaque approche. L'intérêt de cette méthodologie est souligné par la définition de métriques s'appuyant sur ce modèle et reflétant les caractéristiques essentielles prises en compte lors de la conception d'un système complet. La caractérisation de trois accélérateurs par ces métriques a permis une analyse intrinsèque de l'efficacité des compromis de performances dans notre domaine d'applications. Enfin, dans l'optique de comparer des modèles d'architecture mais aussi d'analyser leur *scalabilité*, nous avons proposé une étude de deux modèles abstraits, reconfigurables à grain épais, basés sur deux systèmes d'interconnexion différents. Appliquée à un cas réel, cette méthode s'intègre dans le flot de conception au niveau de l'exploration architecturale comme cela a été illustré pour le Systolic Ring.

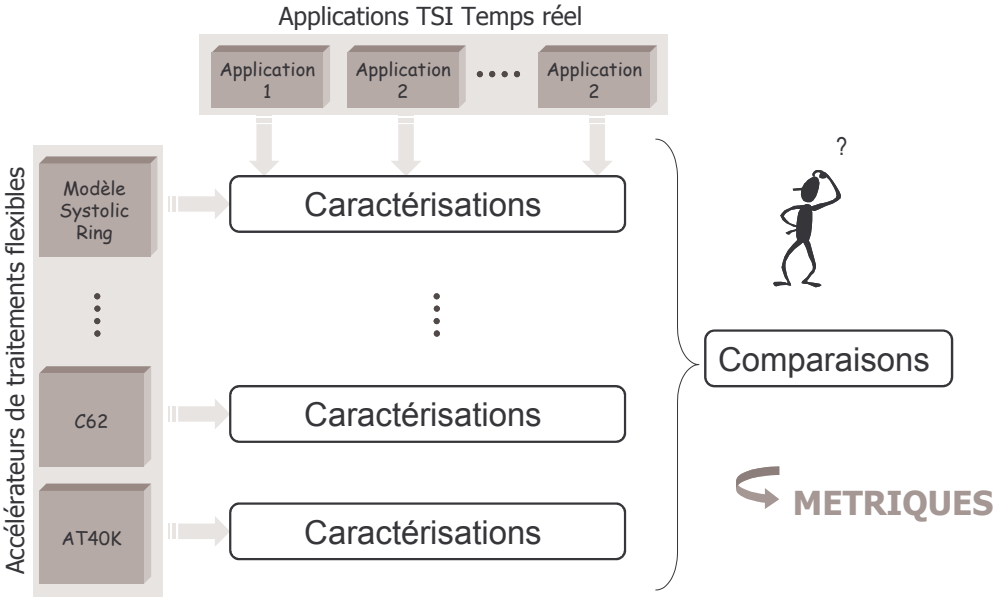


FIG. 77 - *La problématique de caractérisation et de comparaison traitée par les métriques intrinsèques matérielles*

Nos métriques permettent la caractérisation des architectures et ont pour vocation de s'intégrer au plus tôt dans le flot de conception d'un SoC, afin de comparer plusieurs architectures ou modèles génériques existants. L'évaluation des compromis de performances passe par une analyse des propriétés architecturales, et des caractéristiques des motifs de calculs ciblés. Il est tout à fait envisageable de relier ces travaux aux outils logiciels existants pour la caractérisation des applications (figure 77). Ainsi dans un flot de conception complet (figure 78), on peut imaginer dans un premier temps une analyse logicielle, suivie d'une caractérisation matérielle des différents candidats permettant d'extraire un sous-ensemble de solutions. Enfin, une évaluation plus fine du compromis de performances pourra alors être réalisée sur les candidats restant, à partir de métriques extrinsèques, afin de choisir la solution la plus en adéquation par rapport aux spécifications du système.

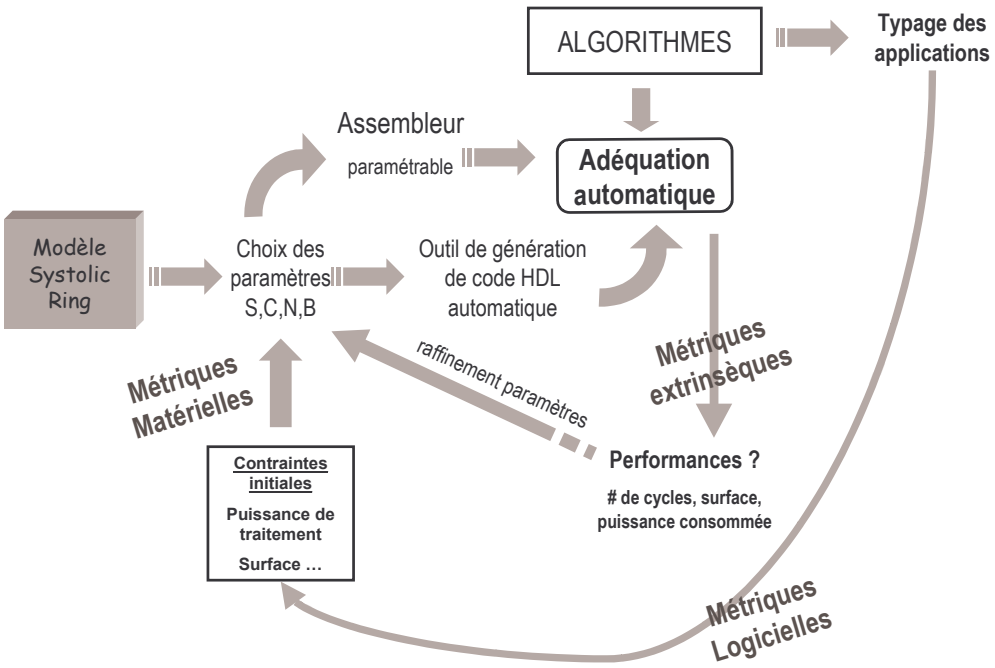


FIG. 78 - *Flot de conception du Systolic Ring intégrant les trois niveaux de caractérisation (logiciel, matériel et extrinsèque)*

De plus, il est important de souligner que ces métriques reposent parfois sur des hypothèses fortes qui pourraient s'avérer inexactes dans certains cas, notamment en ce qui concerne l'exploitation totale du parallélisme : même si le taux d'interconnexion pondère en quelque sorte l'étendue de cette hypothèse, il semblerait que plus les architectures disposent d'opérateurs, plus les outils (ou les utilisateurs) éprouvent des difficultés à utiliser toutes les ressources. Les faibles taux de remplissage des FPGA sont une illustration de cette tendance. De plus, lorsque les DSP et les architectures reconfigurables à grain épais disposent d'outils de compilation, ces derniers se révèlent relativement peu efficaces dans ce registre.

L'évolution des densités d'intégration devrait permettre des réalisations de circuits reconfigurables disposant de grandes quantités d'opérateurs. On est en droit de se demander, cependant, jusqu'où cette augmentation a un sens ? Existe-t-il une limite à l'exploitation du parallélisme ? Ce sentiment est renforcé par certaines publications [Lemo03] qui montrent que la plupart des algorithmes du TSI ne disposent pas d'un parallélisme moyen intra-fonction très élevé (inférieur à 8). Même si en partitionnant l'application, on peut obtenir un parallélisme très élevé localement, cela suggère que l'exploitation du parallélisme ne se situe pas forcément dans les opérations, mais peut-être à un niveau plus élevé.

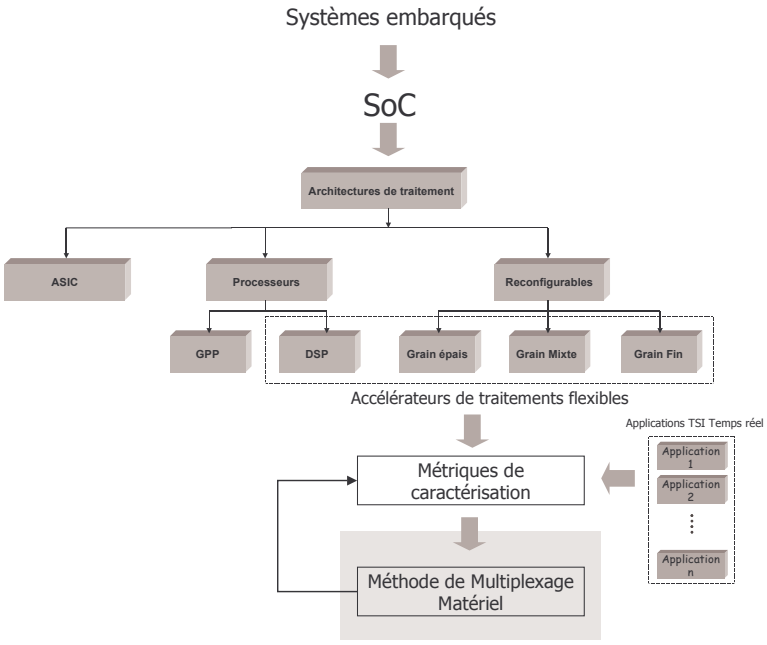
Face à cette problématique, nous proposons une solution permettant de tirer parti du parallélisme entre tâches et ainsi d'améliorer les performances globales de l'accélérateur en autorisant une utilisation plus efficace des ressources par un principe de multiplexage matériel. Ce dernier est rendu possible grâce aux propriétés de reconfiguration dynamique. C'est ce que nous présentons dans le chapitre 4.

Chapitre 4

EXPLOITATION DYNAMIQUE DU PARALLELISME

INTRODUCTION

Les conclusions du chapitre 3 concernant la comparaison des différentes familles d'accélérateurs indiquent que l'exploitation de la puissance de traitement passe par la mise en œuvre de solides méthodes, permettant de tirer parti d'un maximum de parallélisme. Le succès des architectures reconfigurables à grain épais dépendra de la disponibilité d'outils logiciels et/ou matériels allant dans le sens de l'exploitation de la totalité du potentiel opératif. C'est pour tenter de répondre à cette problématique nous proposerons dans ce chapitre un concept original de multiplexage matériel, utilisant les capacités de reconfiguration dynamique de ces architectures. Nous illustrerons sa mise en œuvre pratique par l'architecture Saturne et nous caractériserons les nouvelles performances par le biais des métriques du chapitre précédent.



SOMMAIRE DU CHAPITRE

1. PARALLELISME ET RECONFIGURATION DYNAMIQUE

Nous avons montré dans le chapitre précédent le potentiel accélérateur des architectures reconfigurables à grain épais dans un contexte d'applications TSI pour SOC. Grâce à un ensemble de métriques, nous avons illustré les avantages et les limitations de ces architectures. Parmi les nombreuses aptitudes des architectures reconfigurables à grain épais, l'une d'entre elles nous intéresse tout particulièrement : la reconfiguration dynamique. Nous pensons que c'est par cette caractéristique que ces architectures peuvent largement contribuer à améliorer les performances des systèmes. Or, même si de nombreux travaux de recherche se sont attachés à définir des méthodes permettant d'exploiter efficacement la reconfiguration dynamique (un ensemble de ces méthodes est résumé dans [Boud04]), il n'existe pas, à notre connaissance, de méthodes d'utilisation dynamique des ressources pour améliorer l'exploitation du parallélisme, en particulier pour les architectures reconfigurables à grain épais. C'est donc dans cette optique que nous nous proposons ici de discuter de son intérêt pour de telles architectures.

1.1. Les atouts de la reconfiguration dynamique

1.1.1. Reconfiguration dynamique ≠ Reconfiguration systématique

La reconfiguration dynamique consiste à modifier la fonctionnalité d'une architecture, *i.e.* la fonction des opérateurs et le chemin entre opérateurs, sans mettre le circuit hors tension. Pour la mettre en œuvre, le composant doit disposer de ressources matérielles permettant une écriture de la mémoire de configuration à tout instant : il s'agit en général d'un contrôleur jouant le rôle d'interface entre une mémoire qui stocke le programme de configuration à charger et la mémoire de configuration de l'architecture. Pour que la reconfiguration dynamique soit possible, il faut que le temps passé à reconfigurer soit relativement faible afin de pouvoir rapidement changer l'application.

D'un certain point de vue, les architectures reconfigurables, notamment à grain épais, et les microprocesseurs possèdent des caractéristiques très proches. Cependant, une des différences fondamentales réside dans le procédé de reconfiguration. En effet, pour les processeurs, il s'agit d'une reconfiguration systématique à chaque cycle alors que pour une architecture reconfigurable dynamiquement, celle-ci n'est pas systématique.

L'intérêt majeur des architectures reconfigurables par rapport aux processeurs, c'est de pouvoir exploiter un parallélisme en deux dimensions. Celui-ci se base sur une projection spatiale des algorithmes, *i.e.* les opérations pouvant être réalisées durant le même cycle de calcul et les opérations pouvant être *pipelinées* (parallélisme temporel). Lorsqu'on peut exploiter cette spatialité, l'objectif consiste alors à figer la fonctionnalité afin de limiter le contrôle de la structure. Ce degré d'action n'est pas possible sur les architectures de processeur ce qui oblige à contrôler systématiquement les ressources matérielles. Les architectures reconfigurables dynamiquement disposent donc d'un contrôle leur permettant de modifier leur fonctionnalité mais de manière simplifiée grâce au modèle spatial.

Tab. 27. *Illustration de la consommation d'énergie pour un processeur MIPS et une architecture reconfigurable à grain épais pour une FFT 8 points (ST CMOS 0,13µ) [Lewi04]*

	Energie dissipée *(en nJ)*
Processeur	11.84
Systolic Ring	2.25

La simplification du contrôle n'est pas le seul intérêt de la reconfiguration dynamique. En effet, en figeant le contenu d'une partie de la mémoire de configuration pendant la durée du traitement, on réduit la consommation par rapport à une approche microprocesseur classique où le contrôle systématique implique une modification du contenu des registres d'instruction à chaque cycle. Les résultats du tableau 27 confirment cette tendance.

1.1.2. Avantages potentiels de la reconfiguration dynamique

La reconfiguration dynamique peut avoir de nombreux avantages, par exemple au niveau de la programmation de l'architecture ou bien encore dans l'exploitation des différents niveaux de parallélisme. C'est ce que nous allons voir dans cette section.

a. Faciliter la programmation

Le potentiel dynamique des architectures reconfigurables à grain épais peut amener à adopter deux stratégies pour l'allocation des ressources. Ces deux stratégies sont finalement proches de celles des processeurs programmables. Pour ces derniers, il existe deux manières de traiter ce problème :

- la solution VLIW, où l'allocation et l'ordonnancement des opérations sont réalisés de manière statique, *i.e.* dès la phase de compilation ou de programmation en assembleur
- la solution superscalaire, où l'allocation et l'ordonnancement sont effectués dynamiquement grâce au lancement simultané de plusieurs instructions (remarque : cette propriété induit des aptitudes au parallélisme de flot)

Dans les deux cas, processeurs VLIW et superscalaires sont des composants dynamiques, capables de s'adapter à différents traitement par simple modification du programme. Ils diffèrent principalement dans le principe d'utilisation des ressources qui peut être soit prédéterminée (VLIW), soit dynamique (superscalaire).

Dans tous les cas, un programme, littéralement « ce qui est écrit avant », est « statique ». Pour les architectures reconfigurables, le programme sert à décrire la fonctionnalité des opérateurs et le chemin de données. En revanche, le degré de flexibilité autorisé par ce type de composant pose le problème de la stratégie d'allocation/ordonnancement : ces derniers doivent-ils être identiques à ceux énoncés dans le programme, à l'image des VLIW, ou bien modifiés durant le traitement, à l'image des superscalaires, suivant les besoins du système ?

Il est *a priori* possible pour les architectures reconfigurables à grain épais d'envisager chacune de ces deux possibilités, mais dans quelle mesure ? Pour des applications du TSI, il est généralement admis que celles-ci nécessitent peu de contrôle. Par conséquent, l'exécution des programmes est quasi-déterministe. C'est d'ailleurs une des raisons majeures de la différence entre les VLIW et les superscalaires. Le déterminisme des applications ciblées conduit à une méthode d'allocation/ordonnancement statique alors que l'incertitude relative liée aux applications dominées par le contrôle oriente vers une méthode dynamique. L'objectif étant d'utiliser les architectures reconfigurables à grain épais comme accélérateur d'applications du type TSI, l'approche statique serait donc, peut-être la plus appropriée… Cependant, dans la mesure où les technologies autorisent une augmentation croissante du nombre de ressources de calcul, et que le parallélisme intra-fonction est limité, quelle peut être la valeur ajoutée de ce type de composant ?

b. Exploitation efficace des différents niveaux de parallélisme

Ainsi que nous l'avons évoquée, l'exploitation du parallélisme est possible de l'opération de base, à l'intérieur même des fonctions, jusqu'au parallélisme inter-processus au niveau du système. A quel(s) niveau(x) les architectures reconfigurables à grain épais peuvent-elles intervenir, dans une logique d'accélération ?

Dans la littérature, le parallélisme intra-fonction est la cible de prédilection. Un programme de configuration définit une allocation bien définie des ressources correspondant au motif de calcul à implanter. La philosophie de programmation est proche des VLIW avec un degré de parallélisme temporel supplémentaire. On recense également quelques évocations un peu plus ambitieuses où il est question de parallélisme inter-processus, mais celles-ci restent très conceptuelles et aucun résultat n'a été publié quant au principe de mise en œuvre et à l'amélioration des performances.

Dans ce domaine, nous pouvons citer Jbits [Sing01], une approche dans l'utilisation des propriétés dynamiques des FPGA Xilinx. Cette plate-forme java permet d'exploiter la reconfiguration dynamique partielle des FPGA Xilinx, mais aussi autorise l'allocation dynamique des ressources. Ainsi, l'exploitation du parallélisme de tâche est rendue possible. Quelques contributions récentes proposent l'implantation de contrôleurs de configuration pour FPGA. C'est le cas notamment de [Curd03] et [Blod03] qui mettent en œuvre un contrôle par logiciel pour les FPGA Virtex-II Pro et Virtex-II (implantation processeur), ou bien encore de [Carv04] où il s'agit d'un contrôleur matériel pour Virtex-II.

L'allocation dynamique peut intervenir à un autre niveau plus fin de l'exploitation du parallélisme : il s'agit des boucles déterministes. Dans les processeurs classiques, le déroulage de boucle est une technique très courante pour améliorer les performances. Une possibilité attractive de l'allocation dynamique pourrait être le déroulage dynamique de boucle. En effet, en considérant un déroulage minimal permettant la satisfaction des contraintes, on peut très bien envisager de dérouler plus en profondeur si les ressources matérielles le permettent.

Notre objectif ici est donc d'étudier dans le cadre des architectures reconfigurables à grain épais la mise en œuvre de telles techniques. L'objectif est d'analyser l'intérêt de l'exploitation dynamique des ressources pour plus de flexibilité et de performance. Le support de notre étude sera l'architecture du Systolic Ring, mais cela pourrait être n'importe quel autre type d'architecture reconfigurable à grain épais.

1.2. Le multiplexage matériel

Pour illustrer le problème que nous souhaitons résoudre, prenons l'exemple représenté par la figure 79. L'accélérateur est une architecture reconfigurable à grain épais dotée d'un contrôleur qui fait l'interface entre la mémoire programme et la mémoire de configuration. C'est ce dernier qui permet de reconfigurer dynamiquement les opérateurs et le chemin de données. En guise d'exemple pédagogique, nous limitons la configuration à la fonctionnalité et à la topologie des opérateurs. Chaque processus chargé dans la mémoire programme contient sous forme de codes binaires, la configuration de chacun des opérateurs. Ensuite, il y a deux possibilités qui découlent des aptitudes du contrôleur : dans un cas, il faudra attendre que le premier processus soit terminé pour lancer le second, dans l'autre, on cherche les ressources libres qui permettent d'implémenter le processus suivant. Ainsi, en observant l'état des ressources, le contrôleur peut configurer la tâche suivante, dès qu'il le peut.

Les avantages de l'allocation dynamique sont multiples. Tout d'abord, dans un contexte d'intégration SOC à base d'OS multitâches, la prise en charge de multiples applications critiques simultanément n'est pas forcément marginale. Ainsi, par exemple, pour un appareil mobile, on peut imaginer une partie de l'accélérateur s'occupant des traitements multimédias

pendant que l'autre s'occupe de l'accès au réseau. Le multi-flots simultané permet d'optimiser l'utilisation des ressources. La tendance actuelle à l'augmentation des densités d'intégration pousse les concepteurs d'architectures reconfigurables à ajouter de plus en plus de ressources de calcul, afin d'augmenter la puissance de traitement. Cet accroissement ne peut pas être pensé sans une structure adéquate pour optimiser leur utilisation. De plus, lorsqu'on observe dans le détail les applications, le parallélisme au niveau des opérations n'est pas très élevé. Autrement dit, vu les quantités de données qu'il y a à traiter, il faut définir une méthode pour tirer parti du parallélisme à un niveau plus élevé. En permettant un placement dynamique des tâches, l'accélérateur matériel devient une plate-forme plus performante et plus flexible. En effet, la souplesse engendrée par l'allocation dynamique devrait simplifier la phase de portage des algorithmes.

