# Gestion de l'activité et de la consommation dans les architectures multi-coeurs massivement parallèles

Gilles Bizot

UNIVERSITÉ DE GRENOBLE

**THESE**

Pour obtenir le grade de

**DOCTEUR DE L'UNIVERSITE DE GRENOBLE**

Spécialité: **Nanoélectronique et Nanotechnologie**

Arrêté ministériel : 7 août 2006


Présentée par

**Gilles BIZOT**

Thèse dirigée par **Michael NICOLAIDIS**
et codirigée par **Nacer-Eddine ZERGAINOH**

préparée au sein du **laboratoire TIMA**
et de **l'École Doctorale d'Électronique, Electrotechnique, Automatique et Traitement du Signal**


# Gestion de l'Activité et de la Consommation dans les Architectures Multi-Coeurs Massivement Parallèles

Thèse soutenue publiquement le **25 octobre 2012**,
devant le jury composé de:

**Monsieur Abbas DANDACHE**
Professeur à l'Université Paul Verlaine-Metz - LICM, Président/Rapporteur
**Monsieur Bruno ROUZEYRE**
Professeur à l'Université de Montpellier II - LIRMM, Rapporteur
**Monsieur Michael NICOLAIDIS**
Directeur de Recherche CNRS Grenoble - TIMA, Directeur
**Monsieur Nacer-Eddine ZERGAINOH**
Maître de Conférence à l'Université Joseph Fourier Grenoble - TIMA, Co-Directeur

# Remerciements

Je voudrais remercier M. Michael Nicolaidis et M. Nacer-Eddine Zergainoh de m'avoir donné l'opportunité de faire une thèse au sein du groupe ARIS. Je les remercie pour leur confiance, leurs conseils et pour toute l'aide qu'ils m'ont apporté durant ces quatre années.

Je remercie M. Abbas Dandache d'avoir accepté de présider et de rapporter mon travail de thèse. Je remercie M. Bruno Rouzeyre d'avoir accepté de rapporter mon travail de thèse et de sa patience.

Je remercie Mme Dominique Borrione, directrice du laboratoire TIMA, pour son accueil. Je tiens à remercier tout particulièrement M. Frédéric Rousseau, de m'avoir aidé et permis d'enseigner. Je le remercie pour sa sympathie et ses conseilles. Merci à M. Dimiter Avresky pour son aide, sa patience et sa compréhension.

Un grand merci à tous mes collègues et amis de bureau Claudia, Hai, Diarga, Yi, Saif, Seddik, Thong pour leur soutien et leur gentillesse. Je remercie tous mes collègues et amis de l'équipe ARIS Gilles, Wassim, Michael, Raoul, Vladimir, Fabien, Thierry, Ibou, JB, Paolo, Salma, Mohamed. Je remercie mes amis Hamayun et Shahzad (Ahmad) de l'équipe SLS.

Je remercie Corinne, Laurence, Sophie, Younes, Marie-Christine, Claire, Anne-Laure, Nicolas, Frédéric (Chevrot), Ahmed, Lucie pour leur gentillesse et leur bonne humeur. Je remercie aussi toutes les autres personnes du laboratoire que je n'ai pas cité.

Un grand merci à Séverine qui ma énormément aidé dans la relecture du manuscrit. Merci à toute ma famille, mes amis de m'avoir soutenue, encouragé et supporté durant les moments difficiles.

Je souhaite dédier cette thèse aux personnes qui me sont chères et qui ne sont plus là. A ma maman, qui nous a quitté en 2010. A mon papa, parti en 2011. A ma tante et à mon oncle, tous deux décédés en 2011.

# Table des matières

# Liste des figures

# Liste des tableaux

# Liste des algorithmes

# Première partie

# Introduction Générale

## Sommaire

Chapitre 1

# Introduction

**Sommaire**

Les fournisseurs de semi-conducteurs subissent une pression constante afin de réduire les coûts et les délais de fabrication. De plus, ils doivent fournir des circuits hautes performances avec de riches caractéristiques, tout en ayant une faible consommation énergétique. Les acteurs de l'industrie des semi-conducteurs s'adaptent et améliorent leurs techniques afin de répondre aux demandes des clients, suivant ainsi la loi de Moore. Cependant, durant ces dernières années un écart important s'est creusé entre le nombre de transistors disponibles sur une puce et la capacité des concepteurs à en faire un bon usage. Ainsi, certaines tendances se sont imposées :

- Modularité, réutilisation et parallélisme sont les mots d'ordres. La conception "from scratch" est trop coûteuse en temps. De plus, il est souvent plus efficace de déployer plusieurs instances d'un bloc de calcul existant que d'en recréer un nouveau plus puissant. Ce qui a conduit à une augmentation des librairies dites de "Propriété Intellectuel" ou IP (*Intellectual Property*), fondation du développement à base de plateformes.
- La complexité est de plus en plus déplacée du développement d'unités fonctionnelles vers les tâches d'intégration système. Ce qui est d'autant plus dommageable, du fait qu'il est aujourd'hui impossible de caractériser dans toutes les conditions de fonctionnement un système tout entier, menant ainsi, à des problèmes d'optimisations et de vérifications notables.
- Les outils logiciels permettant une conception assistée automatisée sont primordiaux, et ce à tous les niveaux. Ce qui inclut la caractérisation de performance, l'assemblage de la plateforme, la validation, etc.

De ces tendances ont résulté les MPSoC (*Multi-Processor System-on-Chip*) qui, aujourd'hui semble être devenu monnaie courante chez tous les industriels. Les MPSoCs sont composés de blocs fonctionnels variés (accélérateurs matériels, unités processeurs, mémoires) et intègrent, dans la plupart des cas, un système complet sur une seul puce. Ils sont utilisés dans divers applications, tel que multimédia, station de jeux (Sony PS3, Microsoft XBOX), *smartphones*, équipements automobiles, milieu médical, aérospatiale etc. Les MPSoCs sont principalement des assemblages d'IPs et reposent très fortement sur l'utilisation d'outils CAD (*Computer Aided Design*) que ce soit pour l'exploration initiale, l'optimisation, la vérification ou l'implémentation physique.

Nous constatons deux courants dans les MPSoCs, hétérogènes et homogènes, ayant tous deux leurs avantages et inconvénients. De façon synthétique, les MPSoCs hétérogènes sont composés de divers unités de calcul, qui de part leurs natures sont différentes, par exemple un processeur généraliste et un processeur DSP (Digital Signal Processor). A contrario, les MPSoCs homogènes possèdent une structure régulière ainsi que des unités de calculs qui sont toutes identiques. Les MPSoCs hétérogènes sont en générale considérés comme étant potentiellement plus performant, de part une certaine spécialisation. Cependant, leurs hétérogénéités ajoutent un niveau de complexité supplémentaire tant au niveau conception matériel que logiciel. Dans ce document, nous ne nous intéressons qu'au MPSoC homogènes, qui de part leur structure régulière et identique, permet une intégration matérielle plus aisée et scalable, tout en simplifiant l'intégration logiciel.

Depuis quelques années, le nombre de coeur de processeur intégré sur une même puce augmente.La transition de l'industrie des mono-puces simple coeur puis multi-coeurset enfin Many-Core, pose un véritable challenge à l'industrie logicielle. Comment les applications logicielles peuvent-elles continuer à récolter les bénéfices de performance de ces améliorations matérielles en ruptures, alors que la plupart des modèles et langages de programmation ont co-évolués dans un environnement monoprocesseur simple coeur ? En dépit d'une décennie d'avancées en informatique, la programmation des systèmes parallèles reste hors d'atteinte de la plupart des développeurs logiciels.

## 1.1 Contexte

Dans un premier temps nous donnons un brève aperçu de ce qu'est un système sur puce ainsi que sa *roadmap*, puis nous mettons en évidence ses futurs évolutions et tendances suivant l'ITRS (International Technology Roadmap for Semiconductor)[58], faisant référence en la matière.

### 1.1.1 Les systèmes sur puce

Le SoC (System-on-Chip) ou système sur puce couvre, aujourd'hui, une large part de la production de semi-conducteurs mondiale. Le SoC est une classe de produits et de styles de

conception qui intègre des technologies et des éléments d'autres classes de produits (microprocesseurs, mémoires embarquées, blocs analogiques et signaux mixtes ainsi que la logique reprogrammable) et couvre une large gamme de produits semi-conducteurs de grande complexités et à fortes valeurs ajoutées. Les technologies de conception et de fabrication pour les SoCs sont, à l'origine, typiquement conçues pour les marchés de grands volumes. Réduire les coûts de conceptions ainsi qu'une plus grande intégration sont les principaux objectifs . Du point de vue de la conception du SoC, le but est de maximiser la réutilisation de blocs ou de "coeurs" existants, ce qui inclut les coeurs personnalisés (custom) visant les grands volumes de production, les blocs analogiques, mais aussi les blocs issus de technologies logicielles. Les SoCs s'adressent à différents marchés (Mobile/grand public, Médicale, Réseaux et communications, Militaire, Bureautique, Automobile) qui ont chacun des exigences et des contraintes parfois très différentes.

### 1.1.2 Indicateur de tendance

L'ITRS (International Technology Roadmap for Semiconductor)[58] est une association internationale dont l'objectif est d'assurer la rentabilité (économique) des progrès en terme de performance des circuits intégrés et des produits qui les utilisent, permettant ainsi de perpétuer la "bonne santé" et le succès de cette industrie. L'ITRS fait office de référence en terme d'appréciation des futures exigences de l'industrie des semi-conducteurs et ce depuis plus de 15 ans. Ses estimations permettent aux industriels, laboratoires et universités du monde entier,de diriger leurs stratégies de recherche et développement.

Les produits SoCs sont classés en deux catégories, "mobile" et "stationnaire", avec des applications typiques tel que, respectivement, téléphone mobile et console de jeux. Ces deux catégories se distinguent, principalement, par leur consommation énergétique : les produits mobiles doivent avoir une consommation d'énergie minimale afin de maintenir leur alimentation par la batterie, alors que les produits stationnaire apportent une plus grande importance à une haute performance.

Les paragraphes suivant donnent un aperçu des futurs évolutions et tendances en terme de complexité, consommation d'énergie et de fiabilité.

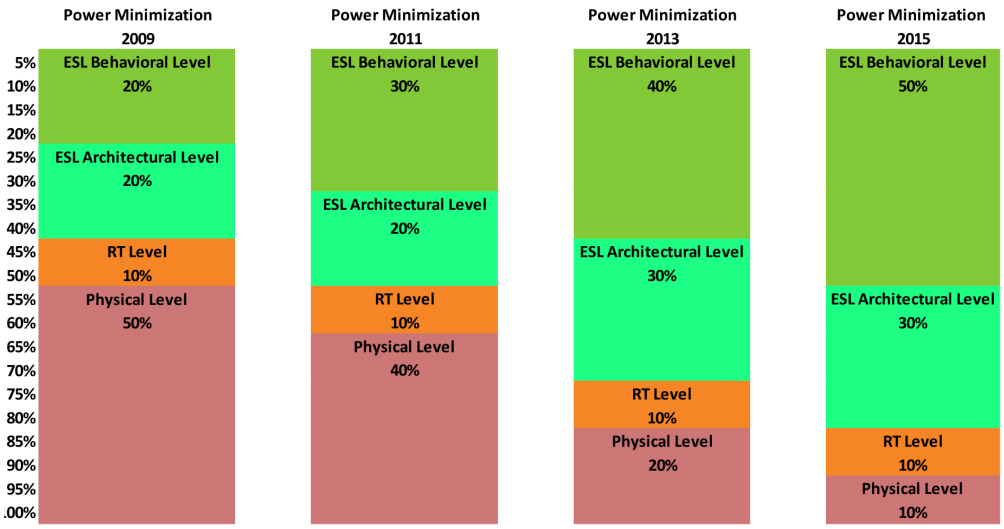



Figure 1.1 – Evolution du rôle des phases de conception dans la minimisation globale de la consommation des systèmes

La Figure 1.1 donne la *roadmap* concernant la progression du rôle de la conception au niveau système dans la perspective d'atteindre les exigences de minimisation de consommation énergétique du système. Dans cette figure, les valeurs en pour cent, indiquent la fraction de réduction de puissance qui devra être apportée à chaque phase de conception du système dans les futurs noeuds technologiques. Selon ces indicateurs, l'ITRS prévoit une progression importante du rôle tenue par la conception au niveau système dans sa participation à la réduction de la consommation d'énergie, avec plus de 50

#### 1.1.2.1 Conpléxité des SoC

La Figure 1.2 montre une estimation des tendances futures, en terme de complexité de conception des SoC mobiles. On peut constater que le nombre d'éléments de calcul ou PE (Processing Engine) augmente très rapidement dans les prochaines années, avec plus de 1000 PE prédit d'ici 10 ans. D'autre part, la quantité de mémoire principale augmente proportionnellement avec le nombre de PE.

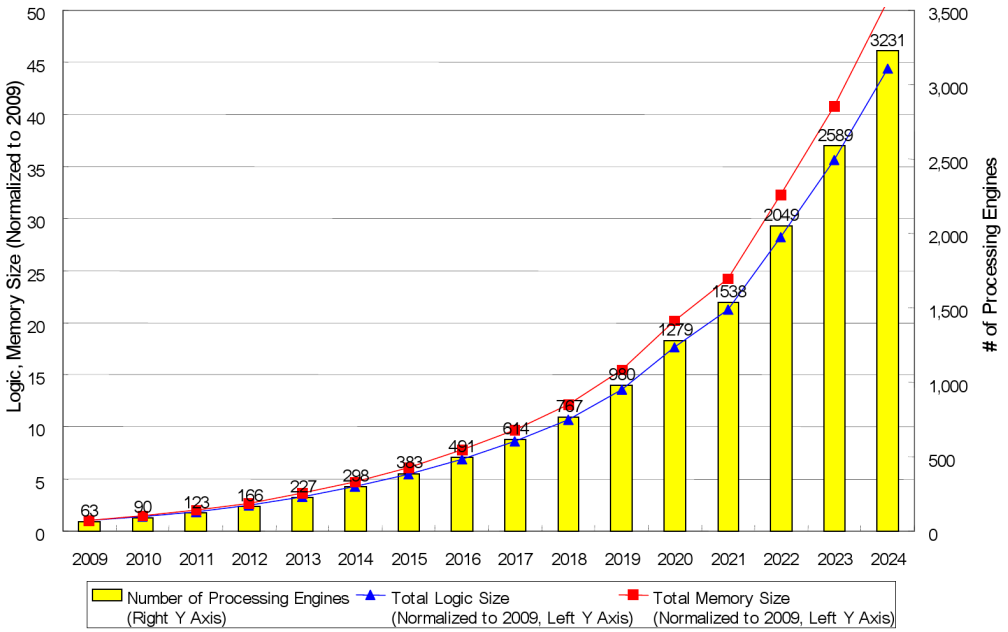



Figure 1.2 – Tendance de la complexité de conception des SoC

#### 1.1.2.2 Consommation énergétique

Alors que la complexité de conception est une tendance clé, la consommation d'énergie est aussi un facteur critique pour la conception de SoC dédiés aux produits mobiles. La Figure 1.3 montre la tendance en terme de consommation totale de la puce. Bien que l'IRTS signal une surestimation des ces valeurs, on peut néanmoins constater une augmentation croissante de l'énergie globale. De plus il est à noter que l'énergie (statique et dynamique) consommée par la partie mémoire représente une portion importante.

#### 1.1.2.3 Fiabilité et robustesse

Concernant la variabilité, certains paramètres, devront être contrôlés par le concepteur, incluant des paramètres au niveau processus de fabrication et au niveau circuit. Les paramètres, tel que la tension de seuil (incluant l'impact sur le dopage du canal, connu pour se réduire avec l'inverse de la surface) augmentera inévitablement conduisant à un point, potentiellement, critique dans les prochains dix ans. La propagation vers le haut de ces paramètres, du niveau processus,

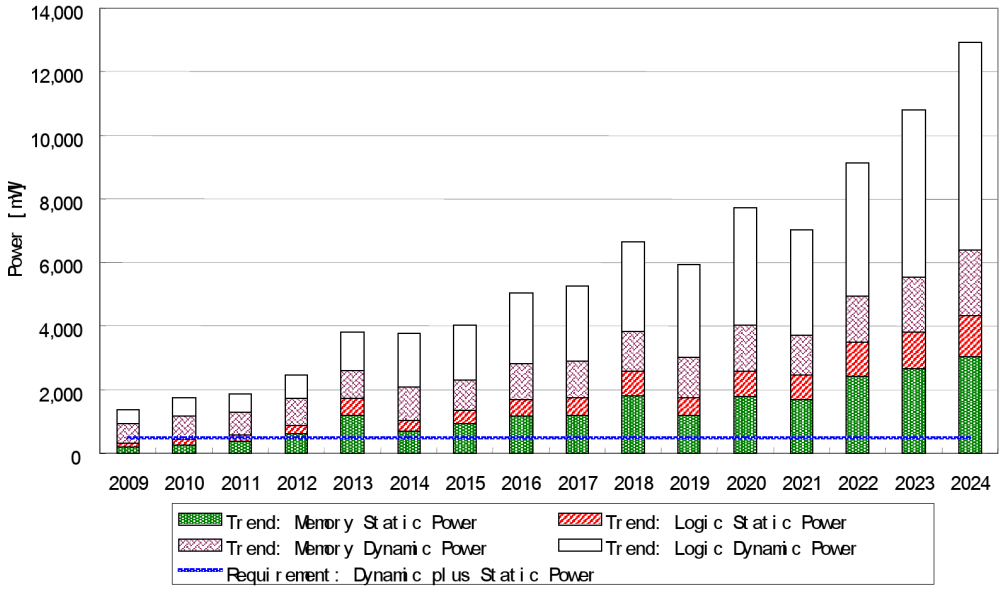



Figure 1.3 – Tendance de la consommation d'énergie des SoC

vers le niveau "device" puis au niveau circuit, obligera à adresser un très large ensemble de variabilités, que ce soit en terme de performance ou en terme de consommation. A moins qu'une nouvelle solution, radicale, ne soit trouvée.

Par conséquent, les frontières deviennent de plus en plus floues entre (i) les fautes catastrophiques, traditionnellement causées par des défauts de fabrication, et (ii) les fautes paramétriques, relatives aux variabilités du dispositif et de l'inter-connect. En effet, avec la variabilité qui augmente, les circuits peuvent manifester un comportement fautif, similaire à un défaut catastrophique. Par exemple, avec l'augmentation de la tension de seuil d'un transistor MOSFET d'une cellule SRAM, celle-ci peut apparaître "collée" à '1' et ne pas passer le test d'écriture. Ce type de phénomène est déjà présent dans les SRAM en 65nm. Les projections pour le futur, montrent que ce type de comportement va continuer, devenant plus probable, visant même à se propager aux "latch" et aux registres.