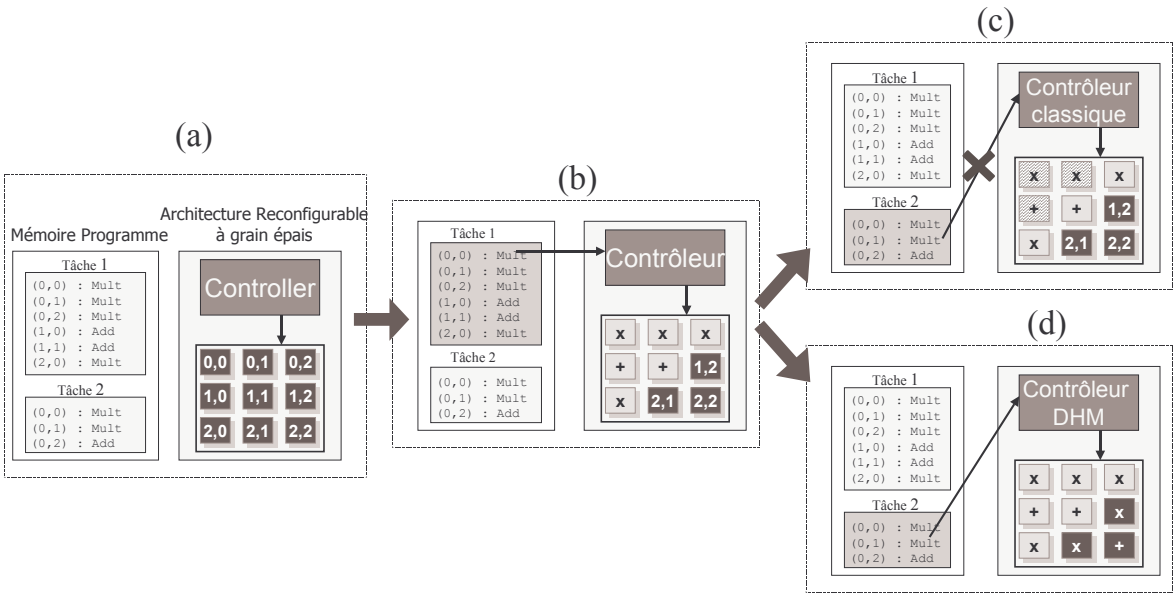


FIG. 79 - *Gestion statique ou dynamique. On suppose dans cet exemple une architecture reconfigurable à grain épais constituée d'une matrice de 9 opérateurs repérés par leurs coordonnées (ligne, colonne) (a). L'accélérateur a chargé dans sa mémoire programme deux flots de traitement disjoints correspondant à deux requêtes du processeur hôte. Le contrôleur, jouant le rôle de l'interface entre la mémoire programme et les registres de configuration, récupère la première configuration et distribue à chaque nœud adressé sa fonction (b). Ensuite, deux scénarios sont possibles : le contrôleur est capable ou non d'allouer dynamiquement les ressources qui sont libres. Dans le premier cas (c), les adresses des opérateurs se recoupent, et donc le contrôleur va attendre que le premier traitement soit terminé. Dans le deuxième (d), le contrôleur va chercher parmi ses opérateurs ceux qui sont libres et qui peuvent prendre en charge le flot n°2.*

En revanche, ce type d'allocation dit « dynamique » nécessite le développement de structures matérielles adaptées. Il faut veiller à ce que le surcoût matériel en termes de latence, de surface et de consommation reste en accord avec les spécifications des systèmes embarqués. C'est pour des raisons en contradiction avec certains de ces critères que les processeurs superscalaires figurent rarement dans ce type d'appareils. Le développement d'une structure d'allocation dynamique devra donc respecter un cahier des charges que nous allons développé par la suite.

1.3. Hypothèses et cahier des charges

Même si le multiplexage matériel s'inspire de méthodes utilisées dans les processeurs superscalaires, il est clair que la spécificité des architectures reconfigurables à grain épais impose la définition de contraintes adaptées. Nous allons notamment analyser ici le type et le nombre de tâches pouvant être prises en charge, ainsi que le codage des informations pour le multiplexage matériel et la gestion de la bande passante avec la mémoire de données. Nous formulerons ainsi un certain nombre d'hypothèses.

1.3.1. Type et nombre de tâches

En tout premier lieu, les caractéristiques de ces tâches seront choisies afin de satisfaire les propriétés du modèle de calcul défini dans le chapitre 3. Il s'agira donc de tâches type « flot de données », sans contrôle et un « grand » nombre de données à traiter à travers un certain nombre d'opérations implémentées en parallèle. On suppose que l'accélérateur ne prend en charge qu'une ou plusieurs sous-parties de ou des applications (consommant le plus de temps CPU), sans se préoccuper pour le moment de ce que fait le processeur au même instant,.

Hypothèse 1 : on suppose que l'OS gère dynamiquement l'exécution des processus et donc, qu'il suspend les processus qui attendent des données issues de l'accélérateur.

Il s'agit d'une file d'attente de tâches correspondant à des sous-processus disjoints. Il n'y a pas de contraintes de précédence. C'est le Système d'exploitation qui, en amont et en aval, se charge de vérifier la cohérence chronologique des processus.

Hypothèse 2 : la mémoire programme de l'accélérateur matériel, de son point de vue, ne contient que des processus indépendants.

La mémoire programme de l'architecture reconfigurable va donc être chargée indéfiniment, de la mise sous tension jusqu'à l'arrêt du système. Afin de simplifier la gestion de cette mémoire, l'utilisation d'un *buffer circulaire* semble s'accorder avec la stratégie d'allocation dynamique que nous souhaitons implémenter sur l'architecture reconfigurable à grain épais. Ainsi, le contrôleur aura la vision d'une profondeur mémoire infinie, ce qui permettra un chargement continu des tâches. C'est ensuite au contrôleur, de gérer suivant l'état de ses ressources, la configuration dynamique de ses éléments.

1.3.2. Codage de l'information

Une des contraintes de mise en œuvre consiste à savoir si le matériel prendra en charge la totalité de l'allocation dynamique ou bien si c'est le logiciel qui permettra de simplifier le rôle du matériel.

Dans une approche SOC, l'utilisation de matériel supplémentaire est toujours un inconvénient pour des raisons de coût et de consommation. Nous pensons donc qu'il est nécessaire de définir des structures permettant la prise en charge de certaines parties de l'allocation dynamique sans pour autant définir une architecture trop complexe qui impliquerait une surface de silicium trop conséquente. Pour cela, nous allons définir un certain nombre d'éléments qui permettront, au sein même du logiciel, de faciliter la tâche du contrôleur et donc, de simplifier le matériel.

Hypothèse 3 : chaque programme devra subir une compilation spécifique à notre cible ce qui empêchera nécessairement la compatibilité binaire des exécutables.

Ceci implique donc forcément une modification du code exécutable initial. La partie dédiée à l'accélérateur sera alors considérée comme le code de configuration initial, ou

configuration virtuelle. C'est à partir de cette configuration virtuelle que le multiplexeur matériel va opérer des modifications afin d'adapter le motif de calcul initial aux ressources libres au moment du chargement de cette configuration initiale.

Pour transformer le motif, chaque processus se verra ajouter un en-tête de configuration permettant de caractériser la tâche à allouer. Les champs de cet en-tête concerneront, par exemple, le nombre de ressources, la topologie de ces ressources, l'ordre de priorité des tâches, les contraintes de consommation, le facteur de déroulage etc.

a. Nombre de ressources

Afin de savoir s'il est possible de projeter une tâche pré-compilée sur la cible, la première chose à analyser, c'est si le nombre de ressources disponibles est suffisant. Ces ressources peuvent être de types différents : opérateurs, générateurs d'adresse, registres, interconnexions...

Hypothèse 4 : un champ doit permettre de spécifier le nombre de ressources nécessaires pour implanter le motif, pour chaque type de ressource dont dispose l'architecture.

b. Masque de configuration

Une fois que le matériel a pu vérifier qu'il disposait du nombre de ressources suffisant pour allouer la tâche à l'accélérateur, il va passer à une analyse topologique. En effet, suivant l'état actuel des ressources, il n'est pas forcément possible d'allouer le motif algorithmique initial dans l'espace libre.

La première étape va donc consister à chercher une solution au placement du motif dans l'accélérateur. Suivant les ambitions des capacités matérielles de l'allocation dynamique, cette étape sera plus ou moins simple et donc plus ou moins performante. Une solution simple de placement consiste à utiliser un masque de configuration qui correspond au motif à implanter.

Hypothèse 5 : on utilisera un masque de configuration qui sera déplacé sur la grille d'opérateurs et comparé successivement à l'état des ressources. Dès qu'il y a compatibilité, on configure, sinon la tâche est mise en attente.

Cette hypothèse sur la méthode de placement ne parcourt pas l'espace des solutions de manière exhaustive : elle aura donc tendance à laisser des espaces libres inutilisés par les motifs. Il est tout à fait possible d'imaginer des procédés plus complexes, qui placeraient un à un les opérateurs dans les espaces libres et qui utiliseraient les connexions globales pour connecter les opérateurs. Si la solution n'est pas satisfaisante, nous reviendrons peut-être sur cette hypothèse.

c. Priorités

La possibilité de configuration ASAP est très intéressante mais doit également prendre en compte les contraintes de temps réel. Dans ce cas, lorsque le système est chargé d'exécuter un processus avant la limite de l'échéancier, l'accélérateur doit être capable de réagir rapidement afin de ne pas pénaliser le reste du système mis en attente par sa faute.

La réactivité de l'accélérateur doit être particulièrement efficace, dans le sens où les cycles de recherche pour des solutions de placement ne doivent pas devenir trop importants devant les contraintes fixées. Une solution consiste à maintenir l'exécution des tâches en cours et si une solution de placement existe, tout se déroule comme s'il ne s'agissait pas d'un processus à priorité temps réel. Sinon, on sauvegarde le contexte des tâches en cours et on exécute directement le processus temps réel. Une fois celui-ci arrivé à terme, l'accélérateur recharge le contexte sauvegardé. La deuxième solution consiste à faire l'hypothèse que le

procédé de recherche de solution de placement est trop gourmand en temps et donc, à sauvegarder directement le contexte, sans se préoccuper s'il est possible ou non de faire du multiplexage matériel. Cette deuxième solution permet de satisfaire les contraintes de temps réel, mais malheureusement pénalise les accélérations en cours. Sur une architecture dotée de nombreux opérateurs, comme on peut l'imaginer dans le futur, ceci impliquerait une sous-utilisation des ressources et c'est exactement ce que l'on cherche à éviter.

Hypothèse 5 : pour satisfaire les contraintes de temps réel, il faut prévoir des mécanismes de sauvegarde de contexte, i.e. que l'on doit être capable de mémoriser l'état de l'accélérateur à n'importe quel instant.

La sauvegarde de contexte étant une solution de recours, il faut que le mécanisme dédié à l'allocation dynamique des ressources soit très simple et performant ce qui nécessite donc de développer une structure matérielle dédiée aux traitements à mettre en œuvre sur le calcul du nombre de ressources et l'analyse topologique.

1.3.3. Partage de la mémoire

Ce problème est au moins aussi important que ceux évoqués jusqu'à maintenant. En effet, impossible d'imaginer un multiplexage matériel de plusieurs configurations qui nécessiteraient une bande passante supérieure à celle présente sur le matériel.

Hypothèse 6 : on suppose que l'architecture dispose d'une batterie de générateurs d'adresses, capables de se partager la bande passante simultanément (grâce à des banques mémoires par exemple)

Pour mettre en œuvre le multiplexage matériel, il suffit donc de vérifier que le nombre de générateurs d'adresse nécessaire est suffisant, lors du décompte des ressources. Par exemple, imaginons qu'un processus nécessite l'injection de deux données simultanément qu'il y ait 2 générateurs d'adresse en cours d'utilisation et 2 de libres, il est donc matériellement possible d'accéder à ces données par ces générateurs d'adresse.

Cette partie, qui nécessite un travail plus spécifique, est en cours d'élaboration, et un sujet de thèse a même été commencé en octobre 2003. Ces travaux ne seront pas exposés dans la présentation de l'architecture Saturne et donc les générateurs d'adresses seront modélisés sous forme de compteurs d'échantillons.

2. LE MULTIPLEXEUR MATERIEL SATURNE

Le multiplexage matériel est une méthode générique pouvant être adaptée à n'importe quelle architecture reconfigurable à grain épais. Cependant, l'homogénéité des opérateurs, la topologie, la hiérarchie et le type de ressources d'interconnexion peuvent influencer la manière de concevoir l'unité de multiplexage matériel.

Afin de valider notre concept, nous avons utilisé l'architecture du Systolic Ring pour laquelle nous avons développé un contrôleur de configuration dont le rôle est de prendre en charge le multiplexage matériel de l'anneau de processeurs : ce nouveau système a été baptisé Saturne. Afin que ce multiplexeur matériel soit facilement portable sur n'importe quel type d'accélérateur, nous l'avons développé sans modifier la structure initiale du Systolic Ring. Nous présentons cette architecture dans le détail dans les prochaines sections.

2.1. Fonctionnement général

Pour commencer, nous allons tout d'abord présenter le fonctionnement général de ce contrôleur dédié au multiplexage matériel. Nous présenterons tout d'abord l'heuristique d'allocation dynamique puis nous exposerons la structure de l'entête. Enfin, nous donnerons une illustration globale du système basé sur l'unité de multiplexage matériel. La spécification générale de l'heuristique d'allocation dynamique a été, dans un premier temps, implantée en C, et a permis de valider théoriquement notre méthode de multiplexage matériel.

2.1.1. Heuristique d'allocation dynamique

La formulation du fonctionnement de l'unité d'allocation dynamique peut être illustrée par l'algorithme de la figure 80. A notre connaissance, aucune heuristique similaire n'a été jusqu'alors formulée dans la littérature pour des architectures reconfigurables à grain épais. Celle-ci a été dans un premier temps réalisée en C afin de valider nos objectifs. Cette première étape nous a permis de revenir plusieurs fois sur l'enchaînement des différentes étapes afin de couvrir tous les cas que nous souhaitions prendre en charge.

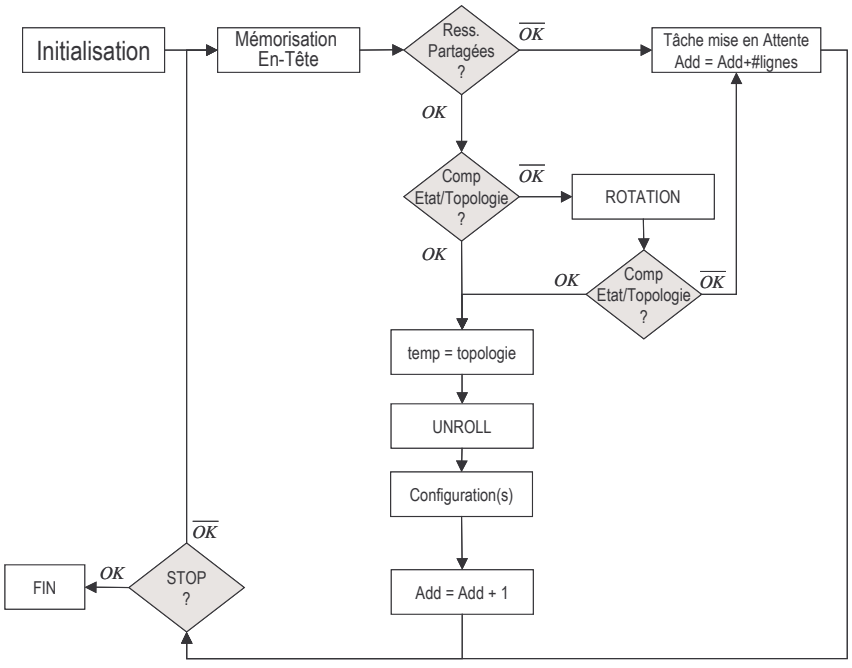


FIG. 80 - *Heuristique d'allocation dynamique.*

Le déroulement de l'heuristique est constitué de plusieurs phases. Après initialisation du système, l'unité d'allocation rentre dans une exécution en boucle qui commence par la mémorisation d'un entête de configuration. La partie de l'entête correspondant aux ressources nécessaires est comparée à l'état des ressources matérielles. Si le nombre de ressources nécessaire est trop grand, cette tâche est mise en attente, sinon, les topologies sont comparées. Si la compatibilité est vérifiée, *i.e.* le motif peut être alloué, on peut passer à la phase de déroulage de boucle. Dans cette phase, l'unité va voir combien de fois il est possible de déployer le motif suivant les ressources libres. Pour l'instant, ce déroulage est systématique et sans contrainte. Il est clair qu'il tend à favoriser le parallélisme spatial. Nous envisageons de rapidement proposer une méthode de gestion du parallélisme de boucle directement liée au niveau de priorité de chaque tâche. La configuration initiale est alors configurée autant de fois qu'elle a été déroulée et placée dans les espaces libres de l'accélérateur. Après la phase de configuration ou de mise en attente, l'adresse de l'entête suivant est générée par le compteur de programme, incrémenté de la valeur *#lignes* figurant dans l'entête de configuration.

2.1.2. Structure de l'entête

Comme nous l'avons évoqué dans la première section de ce chapitre, nous utiliserons un entête de configuration attribué à chaque tâche. Il va jouer le rôle d'interface entre les instructions de configuration de la tâche concernée et le contrôleur de multiplexage matériel. Le format de l'entête manipulé par Saturne est illustré figure 81. Un premier champ renseigne le nombre de lignes de configuration, ce qui permet au compteur du programme de sauter à la tâche suivante si les conditions de configuration ne sont pas remplies. Ces conditions sont en tout premier lieu le nombre de ressources. La version du Systolic Ring utilisée possède 8 opérateurs, 3 bus et 4 générateurs d'adresse d'où le nombre de bits utilisés pour coder le nombre de ressources à utiliser. Ensuite, la topologie Bus, Opérateurs et Générateurs d'adresse sont des vecteurs binaires dont les coordonnées correspondent au numéro de la ressource et l'état '0' ou '1' correspond à l'utilisation de la ressource ('1') ou pas ('0').

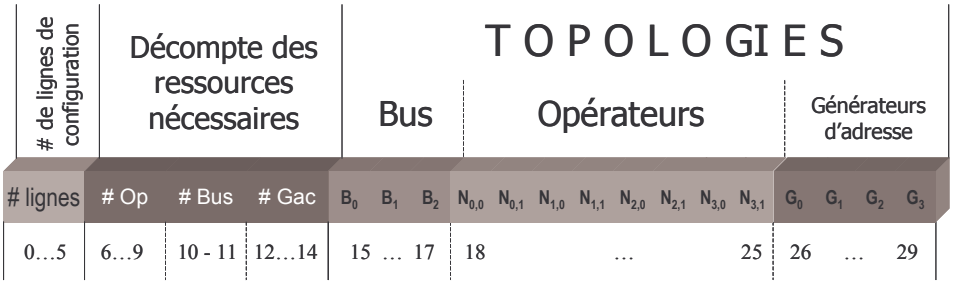


FIG. 81 - *Structure de l'entête utilisé pour le multiplexage matériel dans le Systolic Ring.*

Les différents champs de cet entête correspondent aux caractéristiques du motif algorithmique à implanter suivant les propriétés de l'accélérateur matériel, ici le Systolic Ring. Pour comprendre la manière dont un en-tête est généré, prenons l'exemple d'une tâche réalisant le calcul d'une identité remarquable. La figure 82 illustre les différentes étapes du processus de génération de l'en-tête final.

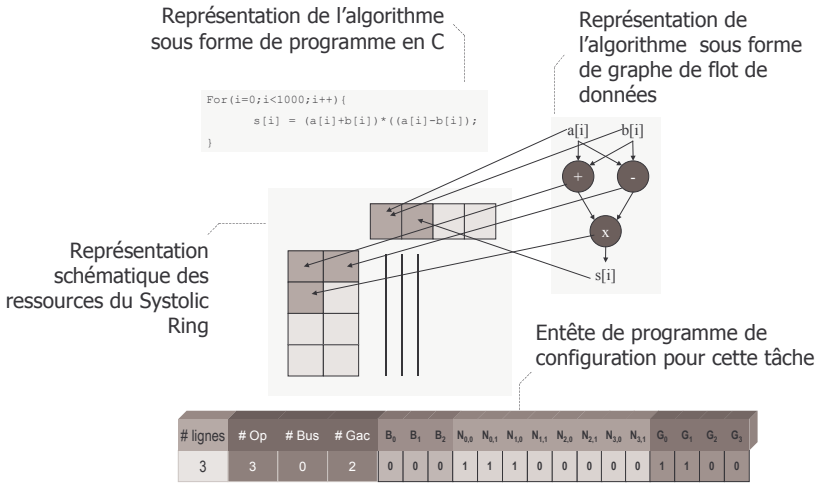


FIG. 82 - *Calcul d'une identité remarquable et en-tête de configuration correspondant. Cette figure illustre le flot partant de la description de l'algorithme sous forme de programme, en passant par un graphe projeté sur les ressources matérielles du Systolic Ring. C'est à partir du portage du motif que l'on tire les informations utiles à l'allocation de la tâche : ici, le nombre de ressources, la topologie et le nombre de lignes de configurations.*

2.1.3. Vue globale du système

Pour concevoir le multiplexeur matériel Saturne, deux possibilités classiques étaient envisageables :

- Solution logicielle, laissant des degrés d'adaptation et d'évolutivité plus grands de l'heuristique initiale, mais avec des performances difficilement maîtrisables, notamment concernant la latence introduite.
- Solution matérielle dédiée, permettant d'optimiser les performances en termes de cycles de latence, la surface et la consommation.

Notre choix s'est porté sur la deuxième solution pour des raisons essentiellement liées à la latence de configuration, et donc à la réactivité du système. Quitte à perdre en cycles de développement ou de mise à jour, nous pensons à l'heure actuelle que l'effort essentiel à apporter à ce niveau là se situe dans le rapport coût(surface) / performances. Il serait cependant tout à fait intéressant d'étudier une réalisation logicielle de multiplexage matériel. Nous reprendrons cette idée dans les perspectives finales.

Le Système Saturne est composé du Systolic Ring est d'une unité de multiplexage matériel comme cela est illustré sur la figure 83. Cette architecture matérielle a été conçue en VHDL. Son fonctionnement global est basé sur l'heuristique présentée auparavant. Pour sa mise en œuvre, nous avons opté pour un système composé des deux parties illustrées sur la figure 83 :

- la première est constituée d'un certain nombre de registres et de logique combinatoire dédiés à stocker l'état des ressources et à les comparer aux différents champs de l'en-tête,
- la deuxième est une Machine à Etats Finis, qui permet de gérer tous les signaux d'activations des différents éléments, correspondant à un séquencement prédéfini des étapes d'allocation dynamique.

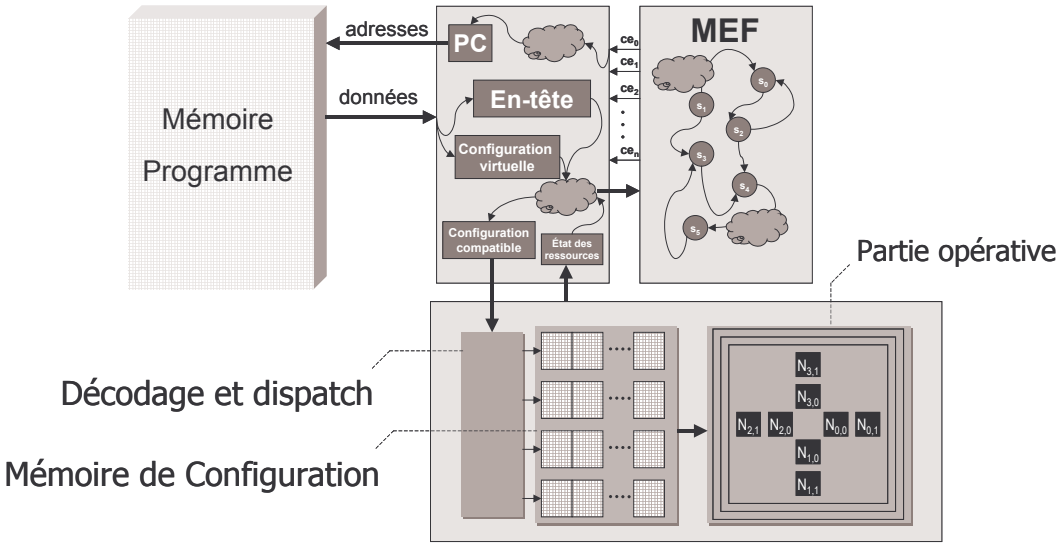


FIG. 83 - *Vue globale du système d'allocation dynamique Saturne, constitué de la mémoire programme et de l'architecture de l'accélérateur reconfigurable ciblé. La partie de gauche du système Saturne récupère les entêtes ou les codes configuration virtuels, en générant des adresses à partir du PC. Suivant les comparaisons faites avec l'état des ressources, la MEF active ou désactive les parties considérées. Ainsi, elle peut ordonner par exemple une rotation de configuration. Une fois qu'une solution compatible est générée, la configuration effective est chargée dans la mémoire de configuration de l'accélérateur qui se charge ensuite de décoder ces instructions de configuration. Pour finir, l'état des ressources est mis à jour.*

2.2. Architecture de Saturne

Le multiplexeur matériel Saturne est composé d'une partie traitement et d'une partie contrôle (MEF). La première est essentiellement composée d'opérateurs et de registres, la deuxième est composée de registres d'état et de la logique pour calculer l'état suivant. Nous présentons dans le détail cette architecture et son fonctionnement dans les paragraphes suivants.

2.2.1. Partie traitement

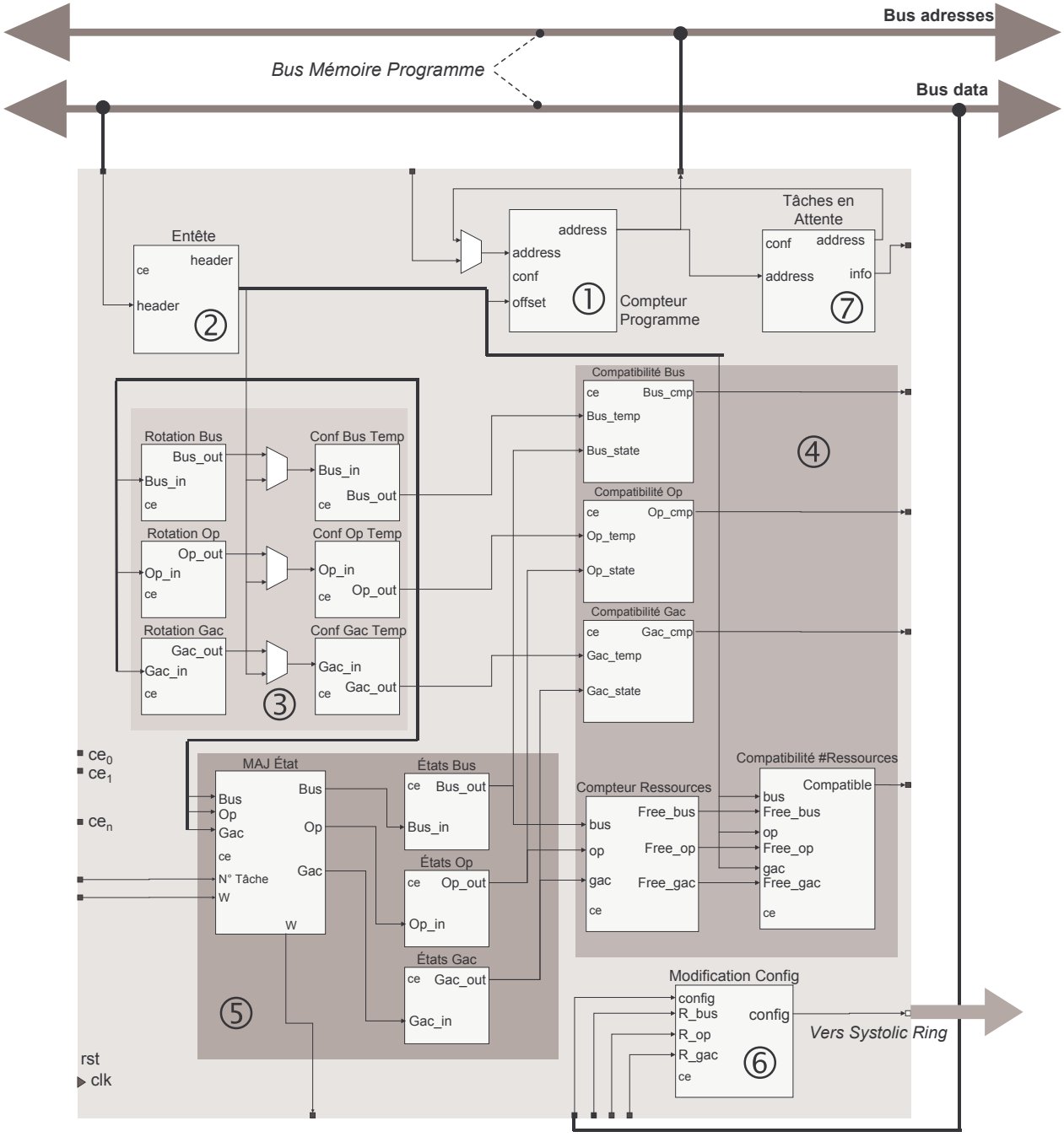
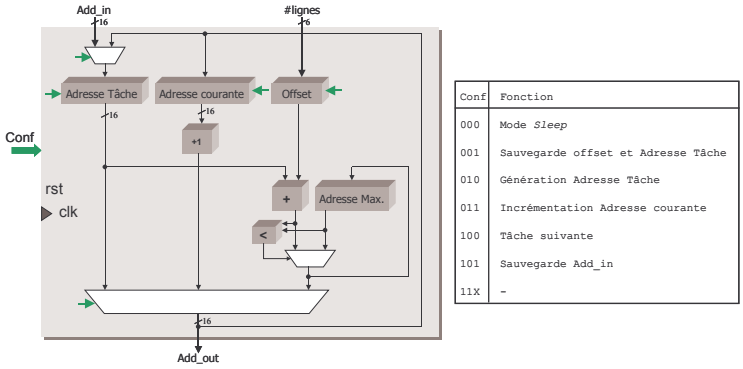


FIG. 84 - *Architecture de la partie traitement de l'unité de multiplexage matériel. Elle est composée de 7 composants essentiels à l'interfaçage entre la mémoire programme et l'architecture du Systolic Ring.*

La partie traitement est représentée dans sa globalité par la figure 84. Les principaux ports d'entrée/sortie sont les bus d'adresses et de données vers la mémoire programme, des signaux d'activation (ce_i), et le bus de configuration vers le Systolic Ring. Cette partie contient registres d'état du Systolic Ring, registres temporaires et opérateurs pour traiter les modifications de configuration. Elle est constituée de plusieurs blocs distincts dont le fonctionnement est détaillé dans les points suivants :

1. Compteur du programme :

C'est ce composant qui permet de générer les adresses de la mémoire programme, et qui va permettre, suivant les cas, de générer une adresse d'entête, ou une adresse de configuration de Systolic Ring. Son architecture est représentée sur la figure 85.



Conf	Fonction
000	Mode *Sleep*
001	Sauvegarde offset et Adresse Tâche
010	Génération Adresse Tâche
011	Incrémentation Adresse courante
100	Tâche suivante
101	Sauvegarde Add_in
11X	-

FIG. 85 - *Architecture du compteur de programme configurable. C'est grâce à ce module que sont gérées les adresses des tâches à lire en mémoire programme, et les adresses des mots de configuration destinés au Systolic Ring. Les différentes configurations possibles du compteur de programme sont explicitées dans le tableau joint à la figure.*

2. Registre d'entête

C'est dans ce registre 30 bits que l'entête d'une tâche va être mémorisé à chaque fois d'une adresse de tâche va être générée.

3. Registres tampons

C'est par leur biais que va être réalisé le stockage temporaire des masques de configuration et le résultat des opérations de rotations sur ces masques.

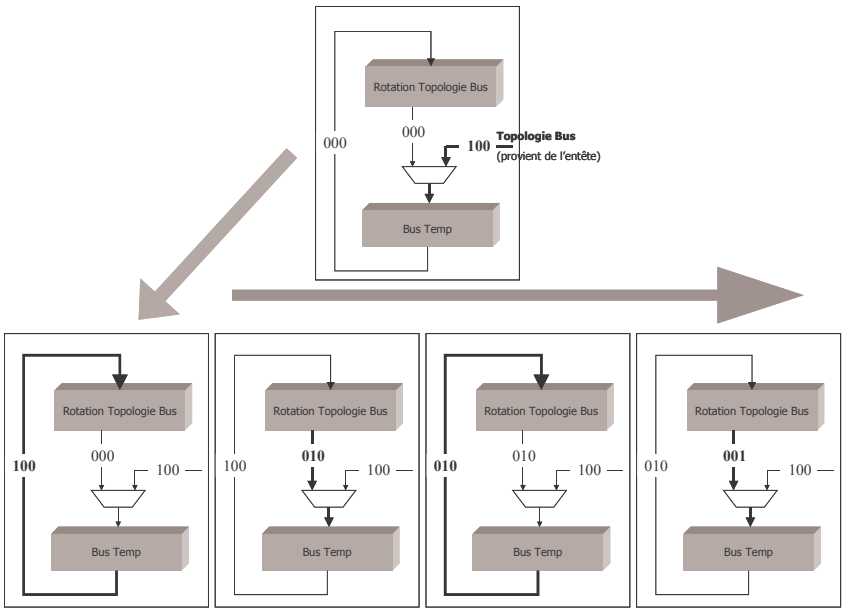


FIG. 86 - *Exemple de stockage / rotation topologique avec l'entête « topologie bus ». Dans un premier temps le champ « topologie bus » est mémorisé dans un registre tampon puis celui-ci est modifié de manière a assuré une rotation à droite de la topologie. Ce processus est réitéré au maximum deux fois si besoin est, i.e. s'il n'y a pas compatibilité topologique.*

4. Module de compatibilité

C'est par ce module que vont être analysées les compatibilités entre le nombre de ressources nécessaires et le nombre de ressources disponibles (figure 88), ainsi que la compatibilité entre topologie active et le masque de configuration(figure 87). Nous avons pris en exemple les Bus (3 bus donc 3 bits pour la topologie).

Pour le premier module, on fait simplement le « ET » des deux mots : si un seul bit est à '1', cela signifie que topologie active et entrante se recoupent. En faisant le « OU », et bien si le résultat est à '1', cela signifie qu'il y a au moins un bit à '1'.

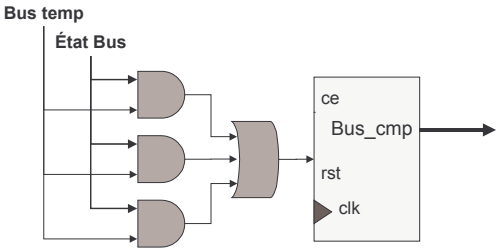


FIG. 87 - *Processus de comparaison de l'état courant et de la topologie entrante.*

Pour le deuxième, en fonction de l'état, un simple module permet d'extraire la quantité de ressources libres restantes et ensuite, si ce résultat est supérieur ou égal au champ de ressources nécessaires correspondant dans l'entête, le bit compatibilité est positionné sur '1'.

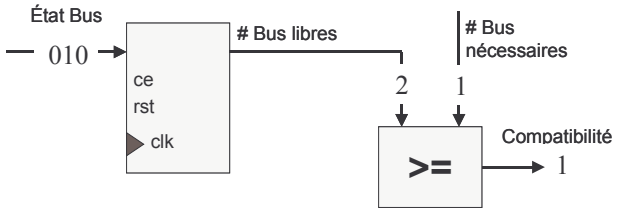


FIG. 88 - *Décompte des ressources libres et comparaison par rapport aux ressources nécessaires.*

5. Registres d'états ressources / tâches

Quatre tâches peuvent être prises en charge par le Systolic Ring : l'unité de multiplexage matériel dispose donc de quatre contextes d'états. Pour chacun de ces contextes, un registre contient l'état de la ressource : '1' si elle est utilisée dans ce contexte, '0' sinon. (figure 89).

Nous avons fixé le nombre maximal de tâches possibles à 4. Il y a donc 4 contextes possibles. Dans le cas des bus, cela représente donc 4 registres 3 bits. A l'initialisation (a), l'ensemble des registres est à '0'. Lorsqu'une tâche vérifie les conditions de compatibilité, alors on peut la configurer. Pour choisir le n° de tâche, le contrôleur va analyser les valeurs des w_{OUT}. Lorsqu'il en trouve un à '0', alors il attribue le n° correspondant à la tâche en cours. Ici par exemple il s'agit de la tâche '0'. La topologie est « 100 », c'est donc ce vecteur qui est mémorisé et qui permet de mettre à jour le bus de sortie d'état, qui est un « OU » de l'ensemble des registres(b). Une fois que la configuration est effective, le bit $w_{IN}(0)$ passe à 1(c). A la fin du traitement, ce même bit repasse à '0' ce qui déclenche un reset du registre d'état correspondant(d).

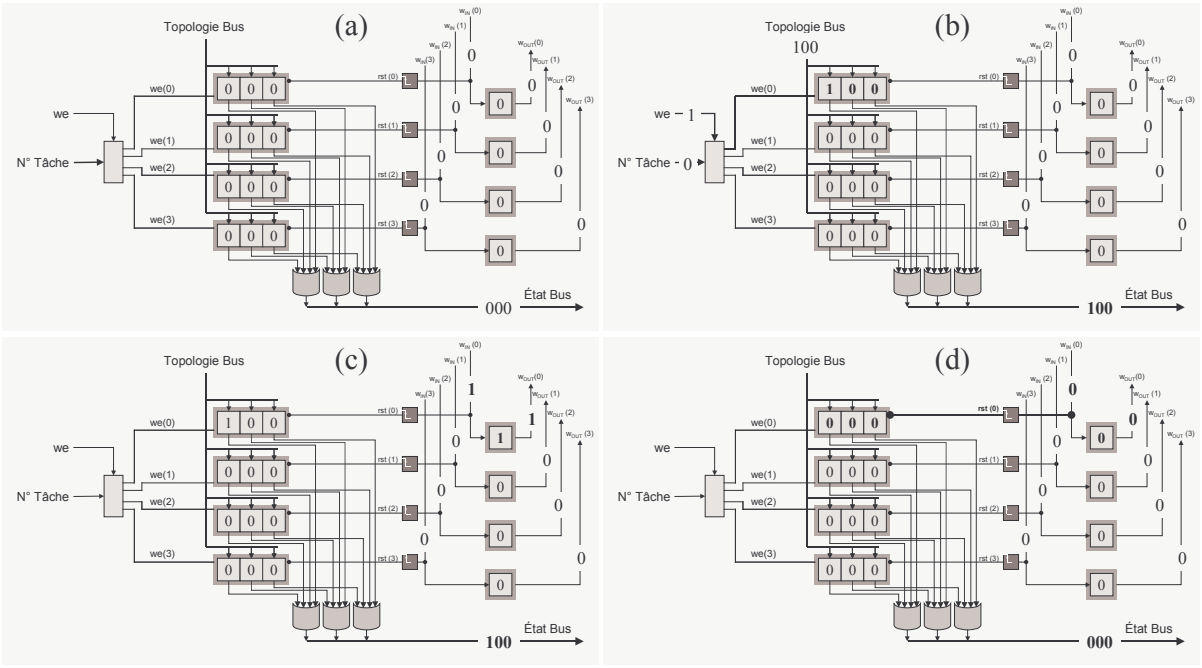


FIG. 89 - *Processus de mise à jour du registre d'état des différentes tâches.*

6. Registre de configuration

Ce registre est un peu particulier car il va permettre d'opérer les modifications finales suivant les rotations qui ont permis la compatibilité entre le masque et la topologie active. Il contient donc des registres et de la logique qui va permettre de permuter certains champs, en fonction des valeurs des rotations.

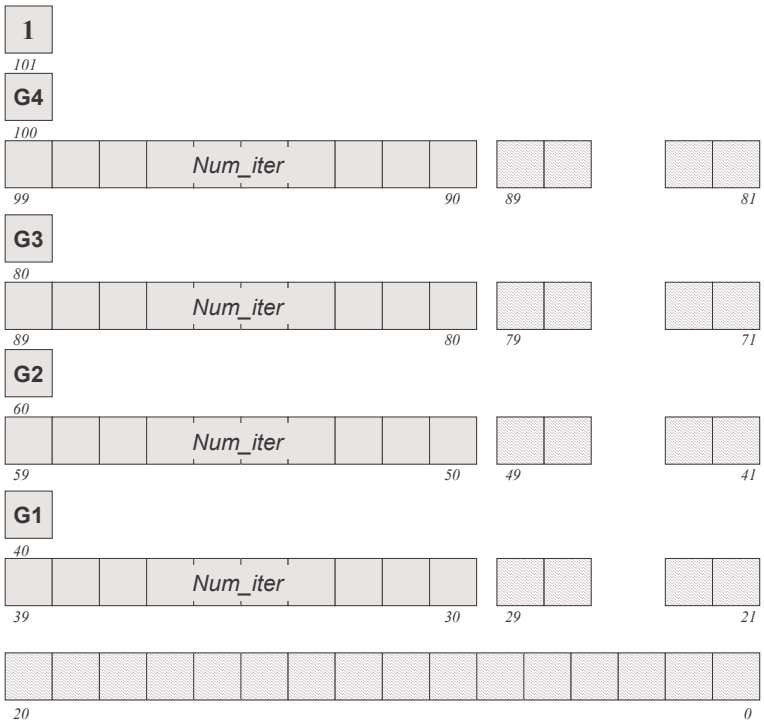


FIG. 90 - *Détail de l'instruction de configuration de type '1'. Il s'agit de l'instruction destinée aux générateurs d'adresse. Les différents champs de cette instruction permettent par le biais de bits de masquage, d'adresser, de 1 à 4 générateurs d'adresses configurables, qui, comme nous le verrons plus tard, ne sont en fait que de simples décompteurs. Ils sont configurés de*

telle sorte que le nombre d'itérations corresponde au nombre de cycles par tâche.