Les sources de défaillances peuvent être classées de la façon suivante :

(i) Variations de processus : Variations statistiques des paramètres des transistors tel que longueur de canal, tension de seuil et mobilité. La variabilité de la longueur du canal résulte de

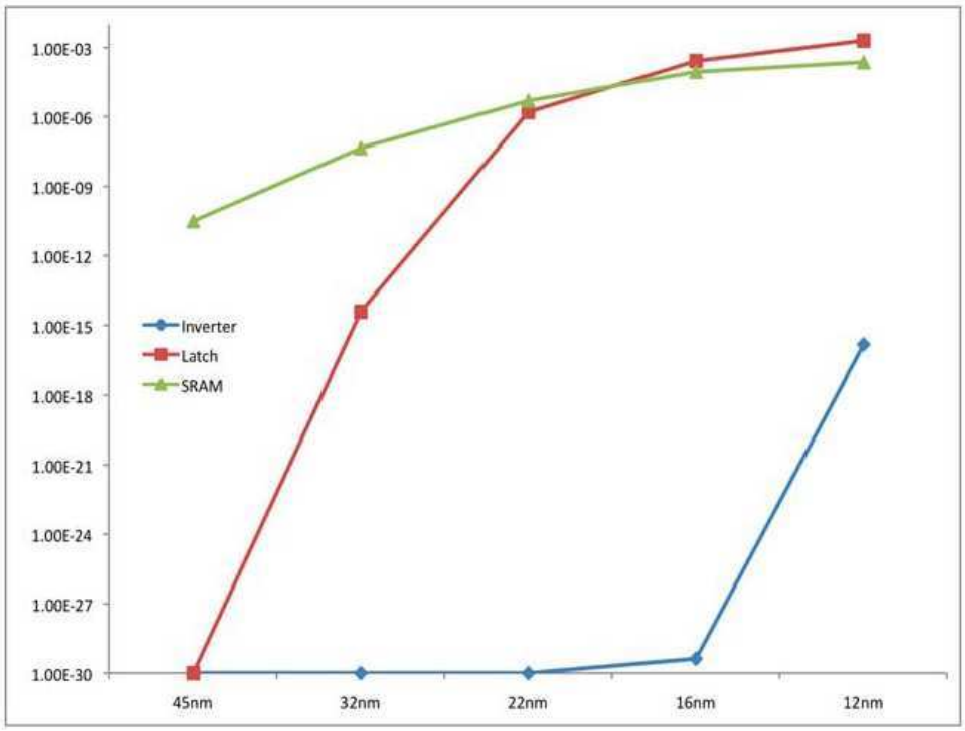



Figure 1.4 – Tendance du taux de défaut induit par la variabilité dans trois circuit typique.

la rugosité des bords de lignes ainsi que d'autre problème de fidélité des motifs, alors que les variations de la tension de seuil et la mobilité sont principalement induites par les fluctuations aléatoires des dopants.

(ii) Variations de la durée de vie : Variations qui changent les paramètres physiques au cours de la vie de fonctionnement du circuit. Les plus significatifs sont, d'une part, les dérives de $V_{th}$ dûes aux *negative-bias temperature instability* (NBTI) et aux *hot-carrier injection* (HCI), et d'autre part les dérives du courant de *gate* dû au *time-dependent dielectric breakdown* (TDDB).

(iii) Bruits Intrinsèque : Sources de bruits inhérentes au fonctionnement normal du circuit, devenant significatif aux tailles extrêmes.

Trois circuits canonique CMOS couramment utilisés : une cellule bit SRAM, un latch, et un inverseur. La Figure 1.4 montre la probabilité de défaillance pour ces trois circuits. On constate

que le latch atteindra, probablement, le taux de défaillance de la SRAM vers le 22nm.

## 1.2 Le projet ARAVIS

Cette thèse s'intègre au sein du projet ARAVIS (***A**rchitecture avancée **R**econfigurable et **A**synchrone pour **V**idéo et radio logicielle **I**ntegrée **S**ur puce*). Ce projet s'inscrit dans le cadre du pôle de compétitivité Minalogic. Il regroupe différent partenaires industriels et académiques du bassin Grenoblois tel que STMicroelectronics, CEA-LETI, INRIA-NECS, INRIA-SARDES et TIMA. Démarré en 2007, il s'est terminé fin 2011.

Celui-ci se propose d'apporter des solutions architecturales aux problèmes de la conception des plateformes de calculs pour les applications multimédia embarquées dans les technologies en limite de la feuille de route (*roadmap*) silicium. ARAVIS se propose de satisfaire conjointement les différents critères d'exigences que sont la flexibilité, la performance et la faible consommation d'énergie.

Les structures multiprocesseurs "Many-Cores", la technologie asynchrone matérielle et logicielle ainsi que la gestion dynamique de l'énergie et de l'activité, basées sur des techniques d'automatique avancées, sont les trois technologies clés qui, combinées, permettront de concevoir et de fabriquer un système sur puce répondant aux défis submicroniques. L'utilisation des techniques asynchrones permet le déploiement de structures fortement inter connectées sur d'importantes surfaces de silicium, tout en s'affranchissant des phénomènes de variabilité qui deviennent prépondérant dans les technologies en dessous de 32 nm. L'infrastructure logicielle, conçue et adaptée aux nouvelles technologies, prenant en compte les besoins de contrôle et de gestion de l'activité dans le circuit, afin de maîtriser la consommation et les dispersions de la technologie. Le système étant basé sur la mise en oeuvre de ressources matérielles dont les performances sont imprévisibles au moment de la fabrication, le recourt à une stratégie globale de gestion de la performance par adaptation du couple tension/fréquence se révèle nécessaire, afin de garantir les contraintes de temps réelles, tout en minimisant le budget énergétique.

Le SoC ARAVIS cible principalement (mais pas seulement) deux domaines d'application ayant un fort potentiel industriel et commercial : les applications dédiées à l'amélioration d'image et de vidéo ainsi que les applications de radio logicielles (*SDR : Software Define Radio*).

### 1.2.1 Présentation

L'architecture du SoC ARAVIS se présente sous la forme d'un système multiprocesseurs (many-core), organisé autour d'une topologie en maille 2D, dont les différents noeuds sont interconnectés par un réseau sur puce (Network on Chip) utilisant la logique asynchrone.

Chaque noeud ou *cluster* est constitué de 4 à 16 processeurs communiquant au travers d'une interconnexion locale, dont ils partagent la bande passante. De plus, une mémoire partagée localement au cluster servant, d'une part à la communication intra-cluster et d'autre part à la communication inter-clusters est présente.

Les coeurs de processeurs utilisés sont basés sur le XP70 de STMicroelectronics qui offrent de bonnes performances à un coût énergétique relativement faible. Le XP70 est un processeur de type RISC 32 bits à 7 étages de pipelines sur une architecture Harvard et offre de nombreuses possibilités de configuration matérielle tel que : compteur de boucles, contrôleur d'évènements, multi contexte, ainsi que la possibilité d'adjoindre des extensions spécialisées comme par exemple, des unités vectorielles. Avec une fréquence de fonctionnement nominale de l'ordre de 440Mhz (C065LP-SVT @1.2V), une consommation dynamique d'environ 65$\mu$W/Mhz et de 0.76 $\mu$W en statique (C065LP-HVT @1.2V), il est comparable aux processeurs ARM9 en terme de performance et aux ARM7 en terme de surface silicium.

Le SoC ARAVIS est une architecture GALS (Globally Asynchronous Locally Synchronous) avec, à l'intérieur des clusters, un fonctionnement synchrone mais communiquant et réagissant vers l'extérieur de façon asynchrone.

Afin de permettre la gestion de la consommation, un mécanisme de contrôle/commande de la tension d'alimentation et de la fréquence de fonctionnement est mis en place au niveau matériel sur chacun des clusters. Ainsi, un seul couple tension - fréquence est alloué pour l'ensemble des processeurs d'un même cluster. Le mécanisme de contrôle tension - fréquence choisit est de type saut de tension (Vdd-Hopping) qui permet un rendement plus élevé par rapport aux approches classiques avec convertisseur DC-DC et permet aussi de réduire de façon importante le coût en surface.

Le principe du VDD-Hopping, est de changer périodiquement la fréquence de façon tout-

ou-rien, c'est-à-dire que seul deux couples tension - fréquence sont disponibles, correspondant à une limite maximale et minimale. Ainsi en faisant varier le rapport cyclique entre fréquence haute et fréquence basse, il est possible d'obtenir une fréquence moyenne proportionnelle au rapport cyclique.

Concernant la gestion de l'activité, le placement et l'ordonnancement des tâches se fait de concert avec la gestion de la consommation. Tout ou partie de ces dernières pourront êtres réalisées de façon statique et donc prédéterminées à l'avance ou bien de façon dynamique, c'est-à-dire durant le fonctionnement.

Trois boucles de régulation imbriquées sont mise en place afin de gérer le compromis performance/consommation et la robustesse face aux phénomènes de variation des procédés de fabrication. La première, vise à réguler l'activité d'un noeud de calcul (cluster), la seconde à gérer l'activité au niveau circuit et enfin, la troisième, user intelligemment des ressources disponibles en exploitant les informations du système d'exploitation.