Les figures 90 et 91 représentent les 2 types d'instructions dédiées au Systolic Ring. Suivant le type, les opérations de rotation vont modifier différemment les micro-instructions. Par exemple, pour une instruction de type 1 (figure 90), si la valeur de rotation est 1, voilà les permutations qui sont réalisées :

- bits 20..0 : ← 20 .. 0
- bits 40 .. 21 ← 100 .. 81
- bits 60 .. 41 ← 40 .. 21
- bits 80 .. 61 ← 60 .. 41
- bits 100 .. 81 ← 80 .. 61

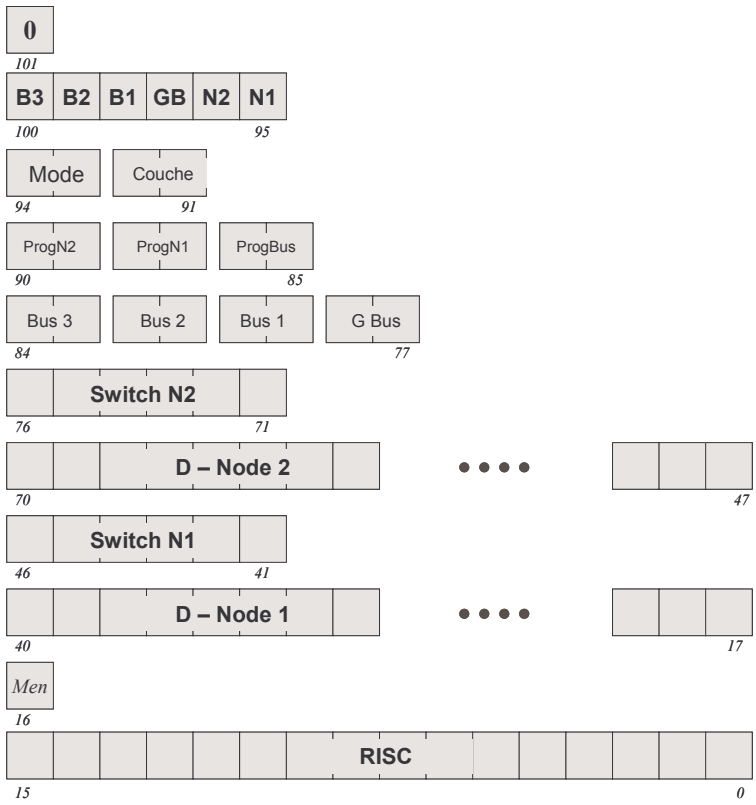


FIG. 91 - *Détail de l'instruction de configuration de type'0'. Il s'agit d'une instruction destinée à une couche donnée de l'architecture de traitement, pour laquelle on va déterminer la fonctionnalité des bus associés, Switch et Dnodes. Afin de considérer des tâches qui se partageraient les ressources d'une même couche, des bits de masquage ont été ajoutés au mot configuration initial.*

7. Pile des tâches en attente

Mise en attente de tâches non allouables. Il s'agit d'une machine d'état qui permet la gestion d'une pile d'adresses de tâches. Le système de contrôle de l'unité de multiplexage matériel et la pile des tâches en attente communiquent par le biais de drapeaux (info) et d'un mode de configuration (conf) (si on stocke une adresse (01), si on tente de configurer une tâche en attente (10), ou bien si cette dernière étape a été un succès pour rempiler les adresses

(11), et enfin, si on ne fait rien (00)). Les figures suivantes illustrent l'architecture (figure 92) et le fonctionnement de la machine d'état(figure 93).

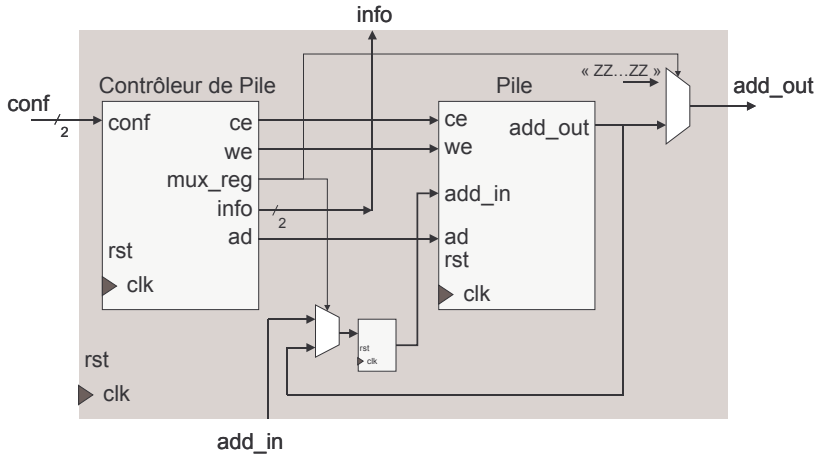


FIG. 92 - *Le module de mise en attente des tâches.*

Le module de mise en attente des tâches (figure 93) consiste en un contrôleur et deux registres qui permettent de stocker les adresses des tâches non configurables à l'instant considéré. Les signaux de lecture / écriture des registres sont contrôlés par une machine d'état représentée ci-après, elle-même dépendant d'une configuration générée par le contrôleur général. Cette configuration est notamment fonction du signal d'information qui indique l'état de la pile.

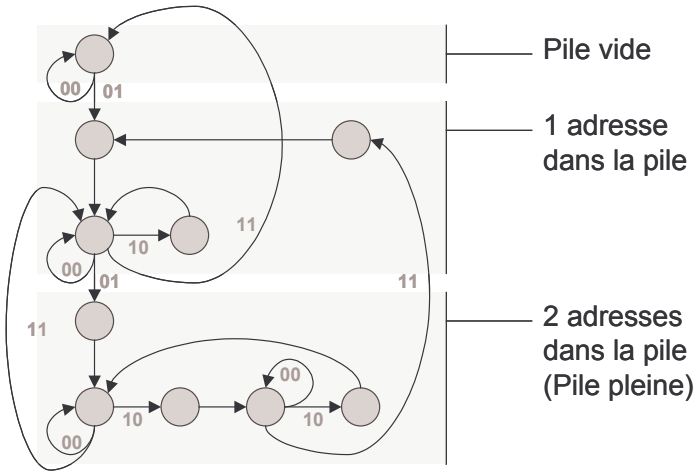


FIG. 93 - *Machine d'états qui permet de contrôler la pile de registres d'attente du système. Compte-tenu du nombre d'opérateurs, la pile contient uniquement 2 registres d'attente. Le signal « conf », sur 2 bits, correspond à la configuration générée par le contrôleur général.*

2.2.2. Partie contrôle

La partie contrôle de l'unité de multiplexage matériel permet d'activer ou désactiver les éléments de la partie traitement, en gérant principalement des signaux ce_i. Le déclenchement de ces signaux est effectué tout d'abord suivant un déroulement prédéfini des états système ainsi que de l'état dans lequel se trouve le système. La figure 94 illustre la vue externe de ce contrôleur, i.e. les signaux d'entrée et de sortie.

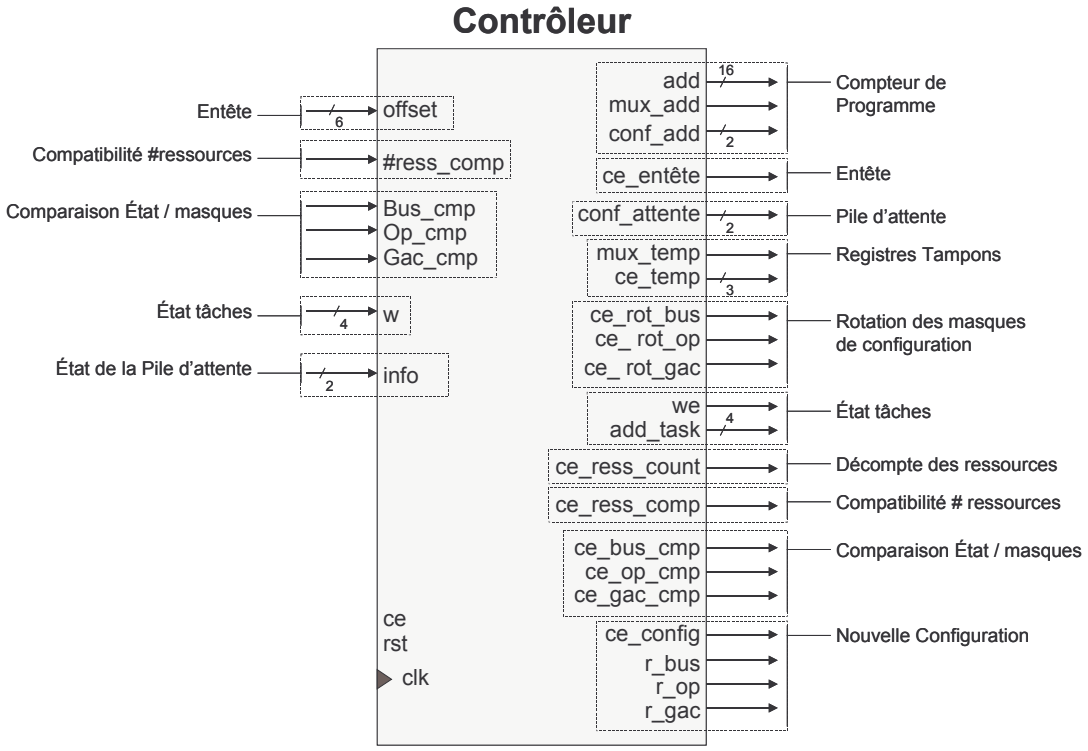


FIG. 94 - *Vue externe de la machine d'état qui permet le contrôle de l'allocation dynamique des tâches. A partir de l'état courant et de signaux d'entrée qui proviennent soit de l'entête (offset), soit de modules du système (x_cmp, info, #ress_comp, w), la machine d'état va calculer l'état suivant et les sorties correspondantes. Ces sorties permettent d'activer/désactiver les modules simples ou de générer des configurations pour les modules un peu plus complexes.*

La machine d'états dédiée au séquencement des différentes phases de l'heuristique d'allocation dynamique est représentée de manière détaillée figure 95. Elle est composée de 4 phases principales que nous présentons plus dans le détail par la suite.

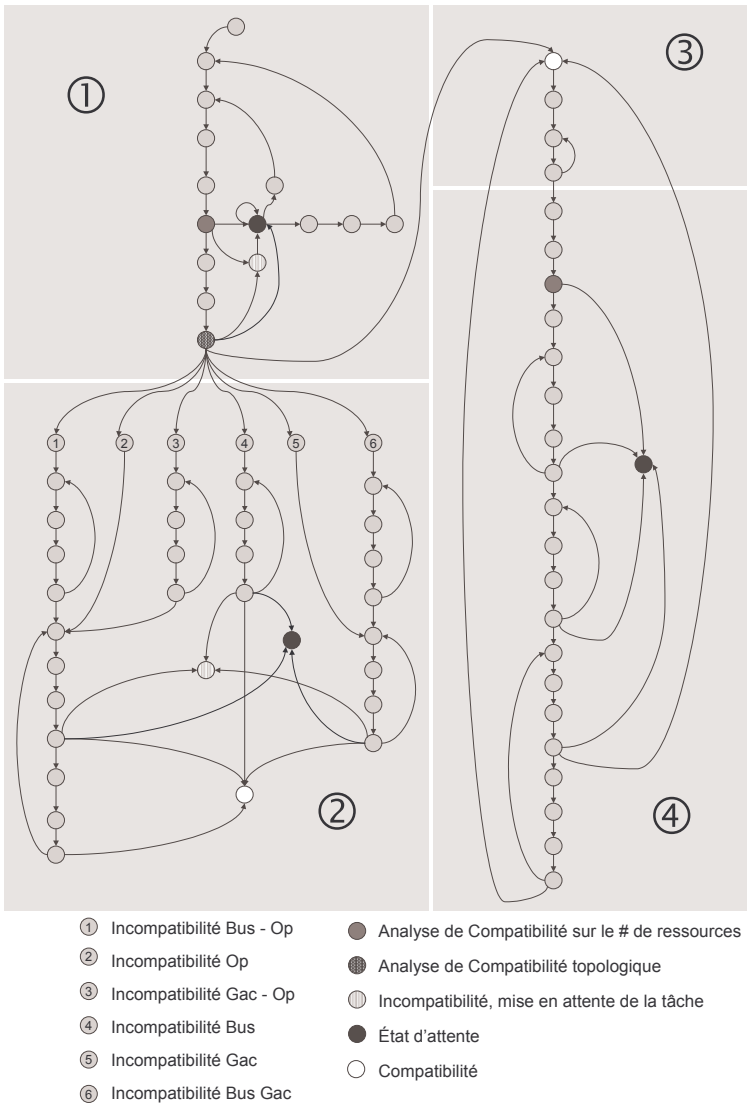


FIG. 95 - *Machine à états du contrôleur. Il s'agit d'une machine d'état composée initialement de 4 parties : la première est chargée principalement des accès à la mémoire programme et des analyses de compatibilité, la deuxième permet des transformations topologiques s'il y a des incompatibilités, la troisième gère les phases de configurations dynamiques et la quatrième permet de démultiplier le motif algorithmique pour dérouler les boucles.*

Une première partie concerne les initialisations, la lecture et mémorisation des entêtes, analyses de compatibilité et gestion des tâches allouables ou non. La deuxième permet après analyse de compatibilité de gérer les topologies non totalement incompatibles afin de les transformer pour tester d'autres solutions topologiques. Le troisième bloc concerne la gestion de la configuration. Enfin, une dernière partie permet de gérer le déroulage de boucles.

1. Initialisations et analyse de l'entête

Cette première phase va consister à récupérer l'entête à partir de la mémoire programme en générant une adresse grâce au compteur de programme, puis à analyser la compatibilité par rapport à l'état courant. Cette analyse est basée dans un premier temps sur le nombre de ressources, puis ensuite sur la compatibilité entre le masque de configuration et la topologie active. Dans les deux cas, si les tests ne sont pas positifs, alors deux possibilités se présentent : soit le contrôleur met la tâche en attente si la pile n'est pas pleine, soit il passe

dans un état d'attente. Lorsque le contrôleur est en état d'attente, il change d'état uniquement s'il y a moins de quatre tâches actives. Si c'est le cas, deux cas sont alors possibles : soit il utilise une adresse de la pile d'attente (pile non vide) soit il passe à l'adresse de la tâche suivante. Si les tests sont passés avec succès, on passe à la deuxième partie de la machine d'état.

2. Gestion des masques non-totalement incompatibles

Pour cette deuxième partie, un succès correspond au fait qu'il existe au moins une compatibilité topologique (soit sur les bus, soit sur les opérateurs ou les générateurs d'adresses). Le cas où il y a compatibilité totale correspond à la partie 3. Dans cette deuxième partie, il existe au moins un champ de ressources pour lequel il y a une incompatibilité. Cela signifie que pour trouver une solution, il va falloir opérer des modifications de configuration. Il existe 6 possibilités représentées sur la machine d'état. Pour chaque ressource (Bus, Opérateur, Gac) et pour chaque rotation, 4 états sont nécessaires pour opérer la transformation topologique et effectuer un test de compatibilité. Cette phase est bouclée et réitérée autant de fois que nécessaire, i.e. tant qu'il y a incompatibilité et que toutes les possibilités n'ont pas été testées. A noter toutefois que pour les opérateurs, il y a deux axes de rotation possible (dans la direction du pipeline et dans la direction perpendiculaire au pipeline) ce qui multiplie par 2 le nombre d'états pour les opérateurs. A l'issue de ces phases de transformation/test, trois possibilités se présentent : soit les modifications sont efficaces et permettent une compatibilité topologique, dans ce cas on passe à la phase 3, soit les tests sont négatifs et alors, le contrôleur passe l'adresse de la tâche dans la pile d'attente si elle n'est pas pleine, soit le contrôleur passe en état d'attente (retour dans la partie 1 de la machine d'états).

3. Phase de configuration

Cette partie correspond à la phase de configuration de l'architecture reconfigurable. Cela signifie qu'une compatibilité topologique a été trouvée. Le compteur de programme va alors générer les adresses mémoire qui correspondent aux instructions de configuration (i.e. de « `adresse_entête+1` » jusqu'à « `adresse_entête+nombre_lignes` ». Ces instructions sont dans un premier temps mémorisées, et on leur applique les modifications apportées au masque via le module « registre de configuration ».

4. Phase de déroulage dynamique de boucle

Dans cette dernière partie, on va appliquer des transformations de configuration comme dans la partie 2, afin de dupliquer le motif algorithmique, *i.e.* dérouler une boucle par exemple. Après une phase d'analyse de compatibilité au niveau du nombre de ressources, pour chaque ressource, quatre états permettent au système de transformer la topologie et tester la compatibilité du motif résultant de la modification. Ces quatre états sont réitérés autant de fois que nécessaires, i.e. tant que toutes les combinaisons de rotation n'ont pas été testées et qu'il n'y a pas de compatibilité. S'il n'y a incompatibilité, on passe en état d'attente, sinon on reconfigure (partie 3) avec les valeurs obtenues de rotations.

2.3. Environnement

Dans cette dernière partie de section, nous allons décrire l'environnement de l'unité de multiplexage matériel. Nous ne reviendrons pas sur le Systolic Ring que nous avons largement détaillé dans le chapitre 2 de ce mémoire, mais nous nous attacherons notamment à présenter le mécanisme de synchronisation des tâches.

2.3.1. Synchronisation des tâches

Une fois l'exécution d'un motif algorithmique lancée, l'arrêt de la configuration sera effectué lorsque toutes les données incidentes auront été traitées. A l'heure actuelle, le système d'adressage de la mémoire de données n'est pas figé et donc nous nous baserons pour notre méthode sur une modélisation de cet aspect du traitement. Afin de simplifier cela, nous utilisons de simples décompteurs d'échantillons (figure 96) qui sont programmés à l'aide d'une instruction de configuration. Cette instruction permet de savoir le nombre d'itérations de décompte. On fait l'hypothèse ici que tous les accès sont immédiats, *i.e.* qu'il n'y a pas de phase de suspension d'accès. Cela suppose donc que les *fifos* d'entrées, qui alimentent le Systolic Ring en données, sont « pleines » en permanence. C'est suivant cette contrainte que devra être conçu le système chargé de gérer la bande passante entre la mémoire de données et le Systolic Ring. Si cela n'est pas raisonnablement faisable, alors il suffira de suspendre ce décompteur à chaque fois que les données ne seront pas directement accessibles.

Lorsqu'un déroulage de boucle est effectué, on effectue un décalage à gauche sur le registre contenant le nombre d'itérations à réaliser. Ceci permet d'avoir un matériel très simple pour cela mais en revanche, n'autorise des divisions que par des puissances de 2.

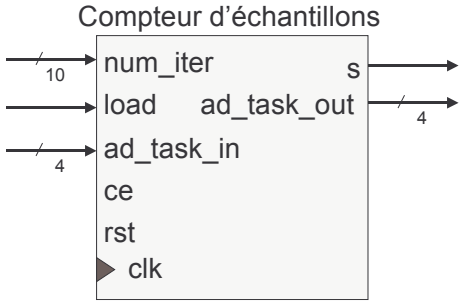


FIG. 96 - *Vue externe du compteur d'échantillons. Il permet de modéliser le fonctionnement des générateurs d'adresse qui seront mis en œuvre afin d'accéder aux données dans la mémoire de données. Il consiste pour le moment en un simple décompteur auquel on passe en phase de chargement un nombre d'itérations, et l'adresse de la tâche à laquelle il est attaché. La sortie est alors initialisée à '1'. Ensuite, la valeur de num_iter est décrémentée de 1 à chaque cycle d'horloge. Une fois que num_iter atteint '0', la sortie de ce module passe à '0', ce qui indique que le traitement est achevé.*

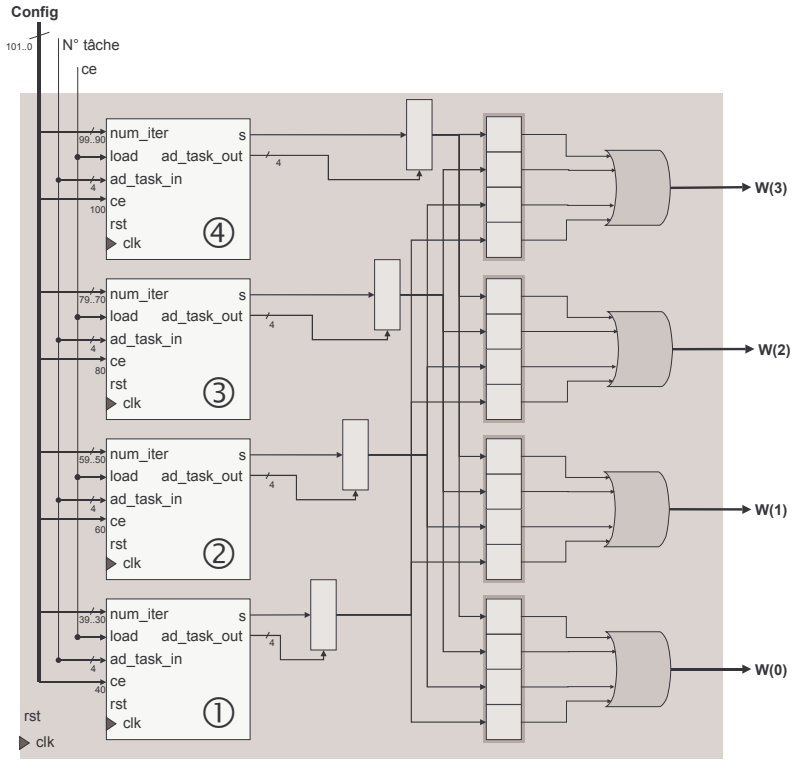


FIG. 97 - *Système permettant de contrôler l'état d'avancement des tâches. Chaque générateur d'adresse est relié via un décodeur à des registres représentant l'état de chaque générateur d'adresses pour une tâche donnée ('0' signifie inactif, '1' actif). Suivant le numéro de tâche auquel chaque générateur est relié, le signal d'état « s » est acheminé vers le registre correspondant. Grâce à un ou câblé en sortie de ces registres, on peut savoir si une tâche est active ('1') ou inactive en fonction de l'état des générateurs d'adresse.*

2.3.2. L'entité « Saturn Ring System »

L'architecture de Saturne est représentée par la figure 98. Elle permet notamment de voir comment les 3 entités principales de notre système, l'unité de multiplexage matériel, le Systolic Ring et le système de synchronisation des tâches, sont interconnectées.