De plus, étant donné la complexité du circuit d'une part et des applications d'autres part, un nouveau canevas permettant une programmation plus aisée ainsi qu'une meilleure maîtrise des ressources processeurs, est utilisé. Celui-ci se base sur le modèle de composant Fractal, avec une implémentation en C, nommée Cécilia, développé au sein de l'INRIA-SARDES.

## 1.3 Contributions et Organisation du manuscrit

Dans ce travail de recherche, nous tentons de répondre à certains des nombreux défis auquel est, aujourd'hui, confrontée l'industrie des semi-conducteurs.

Nous étudions différentes approches, tantôt statique, tantôt dynamique, qui permettent le placement/ordonnancement d'applications et traitent les différents points clés de façon novatrice.

Dans un premier temps, nous proposons une méthodologie hybride (mixte statique - dynamique) originale et novatrice ciblant les architectures multiprocesseurs massivements parallèles (Many-Core). Celle-ci permet le placement et l'ordonnancement d'applications complexes et de grande taille de façon automatisée. Cette méthodologie prend en compte les variabilités du processus de fabrication, ce qui permet de réduire les pertes de performances et les "gaspillages" d'énergie potentielles.

Dans un deuxième temps, nous proposons une technique dynamique auto-adaptative, permettant de rétablir la (les) tâche(s) lors de défaillances, garantissant ainsi, la terminaison de l'application. Enfin, nous intégrons à cette technique, une stratégie entièrement dynamique, à l'exécution, permettant de répondre aux exigences de performances avec une prise en compte des variabilités tout en minimisant la consommation d'énergie.

Le manuscrit, est organisé en quatre grandes parties (Partie I à IV) dont nous détaillons le contenue chapitre par chapitre dans les paragraphes suivants.

La Partie I est composée de deux chapitres (Chapitre 1 et 2). Le premier, détaille le contexte et les motivations, ainsi que le projet fédérateur (ARAVIS) de ce travail. Le deuxième chapitre, fait un état de l'art sur les principaux points traités en relation avec les Systèmes sur puces, à savoir, la variabilité du processus de fabrication et ses impacts, la gestion/réduction de la consommation d'énergie et enfin la fiabilité/robustesse.

La Partie II, organisée en cinq chapitres (Chapitre 3 à 7), présente notre proposition de Méthodologie de Placement/Ordonnancement Multi-niveaux, tenant compte de la consommation d'énergie et de la variabilité technologique. Cette méthodologie, introduite de façon globale

dans le Chapitre 3, se décompose en trois étapes :

Etape 1 : *Placement/Ordonnancement Locale hors-ligne*. Réalisée au moment de la conception, celle-ci se focalise sur le comportement au niveau d'un cluster.

Etape 2 : *Placement/Ordonnancement Globale en-ligne*. Exploration de l'espace des solutions et conservation du front pareto. Chaque élément de ce dernier est un point de fonctionnement "temps/énergie", potentiellement utilisable dans l'étape suivante.

Etape 3 : *Placement/Ordonnancement à l'exécution*. Celle-ci se fait dynamiquement lors de l'exécution en décidant quel est le ou les point(s) de fonctionnement du front pareto offrant le meilleur compromis performance/consommation énergétique.

Les trois chapitres suivants (Chapitre 4 à 6) détaillent chacune des étapes précédemment citées. Le Chapitre 7 présente les expérimentations réalisées.

Dans le Chapitre 4, présentant l'étape 1, nous proposons deux[1] formulations ILP (Integer Linear Programming) afin de résoudre le problème de placement/ordonnancement localement au niveau d'un cluster. Cette méthode, permet d'obtenir une solution optimale. Cependant, lorsque le nombre de tâche augmente, le temps nécessaire à l'optimisation explose (complexité exponentielle) ce qui devient très vite inutilisable en pratique. Afin de pallier à cet inconvénient, nous avons utilisé diverse techniques d'ordonnancement basées sur des algorithmes de liste (*List Scheduling*). Ces derniers ont une complexité temporelle ($O(n)$) bien moins élevée et permettent d'obtenir des solutions relativement proche de l'optimale.

Le Chapitre 5 décrit la phase exploratoire, constituant l'étape 2. C'est dans cette phase que sont pris en compte les hétérogénéités dûes au processus de fabrication du SoC. Cette étape étant réalisée en ligne, c'est-à-dire sur le SoC cible, il est, de fait, possible d'obtenir les caractéristiques fréquentielles et énergétiques très proche des valeurs réelles. Afin de réaliser l'exploration des solutions, nous avons investigué l'utilisation d'algorithmes génétiques qui sont des algorithmes tout à fait adaptés aux optimisations multiobjectifs. L'exploration se fait au niveau du SoC tout entier et détermine quelles sont les solutions de placement/ordonnancement potentiellement intéressantes (au sens pareto) suivant nos deux objectifs qui sont le temps d'exécution

[1] L'une des formulations ILP propose l'utilisation de pipeline logiciel, tandis que l'autre non

et la consommation d'énergie. L'ensemble de ces points de fonctionnements potentielles sont sauvegardés et pourront être utilisés dans l'étape suivante.

L'étape 3, décrite dans le Chapitre 6, réutilise les résultats (points du front pareto) de l'étape 2. Cette étape est réalisée à l'exécution et doit décider quel est le jeux de points de fonctionnement à utiliser afin d'atteindre le meilleur compromis possible. La mise en oeuvre de cette étape, se base sur une heuristique utilisée dans certaines méthodes dite *scenario based*.

Les expérimentations ainsi que les résultats des différentes étapes sont exposées au Chapitre 7. De nombreux graphes ayant divers caractéristiques (types, tailles, CCR) sont testés sur les différents algorithmes de *List Scheduling* (*Static List Scheduling*, *Dynamic List Scheduling*, *pipeline List Scheduling*) et ce avec différentes tailles (2 à 16 processeurs) de *cluster*. De même, l'exploration des solutions est expérimentée avec différents facteurs de variabilité (écart type) sur plusieurs tailles de réseau allant de 9 à 1024 noeuds.

La partie III est constituée de deux chapitres (Chapitre 8 et 9) dans lesquels nous proposons des techniques auto-adaptatives, permettant de prendre en compte, durant l'exécution, les défaillances ainsi que les variabilités de performance et d'énergie.

Le Chapitre 8 présente, une approche hiérarchique de placement de tâches d'une application et permet de garantir la terminaison de celle-ci lors de défaillances d'un ou plusieurs processeurs. Cette approche est entièrement dynamique et s'effectue durant l'exécution de l'application. Lors d'un fonctionnement normal, c'est-à-dire sans défaillance, le placement des tâches s'effectue dynamiquement durant l'exécution au grès des besoins de l'application. Lorsqu'une défaillance survient, un mécanisme de détection, basé sur le modèle de "fail-silent" informe les tâches en relation avec la tâche défaillante. C'est alors, que se déclenche un autre mécanisme qui va se charger de rechercher un processeur disponible afin de recevoir la tâche défaillante pour y être re-exécutée. Que ce soit pour le "placement standard" ou le "replacement" (lors d'une défaillance), la stratégie de recherche d'un processeur disponible est la même, à savoir "le plus proche voisin d'abord" (*Nearby-Search* ou *Nearest Neighbor*). Il est à noter que cette approche ne prend en compte ni les aspects variabilité, ni les aspects consommation. Cependant, nous avons montrés, au travers de simulations, que notre approche permet d'exploiter un parallélisme important avec

un sur-coût négligeable.

Le Chapitre 9, présente une technique de placement de tâches prenant en compte les différents aspects motivant cette thèse. A savoir, les aspects performance, consommation, variabilité et défaillance. Cette technique reprend certains principes exposés dans le chapitre 8 précèdent, principalement la tolérance aux défaillances. La stratégie est une combinaison de *Nearest Neighbor* et de *First-Fit*. En se basant sur les besoins des tâches de l'application (durées des tâches, des communications,...) et sur les capacités de l'architecture (fréquence, charge de travail des processeurs, bande passante des liens de communications, énergie statique, énergie dynamique...) notre algorithme adapte sa zone de recherche en modifiant dynamiquement un critère d'arrêt (facteur de relaxation) ce qui permet de garantir le niveau de performance exigé par l'application en évitant les "mauvais" choix (dû aux variabilités tel que consommation excessive, fréquence processeur trop basse) tout en minimisant la consommation d'énergie. Nos simulations ont montré une réduction d'énergie de 5 à 20% suivant différents scénarios de variabilité.

Enfin, en Partie IV nous concluons ce manuscrit et exposons différentes pistes pour de futures explorations et améliorations.

Chapitre 2

# Placement et Ordonnancement d'applications : Etat de l'art

**Sommaire**

## 2.1 Introduction

Dans un système multiprocesseurs, trois questions se posent : qui ? où ? quand ?. C'est à dire, quelle tâche, sur quel processeur, à quel moment. Ceci correspond à : l'allocation, l'assignation et l'ordonnancement. Il est en général considéré seulement en deux étapes avec l'allocation et l'assignation regroupées au sein d'un même processus dit de placement ou *mapping*. Le placement consiste à déterminer quel processeur exécutera quelles tâches alors que l'ordonnancement détermine dans quel ordre celles-ci seront effectivement exécutées. Que se soit le placement ou l'ordonnancement, c'est deux problèmes sont NP-complet([43]), c'est-à-dire qu'il n'existe pas d'algorithme permettant de trouver une solution exacte en un temps acceptable. Par exemple, étant donné un programme et un système tous deux parallèles et comportant respectivement $N$ tâches et $K$ processeurs, l'espace d'exploration est au plus de $K^N$ combinaisons, ce qui devient très vite prohibitif en temps de calcul lorsque $N$ et/ou $K$ sont grands.

L'objectif de l'ordonnancement de graphe dirigés acycliques DAG (*Directed Acyclic Graph*) est de minimiser la date de terminaison globale d'un programme, en allouant convenablement les tâches aux processeurs et en organisant les séquences d'exécution des tâches. L'ordonnancement est réalisé de tel sorte que les contraintes de précédences entre les tâches soient respectées. La date de terminaison de l'ensemble d'un programme parallèle est communément appelée longueur d'ordonnancement ou *makespan*.

De plus, les variations du processus de fabrication des noeuds technologiques avancées, introduisent des différences significatives entre les coeurs d'une architecture multi-coeurs mono-puce, menant à des pertes de rendement. Les coeurs, dans une architecture many-core deviennent non opérationnels à cause d'effets directes des variations du processus ou indirectement de part la structure coeur - interconnect, défaillante à fournir une totale endurance aux défauts de la structure elle-même. En conséquence, les deux solutions permettant d'augmenter le rendement sont (i) réduire la fréquence d'horloge (de sorte que plus de coeurs commutent) et (ii) concevoir une structure d'interconnect tolérante aux fautes plus robuste (de sorte que moins de coeurs soient isolés). [95] préconise la réduction de la fréquence d'horloge comme une solution plus efficace pour augmenter le rendement des coeurs et le niveau de débit global des architectures

many-core basées sur un réseau en maille.

Les algorithmes de placement et d'ordonnancement multiprocesseurs ont été largement étudiés tout au long de ces dernières décennies et ce sous différentes perspectives. Dans la suite, nous exposons un état de l'art sur différentes technique de placement et d'ordonnancement prenant en compte la consommation d'énergie, la variabilité et la tolérance aux défaillances.

## 2.2 Prise en compte de l'énergie

Dans [86], les auteurs présentent une technique "consciente" de la consommation d'énergie qui permet, à la fois, d'ordonnancer des graphes périodiques *multi-rate* et des tâches apériodiques pour des systèmes distribués temps-réels embarqués. Les tâches périodiques du graphe sont à échéances strictes, alors que les tâches apériodiques peuvent être soit à échéances strictes, soit à échéances souples. Les graphes de tâches périodiques sont tout d'abord ordonnancés statiquement puis des emplacements (slots) sont créés, afin d'acceuillir les tâches apériodiques à échéances strictes. Les tâches apériodiques à échéances souples sont, quant à elles, ordonnancées dynamiquement à l'aide d'un ordonnanceur en ligne. Afin de permettre à l'ordonnanceur en ligne d'apporter, dynamiquement, des modifications, une certaine flexibilité est introduite au sein de l'ordonnancement statique puis optimisée. Ce qui permet d'améliorer le temps de réponse des tâches apériodiques souples au travers de la réclamation de ressources et du vol de temps creux, tout en maintenant la validité de l'ordonnancement statique. L'ordonnanceur en ligne utilise l'ajustement dynamique de la tension (DVS : Dynamic Voltage Scaling) et la gestion de l'alimentation afin d'obtenir un ordonnancement efficace en consommation d'énergie. Leurs expérimentations montrent des résultats relativement bon. La flexibilité introduite dans l'ordonnancement statique améliorerait de 43% le temps de réponse des tâches apériodiques souples. De plus, leur technique permettrait de réduire la consommation d'énergie de 63%.

Les auteurs de [49] présentent une technique de DVS, appelée *Adaptive Stochastic Gradient Voltage and Task Scheduling* (ASG-VTS) permettant de générer rapidement une solution efficace, quelque soit le nombre de modes de tensions (couple tension - fréquence) disponible. La technique proposée sélectionne les modes de tensions pour un jeu de tâches dépendantes map-

pées sur un système hétérogène. L'objectif est de minimiser la consommation d'énergie tout en assurant qu'aucune échéance ne soit transgressée. Un jeu de graphes périodiques représente l'application. Tous les graphes sont des DAG ayant la même période mais leur propre date d'arrivée et leur propre échéance. Ils utilisent une variante de recherche par gradient stochastique pour explorer différentes possibilités de distribution des temps creux. ASG-VTS distribue itérativement les temps creux. Lorsqu'un minimum locale est trouvé, ASG-VTS réclame, aléatoirement, une partie des temps creux précédemment alloués et recommence le processus de distribution. Initialement, le mode de tension le plus rapide est sélectionné, puis découle un nouveau mode de part la distribution de temps creux. Ensuite, le délai d'exécution et la priorité des tâches pour le mode généré, est calculé, suivant un ordonnancement à base de liste. Si, aucune échéance n'est enfreinte, alors l'ancien mode est remplacé par le nouveau pour la prochaine itération de distribution de temps creux. Sinon, le nouveau mode est soit écarté, soit conservé pour une phase de récupération de temps creux.

[60] présente l'algorithme *Energy-Aware Scheduling* (EAS). Cet algorithme EAS ordonnance statiquement les tâches de communication et de calcul sur une architecture NoC hétérogène et considère des contraintes temps-réel. L'algorithme assigne automatiquement les tâches sur différents PEs puis ordonnance leur exécution. Dans un mêmes temps, l'algorithme, prend aussi en considération le délai exacte de communication en ordonnançant les communications en parallèle. Les tâches de communication ainsi que les tâches de calcul s'exécutent en parallèle. Ils utilisent un réseau d'interconnexion en maille 2D avec un algorithme de routage XY, en soulignant le fait que leur algorithme peut s'adapter à d'autres architectures régulières ayant des topologies et des stratégies de routage différentes. Les auteurs présentent leurs algorithmes en trois étapes ; "Budget slack allocation" pour chaque tâche, "Level based scheduling" et "Search and repair". Ce dernier se décomposant en "local task swapping (LTS)" et "Global Task Migration (GTM)". EAS est donc un algorithme d'ordonnancement prenant en compte la consommation des communications inter processus. EAS considère des tâches non-préemptibles et peut être assimilé à une heuristique de liste. De plus, ils génèrent le routage des communications et implémentent le concept de "search and repair" en cas de dépassement d'échéance, ce qui peut complexifier le temps d'optimisation.

Dans [45], les auteurs ont proposé des améliorations de techniques existantes, basées sur DVFS et DPM et présentées dans [44]. De façon générale, deux approches DVS sont possible pour l'ordonnancement : inter-tâches et intra-tâches. La première, détermine la tension sur la base de tâches monolithiques, alors que la seconde, permet de sélectionner différentes tensions au sein même d'une tâche. Plus précisément, les auteurs présentent une méthode afin d'améliorer les performances d'algorithmes d'ordonnancement intra-tâches. Ces algorithmes exploitent les temps creux apparaissant lors de l'exécution dû à la différence entre la longueur du chemin d'exécution pire cas et le chemin d'exécution courant. De plus, c'est un algorithme d'ordonnancement DVS intrinsèquement "scenario-aware" permettant de réduire la consommation d'application temps-réelles.

Les auteurs de [5] proposent une approche faible consommation avec une exploration multiobjectifs de l'espace de placement sur une architecture NoC basée sur une topologie en maille 2D. Leur méthode est basée sur des techniques évolutionnistes (algorithme génétique), obtenant le placement pareto qui optimise la performance ainsi que l'énergie consommée.

Les auteurs de [122] présentent un algorithme d'ordonnancement prenant en compte les communications inter-processeurs dans un système multiprocesseur, exécutant une application composée de tâches dépendantes. Leur algorithme réduit l'énergie de l'ensemble du système en (i) réduisant les communications inter-processeurs globales et en (ii) exécutant certains calculs avec une sélection de la tension adéquate (nombre de cycles à la tension basse). Ils traitent l'assignation et l'ordre des tâches simultanément. Afin de minimiser l'énergie consommée, ils couplent leur approche heuristique de minimisation des communications avec un modèle ILP. En résumé, les auteurs présentent une heuristique d'ordonnancement couplée à un modèle ILP afin de réduire la consommation ainsi que les communications.

Dans [10] des extensions supplémentaires à [11] et [92] sont apportées. Celles-ci concerne la consommation d'énergie, basée sur un budget de consommation et sur l'ajustement de tension d'alimentation (*voltage scaling*) Ils utilisent une méthode basée sur la décomposition de Bender[1] Dans cet environnement, la méthode alloue les tâches aux processeurs et décide de leurs

[1]La décomposition de Bender ([9]) est une technique de programmation mathématique permettant de résoudre de larges problèmes de programmation linéaire, sous réserve qu'ils aient une structure spécifique en blocs. L'algorithme ajoute de nouvelles contraintes au fur et mesure qu'il progresse vers la solution. La décomposition de Bender est une

fréquences d'exécution en tant que problème maître, alors que le sous-problème ordonnance les tâches avec une durée fixée et une assignation statique des ressources. Ce travail est basé sur le mélange de différentes techniques d'optimisation de placement et d'ordonnancement à différents niveaux.