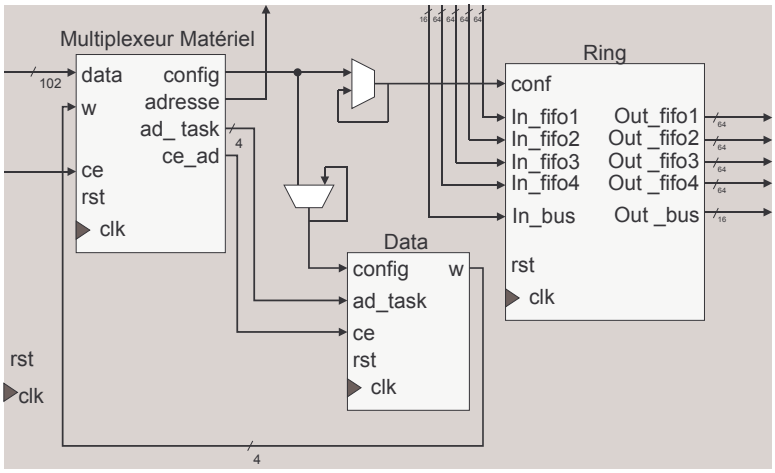


FIG. 98 - *Système complet avec le Multiplexeur Matériel, l'anneau de processeurs (Ring) et le système de compteur d'échantillons pour la mise à jour de l'état des tâches*

2.3.3. Outil de programmation

Un outil développé en C permet de programmer à la fois les entêtes utilisés par le multiplexeur matériel, le système de gestion des échantillons et l'anneau de processeurs Systolic Ring. Le niveau d'abstraction utilisé est le langage machine. Cela suppose que le programmeur connaisse l'architecture et sache par avance la manière dont il va implanter l'algorithme.

L'outil permet de générer un fichier de configuration binaire. De plus, un fichier « rapport » permet de vérifier a posteriori les valeurs saisies (instructions, valeurs décimales etc .) ce qui facilite le développement de nouvelles applications ou la correction d'erreurs.

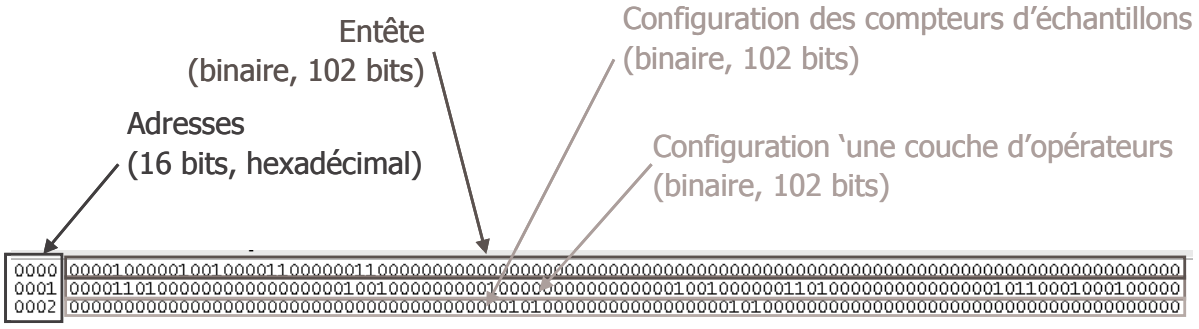


FIG. 99 - *Exemple de fichier de configuration généré par l'outil de programmation.*

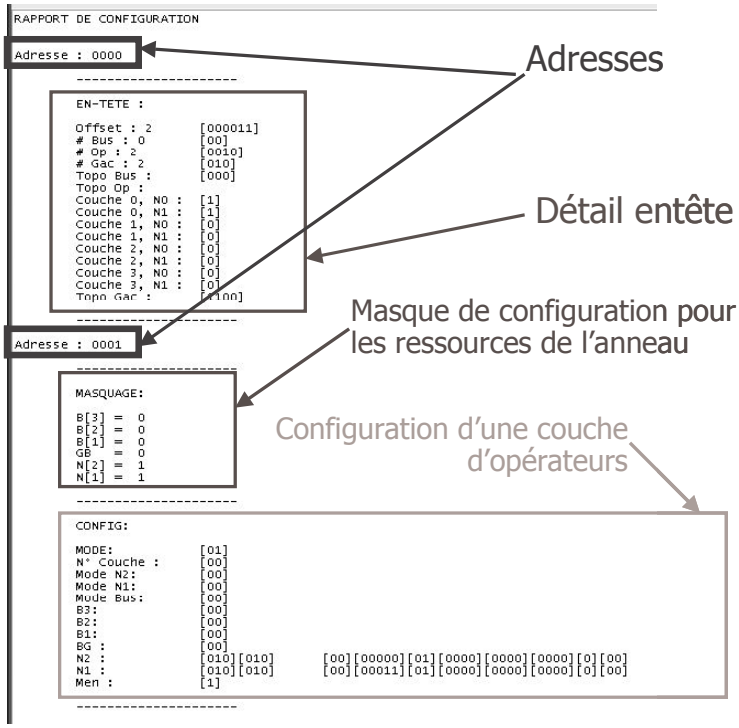


FIG. 100 - *Exemple de fichier « rapport » généré automatiquement après la phase de programmation. Il permet de vérifier les valeurs saisies et de corriger plus facilement un champ incorrect.*

3. VALIDATIONS

Cette troisième partie de chapitre résume les résultats que nous avons obtenus lors de la synthèse de notre composant, ainsi que quelques résultats de simulation permettant d'illustrer notre concept de multiplexage matériel.

3.1. Résultats de synthèse

Nous avons réalisé des synthèses à l'aide de l'outil de Ambit Build Gate de Cadence, en technologie CMOS ST 0.13µ. Ceci va nous permettre dans un premier temps d'estimer la surface silicium requise, mais surtout à nous positionner par rapport à l'ancien système de gestion des configurations, le processeur RISC.

L'estimation de la surface de l'unité de multiplexage matériel est de **39746 µm²**. Afin d'analyser les modules coûteux en surface, nous avons représenté un graphique permettant de voir la proportion de chaque élément par rapport à la surface totale. Ainsi, nous pouvons constater que le contrôleur occupe 28% de la surface. Le registre de configuration quant à lui occupe 20% et le compteur du programme 13%. Si l'on veut minimiser la surface de cette unité de l'architecture Saturne, il faudra donc apporter un effort particulier sur ces trois modules.

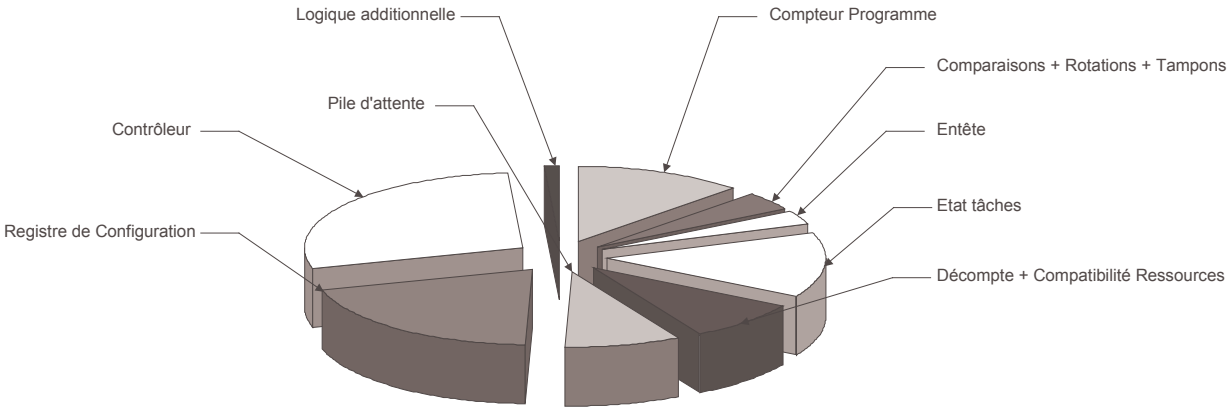


FIG. 101 - *Proportion des différents modules de l'unité de multiplexage matériel par rapport à la surface totale*

La figure suivante permet de comparer les surfaces des deux versions. La première, Systolic Ring, est basée sur l'anneau de 8 processeurs et le processeur RISC comme contrôleur de configuration. La deuxième, Saturne, utilise également l'anneau de 8 processeurs mais aussi l'unité de Multiplexage Matériel et le système de synchronisation des tâches. Nous pouvons constater que l'utilisation de l'unité de multiplexage matériel, outre les avantages qu'elle apporte, permet également de réduire la surface totale du système par rapport à l'utilisation du RISC comme contrôleur de configuration.

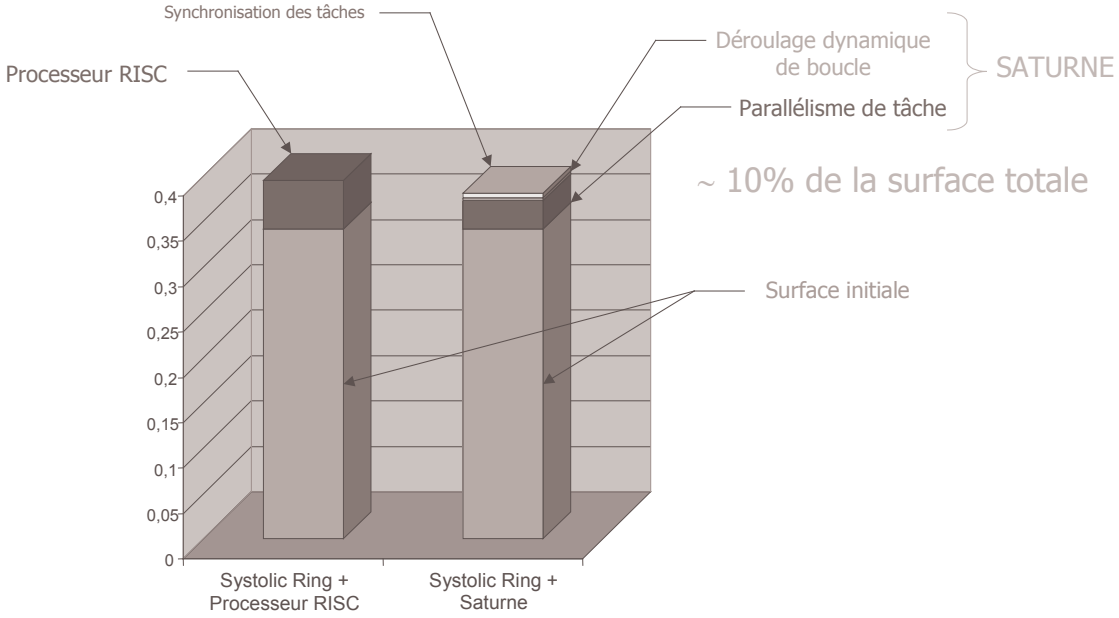


FIG. 102 - *Comparaison des surfaces de l'architecture Systolic Ring, basée sur le contrôleur de configuration RISC et Saturne, basée sur l'unité de Multiplexage Matériel*

La figure 103 permet d'observer la proportion de l'unité de multiplexage matériel par rapport à la surface totale de Saturne. Nous pouvons remarquer que l'organe de configuration ne représente que 10% de la surface totale du système. Les compteurs d'échantillons ne représentent quant à eux que 1% de la surface totale.

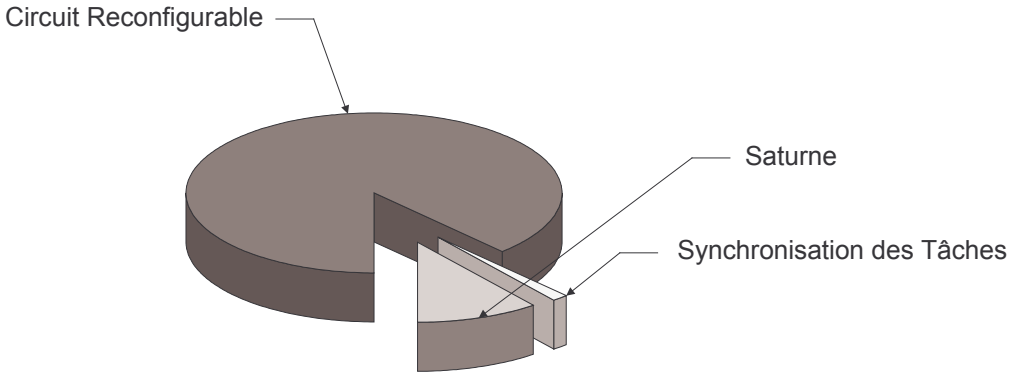


FIG. 103 - *La proportion des trois composants du Saturne par rapport à la surface totale de Saturne.*

3.2. Résultats de simulation

Afin de valider les deux principaux concepts que nous proposons avec l'utilisation de l'unité de multiplexage matériel, nous proposons deux exemples permettant d'illustrer l'intérêt de cette approche. Le premier permet de représenter un déroulage dynamique de boucle, et le deuxième une allocation dynamique de quatre tâches.

3.2.1. Déroulage dynamique d'une boucle

Afin de valider le déroulage dynamique de boucle, prenons par exemple une tâche qui serait chargée d'exécuter le calcul de l'identité remarquable « a^2-b^2 » sur 512 échantillons 'a' et 'b'. La figure 104 illustre cet exemple d'abord sous la forme d'un programme séquentiel puis sous la forme de la configuration destinée à l'architecture reconfigurable.

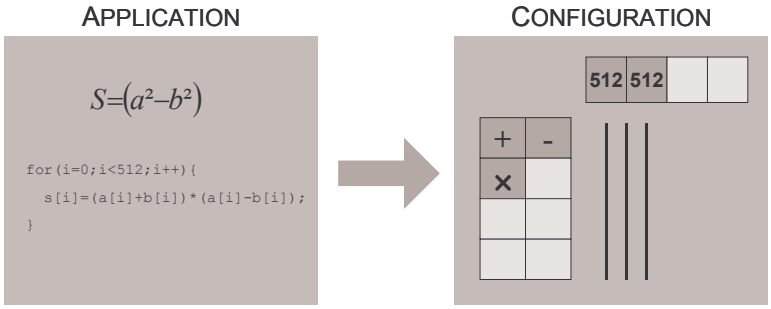
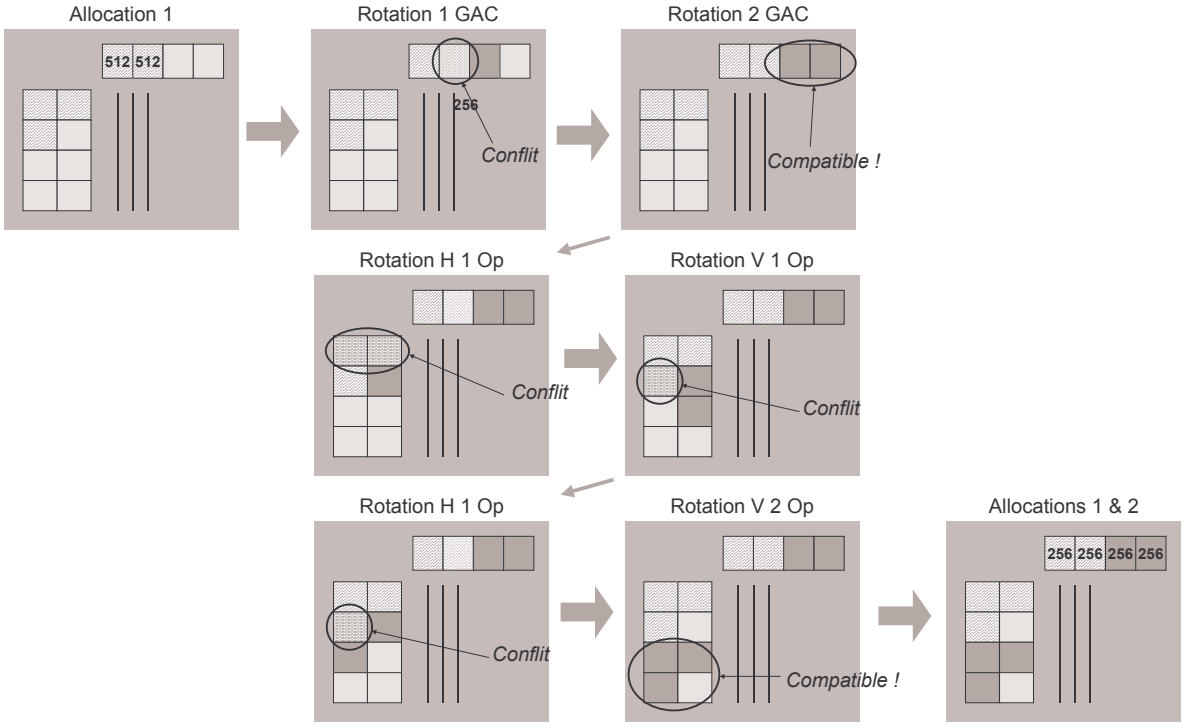


FIG. 104 - *Application et configuration de la tâche à implanter sur l'architecture.*

On suppose que toutes les ressources sont libres au moment où l'on va configurer cette tâche. Il est donc possible, après 2 rotations de la configuration des compteurs d'échantillons et 2 rotations de la configuration des opérateurs, d'allouer une deuxième fois la tâche à l'accélérateur.



Non déroulé

```
for(i=0;i<512;i++){
  y[i] = (a[i]+b[i]) * (a[i]-b[i]);
}
```

Déroulé (duplication du motif de calcul)

```
for(i=0;i<256;i+2){
  y[i]   = (a[i]+b[i]) * (a[i]-b[i]);
  y[i+1] = (a[i+1]+b[i+1]) * (a[i+1]-b[i+1]);
}
```

FIG. 105 - *Illustration de la phase de déroulage dynamique de boucle.*

Comme la phase de déroulage se passe durant le traitement des premiers échantillons, elle est masquée par le traitement et n'affecte pas les performances de la première allocation. Il faut cependant remarquer une fois de plus, que plus le nombre d'échantillons sera important, plus cette approche sera efficace (cf. critère d'accélération chapitre 3).

La figure 106 montre le gain obtenu avec un simple déroulage dynamique, sur 512 puis 1024 échantillons. Le nombre de cycles est quasiment divisé par 2 ce qui prouve l'intérêt d'une telle approche pour accélérer une tâche suivant les dispositions du matériel.

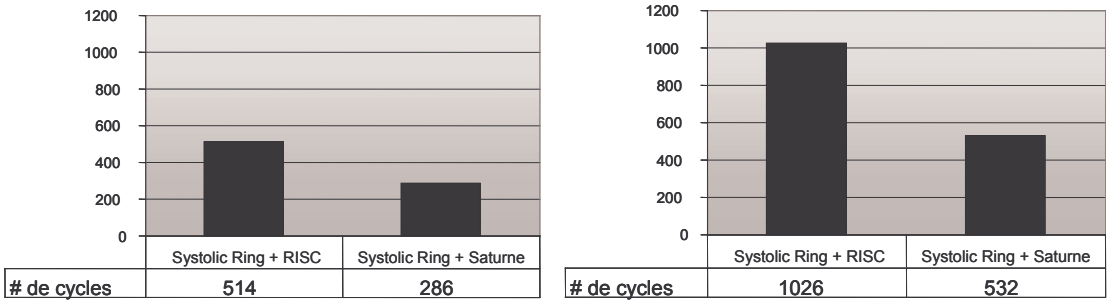


FIG. 106 - *Comparaison des nombres de cycles pour le Systolic Ring et Saturne. Le premier graphique illustre une boucle de 512 échantillons et le deuxième, une boucle de 1024 échantillons. Dans le premier cas, le gain est de 44% et dans le deuxième, de 93% (sans déroulage supplémentaire).*

Si l'on caractérise les résultats obtenus par les métriques intrinsèques, on obtient le tableau 28 qui résume les résultats.

Tab. 28. *Amélioration des performances par le multiplexage matériel*

	Systolic Ring+RISC	*Systolic Ring + Saturne*
D_{OP}	$8{,}6.10^{-2}$	$8{,}8.10^{-2}$
F_{acc} (512)	2,98	5,37
F_{acc} (1024)	2,99	5,77
τ_{inter}	35,7	35,7

3.2.2. Ordonnancement dynamique de plusieurs tâches

Pour illustrer l'intérêt de notre méthode, nous nous sommes basés sur un banc de test composé de quatre tâches pré-chargées dans la file d'attente d'exécution de l'architecture reconfigurable. Les quatre motifs algorithmiques implantés sont représentés sur la figure 107

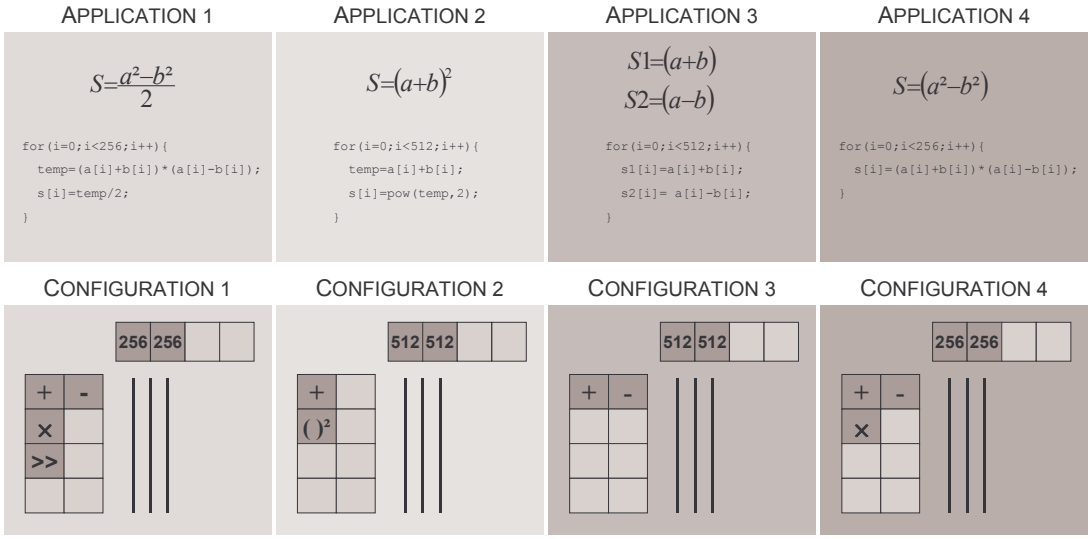


FIG. 107 - *Illustration des trois tâches projetées sur le Systolic Ring. Il s'agit ici de comparer les performances dans la gestion de multiples tâches.*

La figure 108 illustre l'ordonnancement des tâches sur le Systolic Ring, sans multiplexage matériel. On observe bien la sous-utilisation des ressources qui implique un temps de calcul total sous-optimal. Le taux d'utilisation moyen des ressources est dans ce cas égal à 39%.