Dans [110], les auteurs proposent un algorithme statique efficace, permettant d'optimiser l'énergie consommée par les tâches de communications de systèmes à base de NoC, dont les liens sont à tension évolutive. Afin de déterminer une vitesse de lien optimale, l'algorithme proposé (basé sur une formulation génétique) explore globalement l'espace de conception du système, incluant l'assignation des tâches, le placement des tiles, l'allocation de chemins de routage et l'assignation des vitesses de liens. L'assignation des tensions de liens est réalisée hors-ligne. Partant d'un graphe d'application périodique temps-réel, l'algorithme assigne une vitesse de communication à chacun des liens, minimisant ainsi l'énergie consommée par le NoC, tout en garantissant les contraintes temps réelles. De plus, l'algorithme proposé permet de désactiver statiquement des liens, lorsqu'aucune communication n'y est alloué, réduisant ainsi la consommation statique, non négligeable. Ils utilisent trois algorithmes génétiques imbriqués afin d'explorer efficacement l'espace des solutions. Que se soit l'algorithme d'assignation, de placement de tiles ou d'allocation des chemins de routages, tous les trois sont basés sur des algorithmes génétiques. En ce qui concerne l'ordonnancement, il est effectué par un algorithme de liste, utilisant la mobilité des tâches afin de déterminer leurs priorités. La mobilité d'une tâche est définit comme étant la différence entre la date de démarrage ASAP (As Soon As Possible) et la date de terminaison ALAP (As Late As Possible).

Dans [36] les auteurs présentent un canevas appelé *Multi Objective Genetic Algorithm for hw-sw Co-synthesis of distributed embedded systems* (MOGAC), une technique de placement et d'ordonnancement multiobjectifs. MOGAC utilise un algorithme de liste basé sur les temps creux afin de générer un ordonnancement statique non-préemptif, des processeurs et des liens. Un algorithme d'ordonnancement (non-MOGAC) assigne une priorité à une tâche en relation avec la différence entre sa plus proche et sa plus lointaine date de démarrage possible. Les prio-

approche basée sur la génération de ligne, au contraire de la décomposition de Dantzig-Wolfe qui elle, est basée sur la génération de colonnes.

rités relatives des tâches dans différents graphes de tâches ou bien dans différentes copies du même graphe de tâche, sont basées sur les périodes et les échéances des différents graphes. L'ordonnanceur est capable de gérer les spécifications des systèmes embarqués, dans lesquels les graphes de tâches ont une période inférieur à leur échéance. Les auteurs ont proposé une approches basée sur un algorithme génétique multiobjectifs, permettant l'exploration d'ensemble pareto-optimale d'architecture. Ils ont considéré de riches attribus paramétriques pour l'optimisation des liens de communication. Les attribus utilisés sont les suivants : taille du paquet, consommation moyenne par paquet, temps de communication moyen et pire cas par paquet, le coût, le nombre de contactes, le nombre de broches requit et la consommation statique. De part la grande quantité de paramètres, l'ordonnancement s'en trouve meilleur et plus sensible. Cependant, la qualité du résultat dépend grandement des fonctions objectifs de l'algorithme et de leurs évolutions.

## 2.3 Tolérance aux Fautes

Une approche d'ordonnancement pour les applications de sécurité critique tel que les systèmes embarqués tolérants aux fautes est présenté dans [59]. Les processus, ainsi que les messages sont ordonnancés statiquement. Lors de multiples fautes transitoires, le processus à récupérer est ré-exécuté. Ils définissent le concept de récupération transparente, lorsque la récupération d'un processus n'affecte pas les opérations des autres processus. L'algorithme proposé pour la synthèse d'ordonnancement tolérant aux fautes peut supporter le compromis transparence/performance, imposé par le concepteur, et utilisé les informations d'occurrences de fautes afin de réduire le surcoût due à la tolérance aux fautes. L'application est modélisée comme une sorte de DAG nommé *Fault-Tolerant Conditional Process Graph* (FT-CPG). La plateforme est composée de $n$ noeuds connectés via un bus, mais il est souligné le faite que la technique peut être adaptée à d'autres type de liens de communications. L'algorithme d'ordonnancement est basé sur un heuristique de liste. De plus, l'algorithme présenté utilise une fonction de priorité basée sur le chemin critique partiel pour ordonné la liste prête.

[133] souligne l'impacte négatif du DVS sur la fiabilité des tâches et du système. Ainsi, les

auteurs considèrent le problème d'assignation des fréquences à un jeu de tâches temps réelles, afin de maximiser la fiabilité sous des contraintes de temps et d'énergie. Ils formulent le problème comme un problème d'optimisation non-linéaire et montrent comment en obtenir statiquement une solution optimale. Puis, proposent un algorithme dynamique (en ligne) qui détecte les achèvements prématurés et ajuste les fréquences des tâches durant l'exécution, afin d'améliorer la fiabilité de l'ensemble.

Dans le même esprit, [7] examine la gestion, statique et dynamique, d'énergie sensible à la fiabilité. Les auteurs, ciblent un jeu de tâches périodiques, avec comme objectif de minimiser la consommation d'énergie, tout en préservant la fiabilité. Ils proposent deux heuristiques basés sur l'utilisation des tâches. Ils développent aussi, une technique dynamique au niveau *job*. L'idée est de "wrapper" les tâches, afin de monitorer et de gérer les temps creux dynamiques de façon efficace dans un environnement sensible à la fiabilité. Leur technique incorpore des tâches/jobs de recouvrement dans l'ordonnancement, afin de préserver le besoin de fiabilité. Dans un mêmes temps, le reste des temps creux est utilisé pour sauvegarder l'énergie.

Des mêmes auteurs, [134] présentent une nouvelle approche appelée *shared recovery* (SHR) afin de minimiser l'énergie consommée au niveau système. L'idée principale de la technique SHR est d'éviter l'allocation hors-ligne de tâches de recouvrement séparées aux tâches unitaires, en assignant un bloc de recouvrement globale/partagé, pouvant être utilisé durant l'exécution, par n'importe quelle tâche. D'après les simulations, les auteurs comparent leur technique à une autre technique, nommée RA-PM (Reliable-Aware Power-Management), et constatent une réduction d'énergie supplémentaire de 35%

# Deuxième partie

## Méthodologie de Placement / Ordonnancement Multi-niveaux tenant compte de la consommation d'énergie et de la variabilité technologique

**Sommaire**

Chapitre 3

# Présentation générale

**Sommaire**

## 3.1 Introduction

Le problème auquel nous sommes confronté ici, est de réaliser le placement et l'ordonnancement d'une application sur une architecture multiprocesseurs massivement parallèle que l'on peut qualifier de *"Many-Core"*. L'application est potentiellement composée d'un très grand nombre de tâches dépendantes.

Deux contraintes principales, doivent être satisfaites. La première et plus importante, est de garantir un certain niveau de performance[1] pendant l'exécution de l'application. La seconde, est de réduire, dans la mesure du possible, au maximum la consommation énergétique. Afin de respecter ces contraintes, il est important de considérer tout source pouvant impacter l'un de ces deux facteurs.

Dans la suite de ce chapitre, nous présentons une méthodologie permettant, d'une part de réduire la consommation d'énergie, d'autre part de minimiser l'impact sur les performances de la variabilité induite par le procédé de fabrication. La cible architecturale étant les SoC Many-Core. Cette méthodologie se décompose en trois étapes, dont chacune traite une partie du problème principale en s'appuyant sur la hiérarchie et les différents niveaux de granularités inhérents à l'application et à l'architecture.

## 3.2 Représentation et partitions de l'application

Dans les applications réelles modernes, il est fréquent qu'un certains nombre de paramètres soient inconnu au moment de la conception. Par exemple, pour de nombreuses applications multimédia, tel que les décodeur mpeg2, le débit des données d'entrée n'est pas connu avec précision. De même, des chemins d'exécutions différents peuvent exister, comme entre le décodage d'images I (prédiction intra image) et celui des images P ou B (prédiction inter images). Ainsi, il est souvent fait usage de paramètres pire cas ou cas moyen issue de divers méthodes de prédictions souvent empiriques.

[1] Le terme performance est ici à prendre au sens large et dépend des besoin de l'application. Celle-ci peut être caractérisée par une limite à garantir (haute ou basse) tel que : échéance, débit, ou bien par une performance (au sens strict) à atteindre tel que temps d'exécution minimale, débit maximale...

Lors de la conception de l'application, certaines caractéristiques de l'architecture de calcul ciblée ne sont pas connue précisément du fait des variabilités induites par le procédé de fabrication. Ainsi, deux processeurs d'une même puce pourront avoir des fréquences de fonctionnement différentes. Il en va de même pour leur consommation d'énergie.

Afin de représenter une application parallèle, il est en général fait usage de graphe, ce qui permet de mieux appréhender son fonctionnement et les interactions qui s'y déroulent. Différents types de graphes sont possibles, ayant chacun des avantages et des inconvénients. Dans la suite, nous considérons des applications représentées sous la forme d'un Graphe Acyclique Dirigé (*DAG : Directed Acyclic Graph*)(c.f.B).

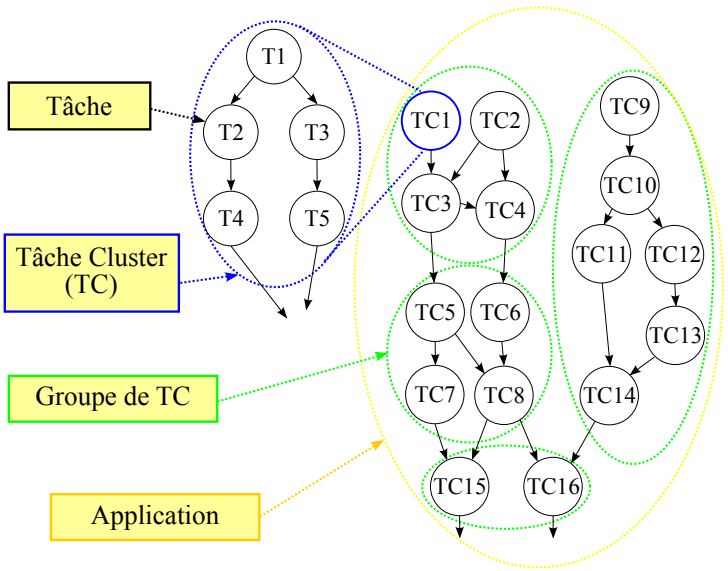



Figure 3.1 – Vue globale de l'application

Afin d'exploiter des architectures Many-Core, il est préférable et judicieux d'avoir une granularité d'application adaptée à l'architecture ciblée. Ainsi, le DAG est potentiellement composé d'un nombre de tâches élémentaires très important permettant de tirer, au maximum, partie de l'accélération parallèle offerte.