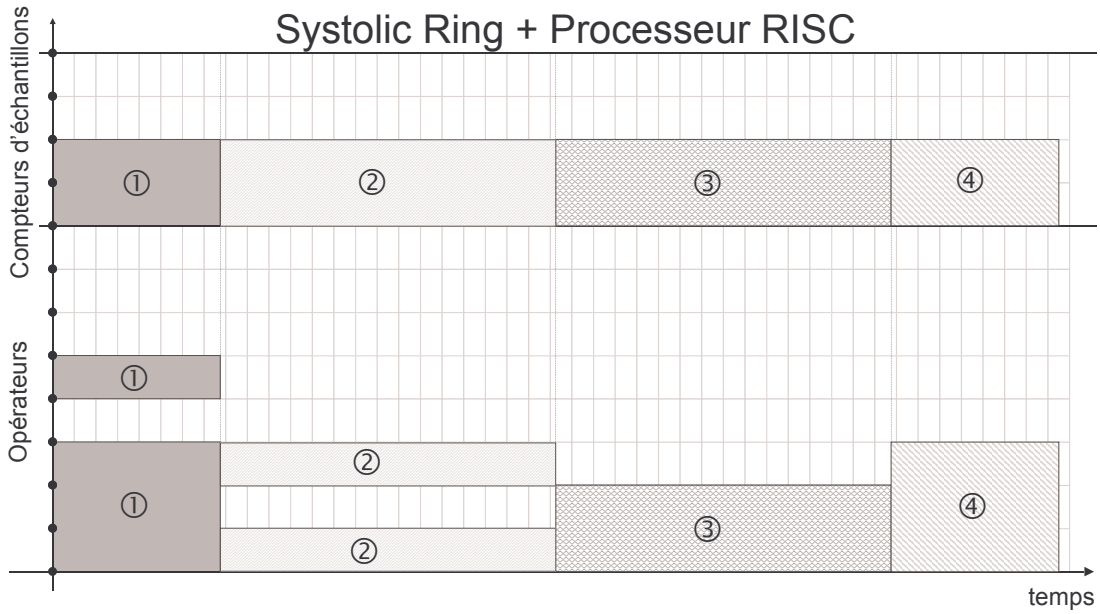


FIG. 108 - *Enchaînement des 4 configurations par ordonnancement statique des tâches. On observe une très nette sous utilisation des ressources qui implique au final un temps d'exécution largement sous-optimal*

La figure 109 montre qu'en répartissant les tâches sur les différentes ressources libres, on minimise le temps de calcul total, même si ce dernier n'est pas optimal. Ceci montre bien que le multiplexage matériel améliore la performance globale de l'accélérateur, sans impact sur la surface totale. Le taux d'utilisation moyen des ressources est largement améliorée puisque égal à 69%.

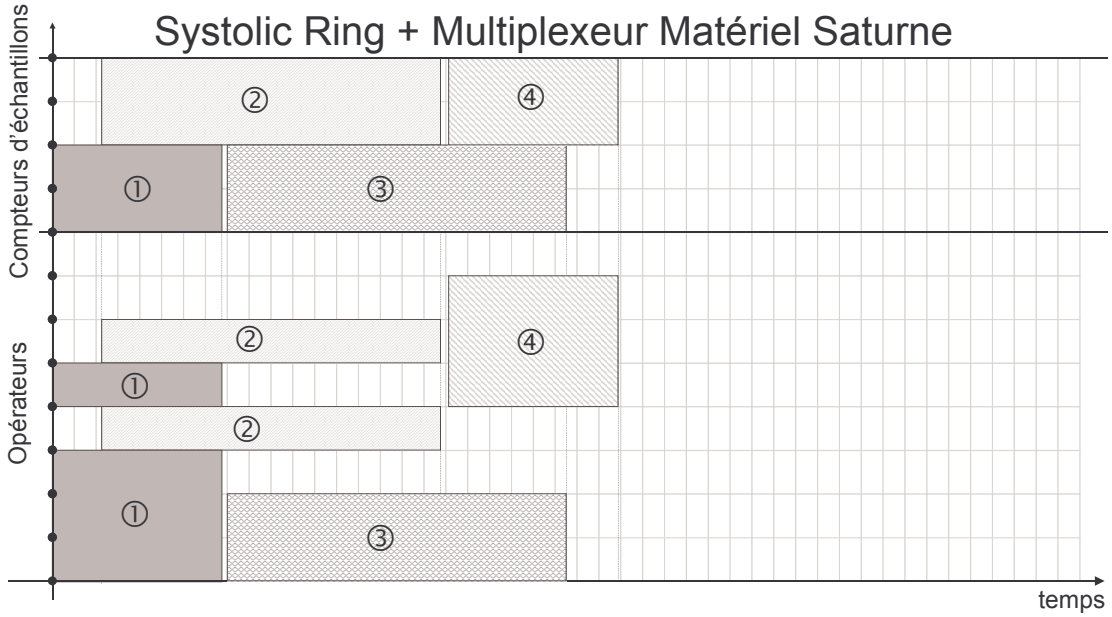


FIG. 109 - *Enchaînement des 4 configurations par ordonnancement dynamique des tâches. Par rapport à la figure précédente, on constate une meilleure utilisation des ressources ainsi qu'un temps d'exécution total quasiment divisé par 2.*

La figure 110 illustre l'amélioration des performances obtenues grâce à la mise en œuvre du multiplexage matériel qui permet de diminuer le nombre de cycles de traitement de 39% par rapport à l'implantation initiale. Il faut aussi préciser que cette amélioration de la performance ne se fait pas au détriment de la surface, comme le montre la densité opérative du tableau 29.

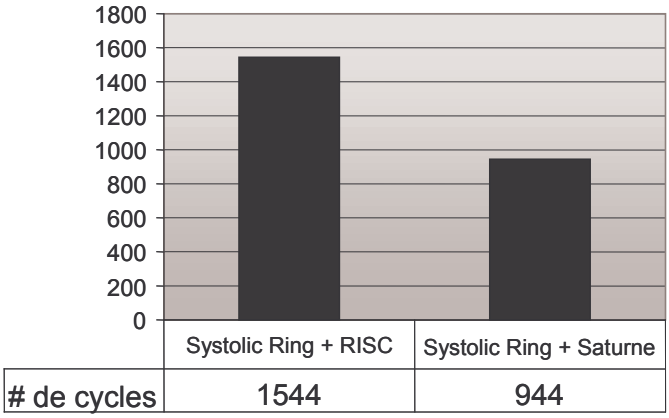


FIG. 110 - *Nombre de cycles pour le Systolic Ring et Saturne*

Si l'on caractérise les résultats obtenus par les métriques intrinsèques, on obtient le tableau 29 qui résume les résultats.

Tab. 29. *Amélioration des performances par le multiplexage matériel*

	Systolic Ring + RISC	*Systolic Ring + Saturne*
D_{OP}	$8{,}6.10^{-2}$	$8{,}8.10^{-2}$
F_{acc}	2,48	4,06
τ_{inter}	35,7	35,7

Nos derniers résultats montrent qu'une modification de l'ordre des tâches dans la file d'attente implique un nombre moyen de cycles de traitement correspondant à un facteur d'accélération borné et compris entre 3,9 et 4,1. De plus, si l'on compare les résultats obtenus sur l'ordonnancement de trois tâches et quatre tâches, on observe une croissance du facteur d'accélération. Autrement dit, les performances ont tendance à tendre vers des valeurs supérieures pour un nombre élevé de tâche. Afin d'étudier les limites d'accélération de la méthode proposée, il serait intéressant de faire une caractérisation en fonction du nombre de tâches, afin d'observer où se situe l'asymptote. L'objectif serait alors de proposer une méthode qui permettrait d'atteindre une valeur asymptotique plus élevée et au plus près de la valeur du nombre d'opérateurs.

4. CONCLUSIONS ET PERSPECTIVES

4.1. Bilan

Dans le chapitre 3, nous avons montré que les performances théoriques des architectures reconfigurables à grain épais devraient permettre des accélérations de traitement proches du maximum autorisé par le nombre de leurs unités de traitement, ceci grâce à leur faible rémanence. Or l'hypothèse faite sur une utilisation totale des opérateurs n'est pas forcément vérifiée (comme nous l'avons vu pour la version précédente du Systolic Ring, avec un taux d'utilisation moyen de 39% sur l'exemple proposé). En effet, la plupart des architectures ont une configuration prédéfinie, et ne savent pas s'adapter au contexte réel (par exemple, un lancement imprévisible de tâches non corrélées).

Afin d'améliorer les performances, nous avons proposé un concept générique de multiplexage matériel permettant une exploitation optimisée des ressources. Celui-ci s'opère soit par des déroulements dynamiques de boucles, soit par une allocation des ressources libres, soit les deux.

Afin de valider ce concept, nous avons proposé une architecture de multiplexeur matériel adapté aux besoins de notre support d'étude, le Systolic Ring. L'architecture proposée, Saturne, permet la mise en œuvre du multiplexage matériel et se révèle être d'un coût relativement faible (10% de la surface totale). En revanche, elle ne permet qu'une économie limitée de silicium par rapport à la version précédente basée sur l'utilisation du processeur RISC.

Sur deux exemples pédagogiques, nous avons présenté le gain de performance obtenu par l'adjonction de l'unité de multiplexage matériel. Avec un taux d'occupation supérieur des ressources de calcul (69%), on obtient des accélérations quasiment deux fois supérieures dans les cas exposés, prouvant ainsi le réel intérêt de cette approche. Ces premiers résultats encourageants nous ont amenés à proposer plusieurs perspectives à ce travail.

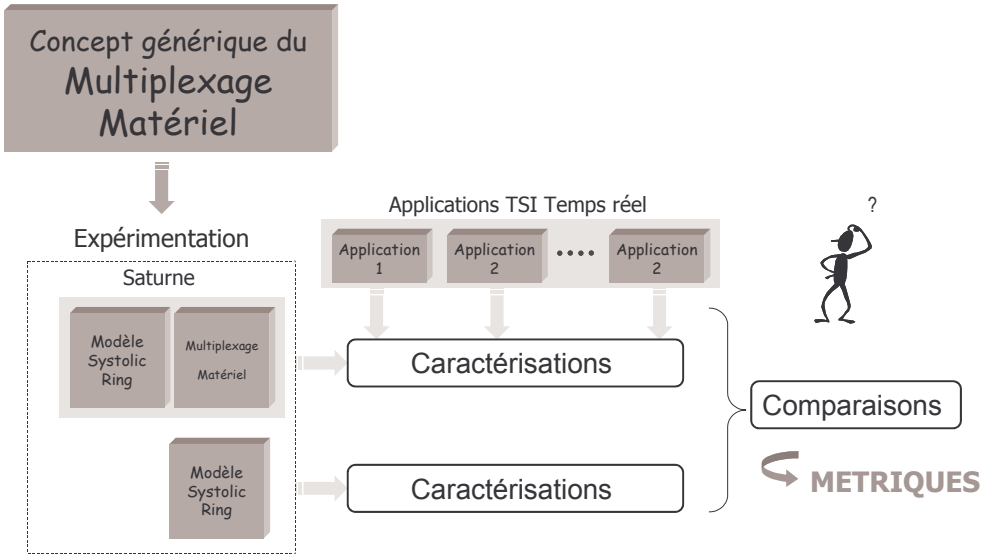


FIG. 111 - *Méthode générique du multiplexage matériel appliqué au Systolic Ring, caractérisation et comparaisons des performances.*

4.2. Perspectives d'évolution

Les premiers résultats obtenus sont satisfaisants et correspondent aux objectifs initiaux. Cependant, il est très certainement possible, à partir de ce travail, d'envisager des améliorations sur plusieurs plans, latence, consommation, surface, flexibilité, portabilité et scalabilité. C'est ce que nous présentons dans les prochains paragraphes.

4.2.1. Amélioration de l'heuristique

a. Minimiser la latence

Certaines phases de notre heuristique peuvent être parallélisées. Notamment, les phases où sont opérées les rotations des masques de configuration et les comparaisons sur les différents types de ressources effectués, pour le moment, en série. Ceci permettrait de réduire les cycles de latence introduits dans cette phase d'analyse et ne devrait impliquer que des modifications mineures du code. Cependant, comme nous l'avons montré dans les exemples choisis, les temps de traitement ont tendance à être très grand devant la latence introduite ce qui a un effet de masquage. En conclusion, nous pouvons penser que le gain obtenu sera minime.

b. Déroulage dynamique conditionnel

Pour l'instant, le principe de fonctionnement du multiplexeur matériel tend à accélérer tout motif algorithmique qu'il traite, ce qui correspond à favoriser le parallélisme spatial (parallélisme de données plus précisément) devant le parallélisme de tâche. Or, suivant la complexité du motif de calcul, le nombre de données à traiter ou encore, la priorité de la tâche, on peut très bien imaginer un multiplexeur matériel plus adaptatif que l'est actuellement Saturne. Une idée serait de conditionné le déroulage par un champ dans l'entête de configuration. Ainsi, une valeur fixée par l'utilisateur ou le système d'exploitation permettrait d'assurer des performances minimales en temps de traitement.

c. Disponibilité des données

On peut supposer que la gestion de la mémoire dans un contexte multi-tâches sera déterminante dans les performances globales du système. Cependant, nous pouvons imaginer également que notre système d'allocation dynamique pourra largement contribuer à un fonctionnement adaptatif de l'architecture reconfigurable en fonction de la disponibilité des données, à la manière des processeurs superscalaires SMT. Pour cela, il faudra envisager des registres permettant de mémoriser l'état d'un contexte. L'idée consisterait donc à disposer d'un deuxième registre d'état des tâches suspendues. Ceci permettrait donc de libérer les opérateurs qui attendent leurs données et les utiliser pour d'autres configurations.

d. Gestion des Priorités

L'accélération des traitements est un des objectifs des systèmes temps réels pour lesquels la fin d'un traitement doit être atteinte avant la date imposée par l'échéancier. Grâce à notre procédé d'allocation dynamique, il serait tout à fait possible de gérer des priorités de tâches. Pour ce faire, à chaque nouvelle génération d'adresse, le système vérifierait qu'une nouvelle tâche prioritaire n'est pas présente dans le *buffer* et procéderait alors à une configuration conditionnelle.

4.2.2. *Scalabilité* de l'unité de multiplexage matériel

La méthode d'allocation dynamique proposée a été mise en œuvre sur le Systolic Ring afin d'analyser la faisabilité et l'intérêt d'une telle approche pour la gestion des

reconfigurations, permettant d'exploiter du parallélisme au niveau tâches ou au niveau boucle. Afin d'étendre ces principes à des versions proposant un plus grand nombre d'opérateurs ou de types de ressources, nous allons essayer de voir quelles seraient les implications matérielles de ces modifications.

Considérons dans un premier temps une augmentation du nombre de ressources. C'est par exemple ce que nous avons proposé avec le modèle générique du Systolic Ring qui peut être personnalisé suivant des besoins de surface et de puissance de traitement, en comparant les différents compromis possibles avec des outils dédiés. Les paramètres variables sont le nombre de Dnodes par couche N, le nombre de couches C, le nombre de Bus B et le nombre d'anneaux S. Nous ne considérerons que les trois premiers paramètres et le nombre de compteurs d'échantillons G. Nous pouvons alors assez facilement formaliser l'effet de la modification de ces paramètres notamment sur la largeur des bus, la taille des registres ou encore le nombre d'états des machines d'états.

La taille de l'entête, tel qu'il est constitué jusqu'à présent, croît en *log2(paramètre)* pour ce qui des champs « nombre de ressources » et *linéairement en fonction du nombre de chaque ressource*. La taille des différents registres, qui est fonction de la largeur de l'entête, sera donc relativement *scalable*.

Si on modifie le nombre de ressources, en toute logique, il faudra également modifier le nombre de tâches prises en charge (actuellement au nombre de 4). Nous avons fixé ce nombre *n* en considérant qu'une tâche occupait au minimum 2 opérateurs. Si on change ce nombre, cela impliquera donc d'augmenter d'autant le nombre de registres d'état. De plus, il faudra également alors augmenter le nombre de tâches de la pile d'attente. On montre, par récurrence, que le nombre d'états de celle-ci sera de :

$$1+\sum_{k=0}^{n} 3k = 1+3.\frac{(n+1).n}{2}$$, n est le nombre de tâches

Pour la partie contrôle de l'unité de multiplexage matériel, les parties 1 (initialisations) et 3 (phase de configuration) ne seront pas modifiées si on change les nombres et types de ressources. Pour les parties 2 (Gestion des masques) et 4 (déroulage de boucle dynamique), si on augmente le nombre de ressources cela ne changera rien au nombre d'états mais cela aura en revanche des conséquences sur le nombre de cycles d'analyse (car il y aura plus de rotations possibles) : ceci aura donc tendance à accroître la latence, qui, même si elle est très inférieure actuellement, pourrait devenir contraignante à force. Cela nous amène à penser que si le nombre de ressources devient trop important, il sera nécessaire d'introduire une hiérarchie dans le multiplexage matériel, similaire à la hiérarchie d'interconnexion utilisée. De plus, si on modifie le type des ressources, en ajoutant des opérateurs spécialisés par exemple, il faudra ajouter autant de combinaisons de chemin supplémentaires permettant de le prendre en compte dans la partie 2 de la machine d'états.

4.2.3. Application à d'autres topologies d'architectures

En s'inspirant de la topologie circulaire du Systolic Ring, nous avons mis en œuvre des procédés de rotation des motifs de calcul pour transformer les topologie virtuelles. Une architecture en anneau est parfaitement adaptée à ce type de transformation car le bouclage du pipeline permet de considérer chaque position du motif comme relative.

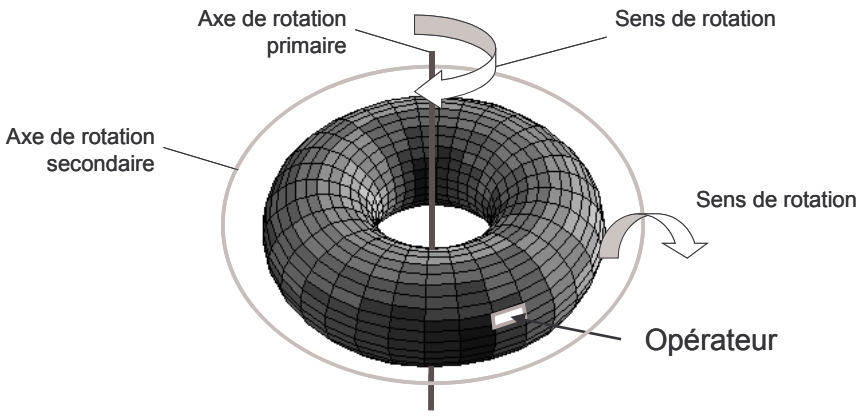


FIG. 112 - *Architecture torique. Il s'agit d'une extension de l'architecture en anneau. Elle permet d'empiler les configurations par le biais de rotations suivant 2 axes, en imposant cependant un sens afin de limiter l'espace des solutions.*

Une extension du principe de l'anneau est l'architecture torique : on pourrait la définir comme un « anneau d'anneaux ». Par notre principe d'allocation dynamique basé sur une rotation du motif algorithmique, l'architecture torique permettrait un deuxième axe de liberté. Une illustration de ce type d'architecture est présentée figure 112. Chaque quadrilatère représente un opérateur. Le sens de rotation primaire correspond au sens du flot de données. La conception du réseau d'interconnexion dans une telle topologie sera primordiale vis-à-vis des performances de l'allocation dynamique.

Aussi, au-delà d'un certain nombre de ressources, le coût des interconnexions devenant trop important, il devient nécessaire d'introduire un niveau de hiérarchie supplémentaire, comme nous l'avons fait pour le Systolic Ring. Ainsi, on aurait une unité de multiplexage matériel sur chaque tore qui serait gérée par une unité de multiplexage matériel supérieure qui répartirait les différentes tâches sur les différents anneaux.

L'étude de la portabilité du multiplexage matériel sur d'autres cibles technologiques, fait également parti des perspectives de travail relatives à cette thèse. C'est que nous présenterons dans les perspectives de ce travail.

CONCLUSION GENERALE ET PERSPECTIVES

Dans cette conclusion, nous allons revenir tout d'abord sur l'ensemble des travaux accomplis en présentant un résumé des contributions majeures de cette thèse. Nous proposerons ensuite quelques perspectives à notre étude.

BILAN

C'est dans un contexte d'applications TSI pour SoC que des architectures comme le Systolic Ring ont été proposées. Le rôle de ce type de composant est d'accélérer le traitement des applications en respectant des contraintes de consommation, de flexibilité et de coût. Les mécanismes de calcul reposent sur trois familles d'architectures : les processeurs, les ASIC et les architectures reconfigurables. Parce-que la flexibilité est un atout majeur dans un contexte évolutif, les accélérateurs tels que les DSP et les architectures reconfigurables semblent être voués à un avenir prometteur. Par conséquent, nous avons tout d'abord établi un état de l'art des architectures de traitement flexibles, en insistant plus particulièrement sur des exemples récents de réalisations tels que les processeurs superscalaires SMT, les DSP VLIW, les FPGA SRAM et les architectures reconfigurables à grain épais. Pour chacune, nous avons insisté sur les caractéristiques architecturales qui permettaient d'améliorer le compromis des performances, plus particulièrement dans le domaine du TSI ; nous avons analysé notamment les mécanismes matériels et/ou logiciels mis en oeuvre pour exploiter un maximum de parallélisme intra ou inter-application, ce qui s'avèrera être un point clé pour l'amélioration des performances.