Au sein du DAG, un certain nombre de regroupement peuvent être effectués suivant un ou plusieurs critères. Ces critères peuvent être de différentes natures (fonctionnelle, performance, localité spatiale et temporelle des données, chemins d'exécutions, *use-cases*...), par exemples,

deux tâches élémentaires participant à la même fonctionnalité ou encore des tâches participants au même chemin d'exécution. Ainsi, des regroupements de tâches sont réalisés de façon automatisé ([50]) ou bien manuellement par le ou les concepteur(s). Cette étape de partitionnement/regroupement a un impact considérable sur l'optimisation du placement/ordonnancement de l'application. Il convient, donc, de la réaliser avec beaucoup de précautions. Nous considérons ici, deux niveaux de regroupements. Ces derniers sont détaillés dans la suite.

### 3.2.1 Partition de premier niveau

Chaque regroupement de tâches élémentaires constitue une partition que nous nommons *Tâche Cluster* (TC). Une TC est destinée à être exécutée sur un seul et unique cluster, c'est-à-dire qu'elle ne sera pas répartie sur plusieurs clusters en même temps. Il est à noter, que dans certains cas, par exemple pour des raison de performance, il pourrait y avoir plusieurs instances d'une même TC traitant des données différentes.

### 3.2.2 Partition de deuxième niveau

A leur tour, les TCs peuvent êtres rassemblées pour former des groupes de plus grande taille. Ces groupes de TC (GTC) sont destinés à être exécutées sur un ensemble de plusieurs clusters. Au sein de l'application, plusieurs groupes de TC peuvent coexister de différentes façons. Une première façon, pourrait être qualifiée de "configuration". En effet, certaines applications, comme par exemple, un *player video* qui permet le décodage de très nombreux standards audio et vidéo. Ainsi, différents groupes, chacun correspondant à un standard différent, permet de combiner ceux-ci sans remettre en cause leur structure interne. Une seconde façon, est en relation avec l'utilisation de *use-cases* (cas d'utilisation), ce qui sous entend un système où plusieurs applications peuvent êtres exécutées en même temps. L'idée des *use-cases* vient simplement du constat que certaines applications ne sont (en général) jamais exécutées au même moment. Enfin, une troisième façon est de considérer chaque groupe comme une accélération potentielle permettant de répondre à des besoins d'adaptabilité au contexte et à l'environnement de fonctionnement, l'idée étant de permettre un parallélisme adaptatif.

## 3.3 Placement/Ordonnancement en trois étapes

Afin d'exploiter la hiérarchie architecturale et applicative ainsi que la prise en compte des incertitudes au moment de la conception, la méthode que nous proposons se décompose en trois parties, représentées en Figure 3.2. La première étape consiste à placer et ordonnancer, localement à un cluster, les différentes tâches et communications composant chaque TC, indépendamment les unes des autres. Puis, dans la seconde étape, une exploration de différents placement/ordonnancement globalement au SoC est réalisé pour chacun des GTC. De même, les GTC sont traités indépendamment les uns des autres. De ces deux étapes, il est possible de collecter un certains nombre de points de fonctionnement et d'offrir une plage de sélection relativement largeEnfin, lors de l'exécution, suivant l'activité de l'application, il est possible de choisir rapidement les points de fonctionnements qui, en les composants, permettront de garantir les performances, ainsi que de réduire de façon importante la consommation d'énergie globale. Afin d'apporter plus de précision, nous détaillons ces trois étapes dans les sous-sections suivantes.


Figure 3.2 – Vue d'ensemble de la méthodologie composée de trois étapes.

### 3.3.1 Etape 1 : Placement/Ordonnancement Local

La première étape, s'attache à réaliser le placement et l'ordonnancement de tâches localement au cluster. Nous nous intéressons ici, qu'à ce qui se passe à l'intérieur d'un cluster et faisons l'hypothèse d'une variabilité locale homogène. De ce fait, tout écart par rapport aux valeurs nominales des paramètres (fréquence, consommation), impactera l'ensemble du cluster. Ainsi, par exemple, pour une solution de placement/ordonnancement donnée, l'augmentation ou la diminution (dû aux variabilités) de la fréquence de fonctionnement diminuera, respectivement augmentera la durée totale d'exécution mais en aucun cas ne remettra en cause la solution elle-

même. Il en est de même pour la consommation d'énergie statique (principalement les courants de fuites) et dynamique. Ainsi, il est possible de placer et d'ordonnancer des tâches localement à un cluster, indépendamment de la valeur réel des paramètres et d'en tirer un temps d'exécution et une consommation relativement aux nombre de cycles nécessaires aux calculs et aux communications.

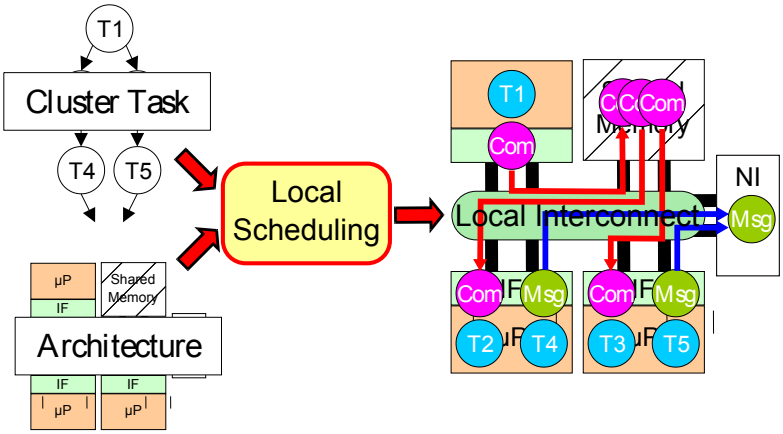


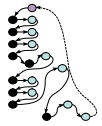

Figure 3.3 – Exemple d'ordonnancement de TC1 localement à un cluster

Ainsi, suivant le pré-partitionnement décrit en section 3.2, l'ensemble des tâches élémentaires et des communications de chaque *Tâche Cluster* sont placées et ordonnancées, indépendamment les unes des autres, de façon locale à un cluster générique (c.f. Figure 3.3).

Cette étape peut être vue comme une étape travaillant sur un niveau de granularité relativement faible, puisqu'elle traite les tâches élémentaires, qui est notre plus petite granularité. De cette étape, les différentes durées d'exécution peuvent être extraites puis une ou plusieurs machines à état peuvent être construite afin de séquencer les tâches et les communications (c.f. Table 3.1).

Bien que les caractéristiques (fréquences de fonctionnement et coefficients énergétiques) des clusters soient encore inconnues de façon précise, il est malgré tout possible d'estimer les temps d'exécution ainsi que la consommation statique et dynamique, en prenant pour référence les caractéristiques nominales.

Table 3.1 – Extraction des temps d'exécution, de l'énergie consommée et de la machine à état de chaque TC (Localement au cluster)

| TC | Execution Time | Consumed Energy | FSM |
|---|---|---|---|
| 1 | 22 | 50 |  |
| 2 | ... | ... | ... |
| 3 | ... | ... | ... |
| ... | ... | ... | ... |

### 3.3.2 Etape 2 : Placement/Ordonnancement Global

La deuxième étape, abstrait le niveau des tâches élémentaires et se situe au niveau de granularité supérieur qui est celui des TC. Les TC sont regroupées en partitions de TC (GTC), c'est donc sur celles-ci qu'intervient cette seconde étape. De la première étape, résulte une solution de placement/ordonnancement pour chaque TC de l'application sur un cluster générique. Chaque TC peut être alors être vu comme une tâche monolithique ayant une durée de calcul et de communication qui sont connues.

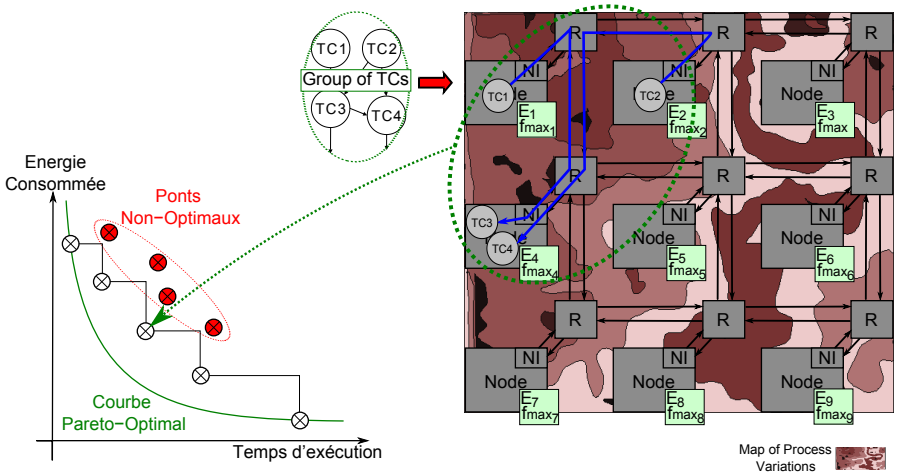



Figure 3.4 – Exemple d'ordonnancement Global de GTC1

Dans la seconde étape, représentée en Figure 3.4, les TC d'un même GTC sont placées et ordonnancées sur plusieurs clusters. Les différentes communications intervenant entre ces TC

sont, elles aussi, ordonnancées sur le NoC.