Après un premier bilan qualitatif, les architectures reconfigurables à grain épais semblent disposer d'un compromis théorique intéressant pour les applications du TSI, notamment en tant que coprocesseur. Les nombreuses publications de ces dernières années démontrent d'ailleurs l'engouement de la communauté pour ce type de composant. C'est pour cette raison que le Systolic Ring a été élaboré dans un premier temps, amélioré ensuite, puis utilisé comme support d'étude et de validation pour les deux contributions importantes de cette thèse sur les thèmes de métriques et de multiplexage matériel. Au delà du cadre même fixé par ce support, cette étude reste néanmoins générique.

La première contribution de ce travail concerne la définition d'un ensemble de métriques afin de caractériser les architectures de traitement flexibles. L'objectif initial de ce travail visait à proposer des outils permettant de comparer différents composants existant et d'intervenir très tôt dans le flot de conception. En se basant sur un modèle d'architecture et un modèle de tâche, nous avons défini un ensemble de cinq métriques et nous avons procédé à une expérimentation sur plusieurs architectures d'accélérateurs (DSP et architectures reconfigurables à grain fin ou à grain épais). De cette étude, nous avons tiré plusieurs conclusions. Suivant la taille des motifs de calcul chaque accélérateur peut s'avérer plus ou moins efficace, ce qui montre l'intérêt des composants à granularité hybride lorsque l'on cherche à obtenir un compromis performances/flexibilité encore plus élevé. Pour bénéficier d'une accélération effective, les itérations de calculs à implanter doivent être grandes devant la rémanence de l'architecture. Ces métriques se révèlent également intéressantes pour la caractérisation d'architectures paramétrables. Elles permettent d'éclairer la définition des compromis pour les générations futures de composants, par une caractérisation basée sur l'évolution des caractéristiques. Par deux exemples théoriques et un pratique, nous avons illustré l'évolution des compromis architecturaux suivant le nombre d'opérateurs de ces architectures. Par cette analyse, il est possible de mettre en relief les paramètres critiques d'une architecture, et d'envisager des modifications telles que l'introduction d'une hiérarchie dans le système d'interconnexion. Pour terminer ce chapitre, nous avons formulé une remarque importante concernant le faible taux de parallélisme au niveau opération, soulignant l'intérêt de mettre en œuvre des méthodes logicielle et/ou matérielles visant à exploiter un

maximum de parallélisme intra et/ou inter-application. Cette nécessité s'avère notamment décisive pour des architectures en cours d'élaboration telles que les architectures reconfigurables à grain épais.

En effet, les résultats démontrent théoriquement un compromis performances/flexibilité très intéressant pour les architectures reconfigurables à grain épais, notamment dû à leur faible rémanence, autrement dit, leur fort dynamisme. Cependant, cette aptitude est soumise à une hypothèse d'exploitation maximale des ressources. Or, si l'on se réfère à un certain nombre de publications, on peut remarquer que le parallélisme moyen intra-fonction est relativement limité. C'est en partant de ce constat que nous avons eu l'idée de proposer une méthode de multiplexage matériel permettant aux architectures reconfigurables à grain épais, d'optimiser l'accélération globale par une utilisation dynamique des ressources. Cette méthode permet d'exploiter un parallélisme de niveau boucle et de niveau tâche de manière quasi-automatique. Par une réalisation matérielle baptisée Saturne, basée sur un contrôleur dédié et l'anneau de processeurs du Systolic Ring, nous avons montré après synthèse et simulation une amélioration significative des performances (d'un facteur 2 quasiment) sans surcoût matériel. De plus, l'automatisation matérielle du processus d'allocation devrait simplifier la mise en œuvre d'outils logiciels (Compilateur, Système d'exploitation).

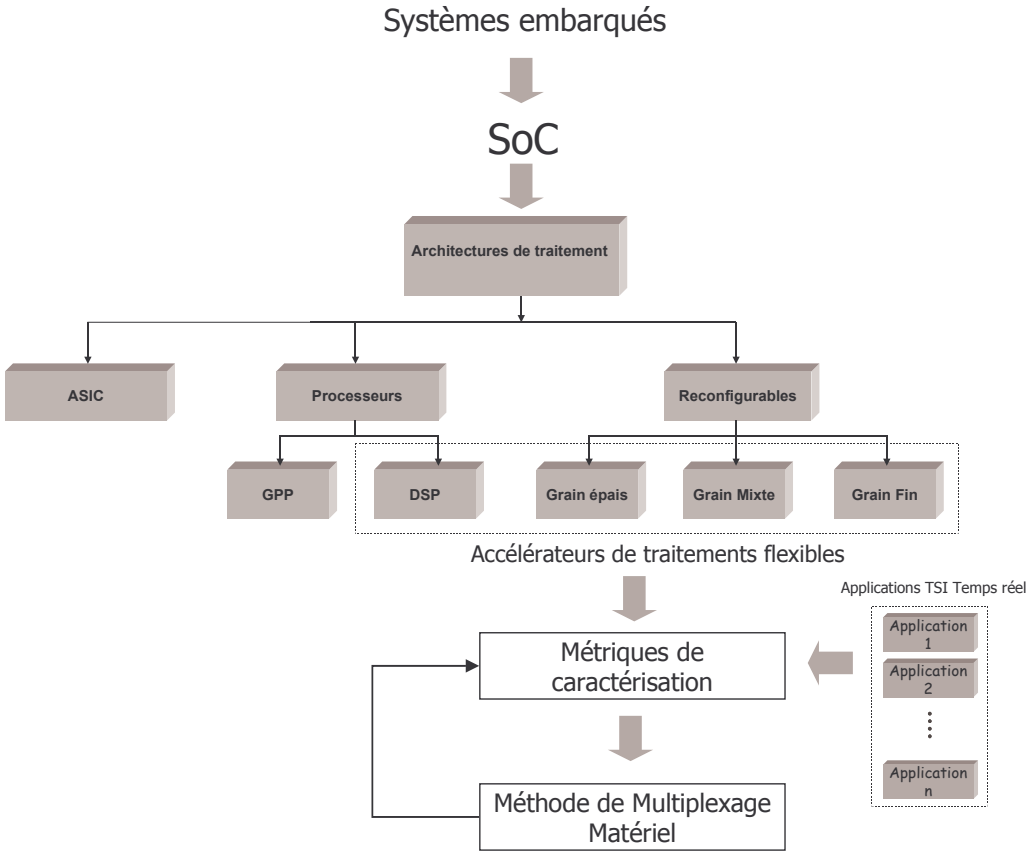


FIG. 113 - *Rappel du flot de thèse*

PERSPECTIVES

Le bilan des contributions doit maintenant être analysé de manière critique par rapport à la problématique initiale. Nous reviendrons successivement sur les deux contributions principales de cette thèse, tout d'abord sur le travail concernant les métriques puis sur notre méthode de multiplexage matériel.

Dans un contexte d'adéquation algorithme architecture, les métriques intrinsèques matérielles doivent être vues comme un outil complémentaire. En effet, celles-ci visent en tout premier lieu la caractérisation des architectures, en se basant sur un modèle de tâche très simple et un modèle d'architecture restreint. La discussion des hypothèses du modèle de tâche passe par une analyse des différentes applications du domaine TSI. Ainsi, un modèle plus précis, disposant de paramètres plus proches de la réalité des applications permettra d'affiner les résultats obtenus. L'amélioration du modèle d'architecture doit également être une perspective à ce travail. La diversité et la complexité des opérateurs, la sémantique des langages de programmation, sont deux aspects fondamentaux de cette étude. De même, une vision hiérarchique, notamment concernant les réseaux d'interconnexion, devra apparaître de façon plus claire dans l'expression des métriques. Enfin, une extension du modèle, prenant en compte les entrées/sorties et leurs débits, les mémoires et les moyens d'accès, fait parti des caractéristiques supplémentaires à prendre en compte.

De plus, les métriques proposées se placent systématiquement dans une vision optimiste de l'accélération, supposant une exploitation optimisée des ressources. Le compromis mesuré donne donc une indication de ce que l'on peut obtenir dans le meilleur des cas. Ce cas correspond à un déploiement maximal du parallélisme sur l'ensemble des unités de calcul. Ceci suppose donc que le parallélisme intrinsèque des applications est suffisamment élevé pour être déployé. Or cette hypothèse est loin d'être justifié au niveau intra-tâche si l'on s'en réfère à plusieurs publications. Par l'utilisation du multiplexage matériel, il est possible de produire plus de parallélisme intrinsèque par l'exécution simultanée de plusieurs tâches, ou bien par un parallélisme de données accru, obtenu par la duplication du motif de calcul. Cette méthode permet donc de se rapprocher des performances théoriques d'une architecture caractérisées par son facteur d'accélération.

L'algorithme de multiplexage matériel proposé est volontairement simple : l'utilisation d'un contrôleur câblé a motivé la spécification dans ce sens. Or, il est tout à fait envisageable de mettre en œuvre une heuristique plus complexe qui permettrait par exemple une gestion segmentée de l'espace des ressources, où il serait possible de modifier le motif initial afin de l'adapter aux espaces libres (non alloués à des opérations). Le support de ce multiplexage matériel pourrait alors très bien être un simple **processeur généraliste** ou de la **logique à grain fin** afin de modifier l'algorithme plus simplement suivant les besoins. Il est bien entendu que la contrainte majeure de mise en œuvre doit être la transparence d'exécution par rapport à l'architecture de traitement : la latence nécessairement introduite dans l'exploration de l'espace des solutions ne doit pas dégrader les performances de l'accélérateur.

La **gestion des injections** de données est également un des goulets d'étranglement de tout accélérateur capable de réaliser du parallélisme notamment en dupliquant des motifs de calcul. Nous avons simulé le comportement vis-à-vis de la mémoire « données » en utilisant un banc de compteurs d'échantillons. La prochaine étape consistera à mettre en commun l'étude qui est faite actuellement sur la gestion de l'alimentation de l'architecture en données et le contrôleur de multiplexage matériel.

Ce type d'architecture étant envisagé comme un coprocesseur, la réalisation d'un système comprenant un processeur, l'accélérateur, de la mémoire et des interfaces analogiques sera nécessaire pour réellement mesurer l'intérêt de ces approches tant du point de vue performance, que flexibilité. La réalisation d'un démonstrateur, utilisant la **plate-forme de prototypage** du Systolic Ring, permettra de valider notre système complet et d'asseoir l'intérêt de cette approche.

Il était nécessaire de choisir une cible technologique pour valider l'ensemble de ce travail, et c'est naturellement vers l'architecture du Systolic Ring, développée au LIRMM,

que nous nous sommes tournés. Néanmoins, nous avons toujours œuvré afin que les propositions faites soient le plus génériques. L'utilisation d'une topologie circulaire a permis la mise en œuvre d'une heuristique basée sur des rotations de motifs de configuration. Une perspective intéressante concerne l'étude de la **portabilité du multiplexage matériel** sur d'autres types de topologies. Cette étude, à plus long terme, devra également concerner l'hétérogénéité des cibles, en considérant par exemple des accélérateurs hybrides grain fin/grain épais.

Un aspect très important des SOC, et relativement peu abordé dans ce mémoire, concerne la **consommation d'énergie**. Nous nous sommes, en effet, essentiellement consacrés aux aspects temporels, à savoir l'accélération des traitements. Une étude de la consommation de l'unité de multiplexage matériel doit faire l'objet d'un travail à court terme. Il est tout à fait probable d'envisager une utilisation étendue de l'unité Saturne, par exemple comme contrôleur d'activité permettant de déclencher ou bien de mettre veille les unités, suivant les besoins opératifs. La mise en veille des ressources non utilisées éviterait alors une consommation inutile, augmentant ainsi l'efficacité énergétique du système.

Le succès des architectures reconfigurables à grain épais dépendra dans le futur de l'aboutissement d'outils logiciels et/ou matériels, permettant d'exploiter efficacement leurs ressources. Le succès des processeurs généralistes est dû en partie à la simplicité de programmation : c'est le compilateur qui le fait le lien entre la description de l'application et l'architecture de traitement. La réalisation de **compilateurs pour architectures reconfigurables** à grain épais fait parti des enjeux de la recherche actuellement dans ce domaine. L'automatisation matérielle de certaines tâches généralement ciblées par le compilateur (déroulage de boucle par exemple) devrait faciliter la génération de code. Une collaboration est en cours avec le Laboratoire d'Electronique des Systèmes TEmps Réel (LESTER) afin de définir une stratégie de compilation.

Enfin, l'utilisation de ce type d'accélérateur ne peut être envisagée sans son environnement opérationnel, à savoir les différentes composantes du système (processeur central, mémoire, etc.) mais aussi la couche logicielle permettant la gestion globale des ressources, à savoir le Système d'Exploitation. L'utilisation d'un contrôleur de multiplexage matériel est voué à la prise en charge de services généralement réalisés par l'OS. Une définition plus aboutie de l'interaction OS-accélérateur doit faire l'objet des **perspectives au niveau système**.

.

BIBLIOGRAPHIE

[Acte04] ACTEL, “40 MX datasheet”, http://www.actel.com

[Aho86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, “Compilers: Principles, Techniques and Tools”, Addison-Wesley Publishing Company, 1986

[Anal04] Analog Devices Inc. Web Site, http://www.analog.com

[Apex04] Altera Corp. Web Site, “APEX II Datasheet”, www.altera.com

[Arm04] ARM, “AMBA 2.0 Specification”, http://www.arm.com

[Atme02] Atmel Corp. Web Site, “AT40K Series”, April 2002, http://www.atmel.com/

[Atme04] Atmel Corp. Web Site, “AT6000 datasheet”, http://www.atmel.com

[Baum01] V. Baumgarte, F. May, A. Nükel, M. Vorbach, and M. Weinhardt. PACT XPP “A self-Reconfigurable Data Processing Architecture”, In International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 01), Las Vegas, USA, Juin 2001

[Beno02a] P. Benoit, G. Sassatelli, M. Robert, L. Torres, G. Cambon, T. Gil, “Architectures Reconfigurables Dynamiquement pour Applications TSI”, Troisième colloque de circuits et systèmes intégrés, Paris, 15-16-17 mai 2002, pp 67-70

[Beno02b] P. Benoit, G. Sassatelli, M. Robert, L. Torres, G. Cambon, T. Gil, “The Systolic Ring: a Scalable Dynamically Reconfigurable Architecture for Embedded Systems”, 5th Edition Sophia Antipolis forum on MicroElectronics, octobre 2002, Sophia Antipolis, France, pp. 85-90

[Beno02c] P. Benoit, G. Sassatelli, M. Robert, L. Torres, G. Cambon, T. Gil, “Caractérisation et comparaison d’architectures reconfigurables dynamiquement. Un exemple: Le Systolic Ring”, Journées Francophones sur l’Adéquation Algorithme Architecture, décembre 2002, Monastir, Tunisie, pp. 30-34

[Beno03a] P. Benoit, G. Sassatelli, L. Torres, D. Demigny, M. Robert, G. Cambon, “Metrics for Reconfigurable Architectures Characterization: Remanence and Scalability”, IEEE 17th International Parallel and Distributed Processing Symposium (IPDPS'03) 10th Reconfigurable Architectures Workshop (RAW’03), avril, 21-25 2003, Nice, France, p. 176

[Beno03b] P. Benoit, G. Sassatelli, M. Robert, L. Torres, G. Cambon, “Comparaison des architectures dédiées au traitement des données numériques”, Journées Nationales du Réseau Doctoral Microélectronique, Micro et Nano Technologies, 14-16 mai 2003, Toulouse, France, pp. 115-117.

[Beno03c] P. Benoit, G. Sassatelli, L. Torres, D. Demigny, M. Robert, G. Cambon, “Metrics for DSP Architectures Characterization : Remanence and Scalability, Systems, Architectures”, Modeling and Simulation workshop (SAMOS’03), 20-24 juillet 2003, Samos, Grèce

[Beno03d] P. Benoit, G. Sassatelli, L. Torres, D. Demigny, M. Robert, G. Cambon, “A Novel Approach for Reconfigurable Architecture Characterization. An example through the Systolic Ring”, 13th International Conference on Field Programmable Logic and Application (FPL’03), 1-3 septembre 2003, Lisbonne, Portugal, pp. 722-732

[Beno03e] P. Benoit, G. Sassatelli, M. Robert, L. Torres, G. Cambon, "Comparaisons intrinsèques des architectures reconfigurables", Symposium en Architectures Nouvelles de Machines, 14-17 octobre 2003, La Colle sur Loup, France, pp. 322-326.

[Blod03] B. Blodget, S. McMillan and P. Lysaght, "A Lightweight Approach for Embedded Reconfiguration of FPGAs", Design Automation and Test in Europe Conference and Exhibition (DATE'03), Allemagne, 2003, pp.399-400

[Boud04] N. Boudouani, "Architectures reconfigurables dynamiquement : synthèse matérielle d'opérateurs de détection et d'estimation de mouvement temps réel" , thèse de doctorat, université de Cergy-Pontoise, 2004

[Bitt96] R. A. Bittner et al., "Colt: An Experiment in Wormhole Run-time Reconfiguration", SPIE Photonics East `96, Boston, MA, USA, Novembre 1996

[Blan02] F. Blanc, "Etude d'un nouveau concept de calculateur reconfigurable : architecture et outils", CEA, Saclay, France, Décembre 2002

[Brow96] S. Brown and J. Rose, "Architecture of FPGAs and CPLDs: A Tutorial", IEEE Design and Test of Computers, Vol. 13, No. 2, 1996, pp. 42-57

[Burd01] T. Burd, "General Processor Infomation", Janvier 2001

[Call98] T. J. Callahan and J. Wawrzynek, "Instruction Level Parallelism for Reconfigurable Computing", In Hartenstein and Keevallik, editors, *FPL'98,* Field-Programmable Logic and Applications, 8th International Workshop, Tallinin, Estonia, volume 1482 of Lecture Notes in Computer Science, Springer-Verlag, Septembre 1998

[Call00] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler", *IEEE Computer*, Avril 2000, pp. 62-69

[Cham00] X. Tang, M. Aalsma, and R. Jou, "A compiler directed approach to hiding configuration latency in chameleon processors", In International Workshop on Field Programmable Logic and Applications (FPL 00), Villach, Autriche, Avril 2000, pp.29-38

[Celo04] Celoxica Web Site, http://www.celoxica.com

[Codi02] J. M. Codina, J. Llosa et A. González, "A Comparative Study of Modulo Scheduling Techniques", Proc. of the 26th International Conference on Supercomputing, USA, New York City, Juin 2002, pp.97-106

[Colw88] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a trace scheduling compiler", IEEE Transactions on Computers, vol. 37, 1988, pp. 967-979

[Const88] Constantine D. Polychronopoulos, "Parallel Programming and compilers", Kluwer Academic Publishers, ISBN 0-89838-288-2

[Cron98] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling. "Specifying and Compiling Applications for RaPiD". In Symposium on Field-Programmable Custom Computing Machines (FCCM 98), Los Alamitos, USA, Avril 1998, pp. 116-125

[Curd03] D. R. Curd, "Partial Reconfiguration of RocketIO Pre-emphasis and Differential Swing Control Attributes", Xilinx, 2003

[Davi02] R. David, D. Chille, S. Pillement, and O. Sentieys, "DART : A Dynamically Reconfigurable Architecture dealing with Next Generation Telecommunications Constraints", In International Reconfigurable Architecture Workshop (RAW 02), Fort lauderdale, USA, Avril 2002, CD-ROM proceedings

[Deho98] A. DeHon, "Comparing Computing Machines", Configurable Computing: Technology and Applications, Proc. SPIE 3526, 2-3 Novembre 1998, pp. 124-133

[Demi99] D. Demigny *et al.*, "Architecture à reconfiguration dynamique pour le traitement temps réel des images", Techniques et Sciences de l'Information, Numéro Spécial Architectures Reconfigurables, 18(10), décembre 1999, pp. 1087-1112

[Demi01] Didier Demigny, "Méthodes et architectures pour le TSI en temps reel", Traité IC2 – Série Traitement du signal et de l'image, ISBN : 2-7462-0327-8

[Demi02] D. Demigny, N. Boudouani, N. Abel, and L. Kessal, "La rémanence des architectures reconfigurables, un critère significatif des architectures", proc. of Journées Francophones sur l'Adéquation Algorithme Architecture, décembre 2002, Monastir, Tunisie, pp. 49-52

[Dub90] P.K. Dubey, M.J. Flynn, "Optimal pipelining", Journal of Parallel and Distributed Computing, vol. 8, 1990, pp.10-19

[Ebel96] C. Ebeling *et al.*, "RaPiD A Configurable Computing Architecture for Compute Intensive Applications", 1996, http://www.citeseer.nj.nec.com/article/ebeling96rapid.html

[Ebel03] C. Ebeling, C. Fisher, G. Xing, M. Shen, H. Liu, "Implementing an OFDM receiver on the Rapid Reconfigurable Architecture", 13th International Conference on Field Programmable Logic and Application (FPL'03), 1-3 septembre 2003, Lisbonne, Portugal, pp. 21-30, ISSN 0302-9743

[Fish84] J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", in IEEE Computer, vol. 17, no. 7, Juillet 1984, pp. 45-53

[Gold99] S. C. Goldstein *et al.*, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration", Proceedings ISCA'99, Atlanta, 2-4 mai 1999

[Gucc94] S. A. Guccione, "List of FPGA-based computing machines", World Wide Web page http://www.io.com/~guccione/HW_list.html, mis à jour depuis 1994

[Guer00] P. Guerrier, A. Greiner, "A Generic Architecture for On-chip Packet-switched Interconnections", Proc. Of International Conference on Design Automation and Testing in Europe 2000 (DATE'2000), Paris, France, Mars 2000, pp. 250-256

[Gwen94a] L. Gwennap, "Digital leads the Pack with the 21164", microprocessor report, 1994, pp. 6-10

[Gwen94b] L. Gwennap, "MIPS R10000 uses decoupled architectures", microprocessor report, 1994, pp. 18-22

[Hart00] R. Hartenstein, M. Herz, Th. Hoffmann et U. Nageldinger, "KressArray Xplorer: A New CAD Environment to Optimize Reconfigurable Datapath

Array Architectures", Proc. of 5th Asia and South Pacific Design Automation Conference, Japon, Yokohama, Janvier 2000, pp. 163-168

[Hart01] R. Hartenstein, "A Decade of Research on Reconfigurable Architectures - a Visionary Retrospective", Proc. Of International Conference on Design Automation and Testing in Europe 2001 (DATE'2001), Exhibit and Congress Center, Munich, Germany, March 2001

[Haus00] J. Hauser, "Augmenting a microprocessor with reconfigurable hardware", PhD thesis, University of California, Berkeley, 2000

[Henn03] J. L. Hennessy, D. A. Patterson, "Architecture des ordinateurs : une approche quantitative", (troisième édition). Vuibert Informatique, 2003

[Hwa84] K. Hwang and F. A. Briggs, "Computer Architecture and Parallel Processing", McGraw-Hill Book Company, 1984

[Inte01] Intel Application Notes for Pentium MMX, http://developer.intel.com/

[Inte04] Intel, "Mobile Intel® Pentium® 4 Processor supporting Hyper-Threading Technology on 90-nm process technology", specification update, juin 2004, Document Number: 302441-001

[John78] P. M. Johnson, "An introduction to vector processing" Computer Design, 17(2), Février 1978, 156, pp.89- 97

[John90] M. Johnson, "superscalar design", Englewood Cliffs, New Jersey: Prentice Hall, 1990

[Kung82] H. T. Kung, "Why systolic architectures?" IEEE Computer, 15(1), Janvier 1982, pp. 37-46

[Laca98] L. Lacassagne, F. Lohier , P. Garda, "Real time execution of optimal edge detectors on risc and dsp processors", Congrès ICASSP, 1998, pp. 3101-3104

[Laga00] L. Lagadec, "Abstraction, modélisation et outils de CAO pour les circuits intégrés reconfigurables", thèse de doctorat, Université de Rennes 1, 2000

[Lemo03] Y. Le Moullec, D. Heller, J-Ph. Diguet et J-L. Philippe, "Estimation du parallélisme au niveau système pour l'exploration de l'espace de conception de systèmes enfouis" , Technique et Science Informatiques (RSTI-TSI), Vol. 22, n°3/2003, Lavoisier Hermes-Science publications, pp. 315-349

[Lewi04] A. Lewillion, "Plateforme d'évaluation des systèmes sur puce", rapport de DEA, Université de Montpellier II, 2004

[Mad00] D. Madon, "Processeur RISC multitâche", thèse de doctorat, Ecole polytechnique de Lausanne, 2000

[Mirs96] E. Mirsky, A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources", Proc. of IEEE FCCM'96, Napa, CA, USA, 17-19 Avril 1996

[Moo93] C. R. Moore, "The PowerPC 601 microprocessor", Proc. Compcon 1993, février 1993, pp. 109-116

[Mori98] C. A. Moritz, D. Yeung, A. Agarwal, "Exploring Optimal Cost-Performance Designs for Raw Microprocessors", Proc. of the International IEEE Symposium on Field Programmable Custom Computing Machines, avril 1998, pp. 12-27

[Moto04] Motorola Inc., http://www.motorola.com

[Nade01] U. Nadelginder, "Coarse-Grained Reconfigurable Architecture Design Space Architecture Exploration", *Ph.D. Thesis*, Allemagne, University of Kaiserslautern, Juin 2001

[Nios00] Altera. Nios Soft Core Embedded Processor, Altera, Juin 2000.

[Omap04] OMAP, "OMAP 2 Architecture: OMAP2410 and OMAP2420 Processors", www.ti.com/omap2

[Pete 95] R. Petersen and B. L. Hutchings, "An assessment of the suitability of FPGA based systems for use in digital signal processing", in The 5th International Workshop on FieldProgrammable Logic and Applications (FPL'95), Oxford England, Août 1995, pp. 293-302

[Poma04] http://r2d2.enssat.fr/pomard/

[Raba96] J.M. Rabaey, M. Pedram, "Low Power design Methodologies", Edition Kluwer Academic Publishers, 1996

[Raba00] J.M. Rabaey, "Low-Power Silicon Architectures for Wireless Applications", Proc. Of ASPDAC'2000, Yokohama, Japon, Janvier 2000, pp. 377-380

[Riso04] S. Riso, L. Torres, G. Sassatelli, M. Robert, F. Moraes, "Réseau d'Interconnexion pour les Systèmes sur Puce : le Réseau HERMES", Proc. Of SCS'2004 Signaux, Circuits et Systèmes , 2004, pp. 35-40

[Rubi97] S. Rubini, D. Lavenier, "Les Architectures Reconfigurables", Calculateurs Parallèles, 9(1), 1997

[Rupp98] C. Rupp, M. Landguth, T. Graverick, E. Gomersall, and H. Holt, "The NAPA Adaptative Processing Architecture", Proc. Of IEEE Symposium on Field-Programmable Custom Computing Machines 1998 (FCCM'1998), Avril 1998, pp. 28-37

[Sass01] G. Sassatelli, L. Torres, J. Galy, G. Cambon, and C. Diou, "The Systolic Ring : A Dynamically Reconfigurable Architecture for Embedded Systems", In Proc. Of International Workshop on Field Programmable Logic and Applications 2001(FPL'2001), Lecture Notes in Computer Science 2147, pp. 409-419

[Sass02] G. Sassatelli, "Architectures reconfigurables dynamiquement pour les Systèmes sur Puce", thèse de doctorat, Université de Montpellier II, 2002

[Shiv00] S. G. Shiva, "Computer Design and Architecture", third edition, Marcel Dekker editor, ISBN 0 8247 0368 5, 2000

[Sing98] H. Singh *et al.*, "MorphoSys: An Integrated Reconfigurable Architecture", Proc. of the NATO RTO Symp. on System Concepts and Integration, Monterey, CA, USA, Avril 1998, pp. 20-22

[Sing00] H. Singh *et al.* , "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications", IEEE Trans. on Computers, Vol.49, No.5, Mai 2000, pp.465-481

[Sing01] S. Singh and P. James-Roxby, "Lava and JBits: From HDL to Bitstreams in Seconds", IEEE Symposium on FPGAs for Custom Computing Machines, Avril 2001

[Smit95] J.E. Smith, G.S. Sohi, "The microarchitecture of superscalar processors", Proceedings of the IEEE, vol. 83, Décembre 1995, pp. 1609-1624

[Strat04] Stratix II, www.altera.com, Altera, 2003

[Socl04] Soclib, "the Soclib project", http://soclib.lip6.fr/Soclib

[Soni04] Sonics Inc. Corporate Web Site, http://www.sonicsinc.com

[Stmi04] ST Microelectronics Inc. Web Site, http://www.st.com

[Syst04] SystemC Web Site, http://www.systemc.org

[Texa00a] Texas Instrument, "TMS320VC5410 Fixed-Point Digital Signal Processor", Data Manual, Décembre 2000

[Texa00b] Texas Instrument, "TMS320C62X Image/Video Processing library Programmer's Reference", Mars 2000, www.ti.com

[Texa04] Texas Instrument Inc. Web Site, http://www.ti.com, 2004

[Tull95] D.M. Tullsen *et al.*, "Simultaneous Multithreading: Maximizing on-chip parallelism", In Proc. Of the 22nd. annual. Intl. Symp. On Computer Architecture, Juin 1995, pp. 392-403

[Virt04] Xilinx Virtex II Pro, www.xilinx.com, Xilinx

[Von45] J. Von Neumann, "First Draft of a Report on the EDVAC", Moore School of Electrical Engineering, University of Pennsylvania, Juin 1945

[Wirth98] M. J. Wirthlin and B. L. Hutchings, "Improving Functional Density Using Run-Time Circuit Reconfiguration", IEEE Transactions On Very Large Scale Integration (VLSI) Systems, Vol. 6, Juin 1998, pp. 247-256

[Xili00] Xilinx, the Programmable Logic Data Book, 2000

[Xput01] "Why reconfigurable computing", Department of Computer Science, Computer Structures Group, http://xputers.informatik.uni-kl.de/

PUBLICATIONS RELATIVES A LA THESE

Brevets

[Brev-a02] GILLES SASSATELLI, LIONEL TORRES, GASTON CAMBON, MICHEL ROBERT, JEROME GALY, PASCAL BENOIT, *Architecture de calcul logique comprenant plusieurs modes de configuration*, 2002

[Brev-b02] GILLES SASSATELLI, LIONEL TORRES, GASTON CAMBON, MICHEL ROBERT, JEROME GALY, PASCAL BENOIT, *Architecture de transport de données en anneau comprenant un réseau de rétro-propagation*, 2002

Revues

[TS'02] GILLES SASSATELLI, PASCAL BENOIT, LIONEL TORRES, GASTON CAMBON, JEROME GALY, MICHEL ROBERT, CAMILLE DIOU, *Systolic Ring :Une Nouvelle Approche Pour Les Architectures Reconfigurables Dynamiquement*, Traitement du Signal, 2002, Vol. 19, n° 4, pp 293-313.

Contributions à Ouvrages Scientifiques

[KLU'02] GILLES SASSATELLI, LIONEL TORRES, PASCAL BENOIT, GASTON CAMBON, MICHEL ROBERT, JEROME GALY, *Dynamically Reconfigurable Architecture for Digital Signal Processing Applications*, Kluwer Academic Publishers, SoC Design Methodologies, selection des meilleurs papiers de IFIP 12th International Conference on Very Large Scale Integration of Systems-on-Chip, 2002, pp 63-74. ISBN 1-4020-7148-5.

[SAMOS'04] PASCAL BENOIT, GILLES SASSATELLI, LIONEL TORRES, MICHEL ROBERT, GASTON CAMBON, DIDIER DEMIGNY, COMPUTER SYSTEMS: ARCHITECTURES, MODELING AND SIMULATION, *Metrics for Digital Signal Processing Architectures Characterization : Remanence and Scalability*, Systems, Architectures, 3rd and 4th International Workshops SAMOS 2003 and SAMOS 04, july 2003 and july 2004, Samos, Greece, pp 128-137. ISBN 3-540-22377-0

Conférences avec actes et comité de lecture international

[DATE'02] GILLES SASSATELLI, LIONEL TORRES, PASCAL BENOIT, THIERRY GIL, CAMILLE DIOU, GASTON CAMBON, JÉRÔME GALY, *Highly Scalable Dynamically Reconfigurable Systolic Ring-Architecture for DSP applications*, IEEE Design, Automation and Test in Europe (DATE'02), march, 4-8 2002, Paris, France, pp 553-558. ISBN 0-7695-1471-5.

[SOC'02] GILLES SASSATELLI, LIONEL TORRES, PASCAL BENOIT, GASTON CAMBON, MICHEL ROBERT, JÉRÔME GALY, *Dynamically Reconfigurable Architectures for Digital Signal Processing applications*, SOC Design Methodologies, 2002, pp 63-74.

[RAW'03] PASCAL BENOIT, GILLES SASSATELLI, LIONEL TORRES, DIDIER DEMIGNY, MICHEL ROBERT, GASTON CAMBON, *Metrics for Reconfigurable Architectures Characterization: Remanence and Scalability*, IEEE 17th International Parallel and Distributed Processing Symposium (IPDPS'03) 10th Reconfigurable Architectures Workshop (RAW'03), april, 21-25 2003, Nice, France, p. 176.

[SAMOS'03] PASCAL BENOIT, GILLES SASSATELLI, LIONEL TORRES, MICHEL ROBERT, GASTON CAMBON, DIDIER DEMIGNY, *Metrics for DSP Architectures Characterization : Remanence and Scalability*, Systems, Architectures, Modeling and Simulation workshop (SAMOS'03), july, 20-24 2003, Samos, Greece.

[FPL'03] PASCAL BENOIT, GILLES SASSATELLI, LIONEL TORRES, MICHEL ROBERT, GASTON CAMBON, DIDIER DEMIGNY, *A Novel Approach for Reconfigurable Architecture Characterization. An example through the Systolic Ring*, 13th International Conference on Field Programmable Logic and Application (FPL'03), september, 1-3 2003, Lisbon, Portugal, 722-732.

Conférences sans actes

[GDRCAO'02] PASCAL BENOIT, GILLES SASSATELLI, MICHEL ROBERT, LIONEL TORRES, GASTON CAMBON, THIERRY GIL, *Architectures Reconfigurables Dynamiquement pour Applications TSI*, Troisième colloque de circuits et systèmes intégrés, Paris, 15-16-17 mai 2002, pp 67-70.

[SAME'02] PASCAL BENOIT, GILLES SASSATELLI, MICHEL ROBERT, LIONEL TORRES, GASTON CAMBON, THIERRY GIL, *The Systolic Ring: a Scalable Dynamically Reconfigurable Architecture for Embedded Systems*, SAME'02, 5th Edition, Sophia Antipolis forum on MicroElectronics, 9-10 octobre 2002, Sophia Antipolis, France, pp 85-90.

[JFAAA'02] PASCAL BENOIT, GILLES SASSATELLI, MICHEL ROBERT, LIONEL TORRES, GASTON CAMBON, THIERRY GIL, *Caractérisation et comparaison d'architectures reconfigurables dynamiquement. Un exemple: Le Systolic Ring*, Journées Francophones sur l'Adéquation Algorithme Architecture (JFAAA), 16-18 décembre 2002, Monastir, Tunisie, pp 30-34.

[JNRDM'03] PASCAL BENOIT, GILLES SASSATELLI, LIONEL TORRES, MICHEL ROBERT, GASTON CAMBON, *Comparaison des architectures dédiées au traitement des données numériques*, Journées Nationales du Réseau Doctoral Microélectronique, Micro et Nano Technologies, 14-16 mai 2003, Toulouse, France, pp 115-117.

[SympAAA'03] PASCAL BENOIT, GILLES SASSATELLI, LIONEL TORRES, MICHEL ROBERT, GASTON CAMBON, *Comparaisons intrinsèques des architectures reconfigurables*, Symposium en Architectures Nouvelles de Machines, 14-17 octobre 2003, La Colle sur Loup, France, pp. 322-326.

[Doctiss'04] PASCAL BENOIT, GILLES SASSATELLI, LIONEL TORRES, MICHEL ROBERT, GASTON CAMBON, *Architectures Reconfigurables : les processeurs du futur*, Journées des doctorants de l'école doctorale I2S, 2 mars 2004, Montpellier, France, CD-ROM proceedings.

Diffusions et distinctions

[SOC'01] Conférence invitée, LIONEL TORRES, GILLES SASSATELLI, GASTON CAMBON, PASCAL BENOIT, MICHEL ROBERT, *The Systolic Ring : A Dynamically Reconfigurable Architecture for SOCs and Embedded Systems*, Systems on Chips workshop, organized by NOKIA and the Tampere University of Technology, Finland, CDROM proceedings.

[EETimes'02] *Systolic Ring balances Power and Throughput*, Article concernant l'architecture développée, dans le journal EETIMES, Octobre 2002.

EN COURS D'EVALUATION

Revues

[TSI'04] PASCAL BENOIT, GILLES SASSATELLI, LIONEL TORRES, DIDIER DEMIGNY, MICHEL ROBERT, GASTON CAMBON, *Applications des métriques intrinsèques pour Architectures Reconfigurables Dynamiquement*, Techniques des Sciences Informatiques, 2004, évaluation en cours.

Conférences avec actes et comité de lecture international

[DATE'05] PASCAL BENOIT, GILLES SASSATELLI, LIONEL TORRES, MICHEL ROBERT, GASTON CAMBON, *Improving throughput in coarse grain reconfigurable architectures*, Design Automation and Test in Europe (DATE'05), mars 2005, Munich, Allemagne.

[FPGA'05] PASCAL BENOIT, GILLES SASSATELLI, LIONEL TORRES, MICHEL ROBERT, GASTON CAMBON, *Dynamic hardware multiplexing for coarse grain reconfigurable architectures*, Field Programmable Gate Array (FPGA'05), février 2005, Monterey, Etats-Unis d'Amérique.

[RAW'05] PASCAL BENOIT, GILLES SASSATELLI, LIONEL TORRES, MICHEL ROBERT, GASTON CAMBON, *A RTR hardware controller for DSP kernel scheduling in coarse grain reconfigurable architectures*, Reconfigurable Architectures Workshop (RAW'05), avril 2005, Denver, Etats-Unis d'Amérique.

Conférences sans actes

[JFAAA'05] PASCAL BENOIT, GILLES SASSATELLI, MICHEL ROBERT, LIONEL TORRES, GASTON CAMBON, *Gestion matérielle du parallélisme pour les architectures reconfigurables à grain épais*, Journées Francophones sur l'Adéquation Algorithme Architecture (JFAAA), janvier 2005, Dijon, France.

Architectures des accélérateurs de traitement flexibles pour les Systèmes sur puce

Les systèmes sur puce intègrent sur un même substrat de silicium l'ensemble des organes nécessaires à la prise en charge des différentes fonctionnalités du système. Pour la partie dédiée aux traitements numériques, le microprocesseur central est souvent déchargé des applications critiques (traitement du signal et des images en général) par un accélérateur de traitement. C'est par rapport à l'architecture du coprocesseur que se pose la problématique de cette thèse. En effet, de nombreuses approches sont possibles pour ce dernier, et vouloir les comparer s'avère être une tâche complexe. Après avoir fait un état de l'art des différentes solutions architecturales de traitement flexibles, nous proposons un ensemble de métriques dans une optique de caractérisation. Nous illustrons alors notre méthode par la caractérisation et la comparaison d'architectures représentatives de l'état de l'art. Nous montrons que c'est par une exploitation efficace du parallélisme que les coprocesseurs peuvent améliorer significativement leurs performances. Or, malgré de réelles aptitudes, les accélérateurs ne sont pas toujours capables de tirer parti de ce potentiel. C'est pour cela que nous proposons une méthode générale de multiplexage matériel, qui permet d'améliorer les performances par l'exploitation dynamique du parallélisme (boucle et tâches). Par son application à un cas concret, un système baptisé Saturne, nous prouvons que par l'adjonction d'un contrôleur dédié au multiplexage matériel, les performances de l'accélérateur sont quasiment doublées, et ce avec un faible surcoût matériel.

Mots-clés

Systèmes sur puce, Microprocesseur, DSP, Architecture Reconfigurable, Granularité, Traitement du Signal et des Images, Adéquation Algorithme Architecture, Métriques, Parallélisme, Multiplexage Matériel

Architectures of flexible processing accelerators for Systems on chip

The systems-on-chip integrate on a same silicon die the whole set of cores necessary to handle the various functionalities of the system. For the digital processing part, the central microprocessor is often discharged from the time consuming applications (generally, digital signal processing applications) by a processing accelerator. The thesis problematic stands-on the architecture of this coprocessor. Indeed, many approaches are possible for it, and comparing them is proved to be a complex task. After a state of the art of the various architectural solutions for flexible processing, we propose a whole set of metrics with a perspective of characterization. Then, we illustrate our method by the characterization and the comparison of architectures representative of the state of the art. We show that it is by an effective exploitation of parallelism that the coprocessors can improve significantly their performances. However, in spite of real aptitudes, the accelerators are not always able to benefit from this potential. From this observation, we propose a general method based on hardware multiplexing, allowing effective loop and task parallelism exploitation. By its application to a concrete case, a system named Saturn, we prove that by the addition of a controller dedicated to the hardware multiplexing, the performances of the accelerator are almost doubled, without hardware overcost.

Key words

Systems on chip, Microprocessor, DSP, Reconfigurable Architecture, Granularity, Digital Signal and Image Processing, Architecture Algorithm Mapping, Metrics, Parallelism, Dynamic Hardware Multiplexing

Discipline

Génie Informatique, Automatique et Traitement du Signal

Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM),
Université Montpellier II, 161 rue ADA, 34392, Montpellier Cedex 05, France

Nom du document : These_V4.2.doc
Dossier : C:\Documents and Settings\Pascal Benoit\Mes documents\Travail\Thèse\Itérations
Modèle : C:\Documents and Settings\Pascal Benoit\Application Data\Microsoft\Modèles\Normal.dot
Titre : UNIVERSITE MONTPELLIER II
Sujet :
Auteur : Pascal Benoit
Mots clés :
Commentaires :
Date de création : 09/11/2004 10:54
N° de révision : 3
Dernier enregistr. le : 09/11/2004 11:46
Dernier enregistrement par : Pascal Benoit
Temps total d'édition :49 Minutes
Dernière impression sur : 09/11/2004 11:49
Tel qu'à la dernière impression
Nombre de pages : 178
Nombre de mots : 50 047 (approx.)
Nombre de caractères : 285 270 (approx.)