Dû à la variabilité du processus de fabrication, les paramètres des différents clusters ne pourront être connus de façon précise, qu'une fois la puce effectivement fabriquée. Pour cette raison, cette étape est réalisée sur la puce cible, c'est-à-dire en ligne. C'est une phase exploratoire, dans laquelle les valeurs de fréquence et de consommation sont connues et permettent donc de constituer différentes solutions de placement/ordonnancement exhibant chacune une performance et une consommation d'énergie différentes. Il est donc possible d'explorer l'espace des solutions de chaque groupe sur tout ou partie de la topologie 2D-mesh et d'obtenir un certain nombre de points de fonctionnements dans le plan : temps d'exécution - consommation d'énergie.

De l'ensemble des solutions trouvées, seules les plus optimales sont conservées et seront utilisées lors de la troisième étape.

### 3.3.3 Etape 3 : Placement/Ordonnancement à l'exécution

La troisième étape, est une étape de décision se déroulant lors de l'exécution. Celle-ci travaille au niveau des groupes de TC et abstrait leurs structures internes. Suivant les critères sur

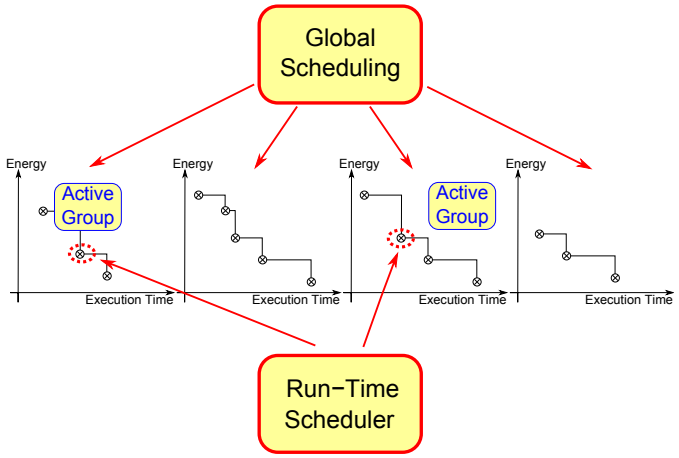



Figure 3.5 – Exemple d'ordonnancement à l'exécution

lesquels les partitionnements de l'application ont été fait lors de la conception, les différents GTC pourront être actifs ou non, et ce à différents moments d'exécution de l'application. L'activation ou la désactivation d'un GTC n'est à priori pas réductible. Cependant, grâce l'étape 2 qui

a explorée l'espace des solutions de chaque GTC séparément et ainsi constituée un jeu de solution potentielle, cela permet de pouvoir combiner différentes solutions de GTC et de sélectionner la meilleur combinaison. L'étape 3, a donc pour rôle de sélectionner, parmi leur jeu respectif, un point de fonctionnement (une solution) pour chacun des groupes actifs, afin de maximiser la performance à un coût énergétique minimale. La Figure 3.5 illustre cette étape avec quatre groupes dont deux sont actifs en même temps.

## 3.4 Illustration

Afin d'illustrer les propos de la Section précédente (Section 3.3), nous nous proposons de suivre le déroulement des différentes étapes au travers d'un exemple.

### 3.4.1 Application et Architecture cible

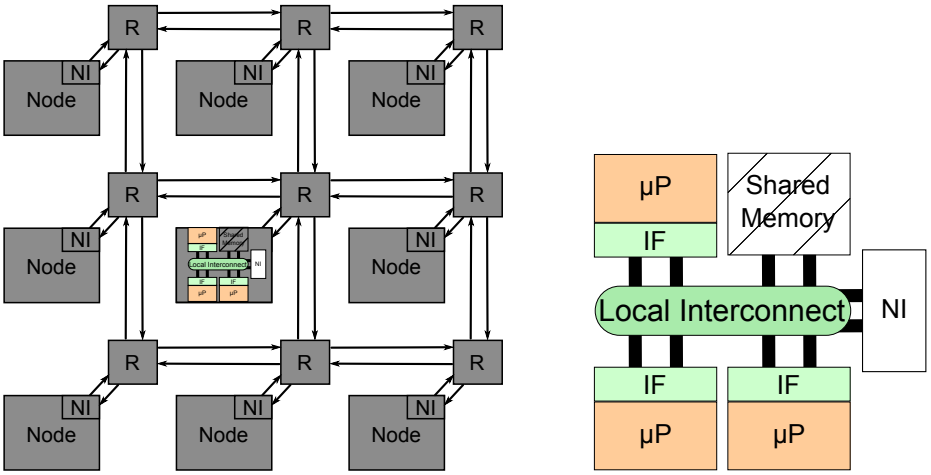
L'application utilisée pour l'exemple est celle représentée en Figure 3.1. Celle-ci est composée de quatre groupes de TC (GTC1 à GTC4) ayant chacun un nombre de TC variable, eux mêmes composées d'un certains nombre de tâches élémentaires.

L'architecture considérée est représentée en Figure 3.6. Dans un souci de lisibilité et de simplicité, le nombre de cluster et de processeur a été limité. Ainsi, l'architecture cible ne comporte que neuf clusters, chacun composés de trois processeurs et d'une mémoire locale partagée.

Nous ne détaillons de l'application que le contenu de GTC1, qui servira durant l'illustration. Représenté en Figure 3.7, le graphe de GTC1 est composé de quatre TC (TC1 à TC4) ayant chacune un nombre de 5 ou 6 tâches élémentaires, notées T$x$, ou $x$ est l'étiquette de la tâche. Chaque tâche et chaque communication sont annotées de leur coût, respectivement, en nombre de cycle d'exécution et en nombre d'octet à transférer.

### 3.4.2 Etape 1 : Placement/Ordonnancement local au cluster

Suivant l'étape 1, chaque TC est placée/ordonnancée sur un cluster générique. En Table 3.2, le résultat d'ordonnancement de TC1, où la date de démarrage ainsi que la date de terminaison de chaque tâche et de chaque communication (COM(Wx-y)) y est reportée. Ainsi, nous pouvons




(a) Réseau de noeuds en maille 2D (b) Zoom sur un cluster

Figure 3.6 – L'architecture ciblée

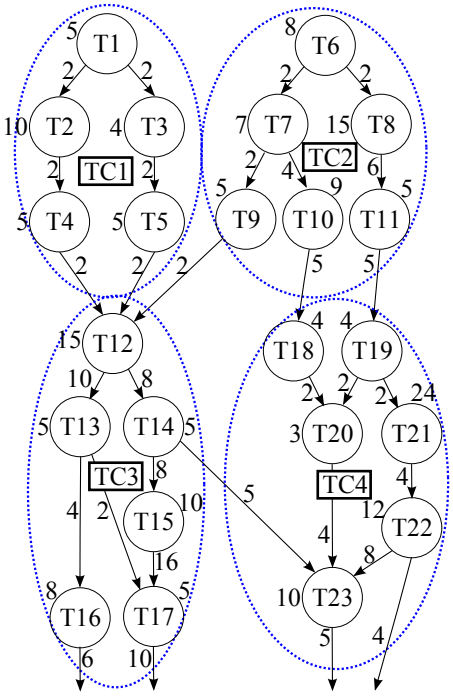



Figure 3.7 – Détail interne de GTC1. Composé de quatre TC (TC1 à TC4)

voir que la durée d'exécution de TC1 est donnée par T4 qui est la dernière à se terminer.

Table 3.2 – Table d'ordonnancement de TC1

| Task Name | Start Time | End Time |
|---|---|---|
| T1 | 0 | 5 |
| COM(W1-2) | 5 | 6 |
| COM(R1-2) | 6 | 7 |
| COM(W1-3) | 7 | 8 |
| T2 | 7 | 17 |
| T4 | 17 | 22 |
| COM(R1-3) | 8 | 9 |
| T3 | 9 | 13 |
| T5 | 13 | 18 |

### 3.4.3 Etape 2 : Placement/Ordonnancement Exploratoire

L'étape 2, est une phase exploratoire, réalisée globalement sur l'ensemble des clusters du SoC. Cependant, afin de conserver une certaine lisibilité dans cette illustration, nous n'exploitons que les clusters 1 à 4.

Une fois la phase exploratoire terminée, nous obtenons les graphiques décrit en Figure 3.8 pour les quatre GTC. Chaque graphique comporte deux courbes qui sont nommées, respectivement, "original" et "pareto". La première correspond à l'exploration à proprement parler. Quant à la seconde, elle est identique, si ce n'est que les points non-optimaux ont étés supprimés. C'est donc cette dernière (le front *pareto*) qui est conservée pour la suite.

### 3.4.4 Etape 3 : Choix des points de fonctionnement à l'exécution

Nous faisons, ici, l'hypothèse que l'application possède deux chemins d'exécution principaux[2] qui sont mutuellement exclusifs. La Table 3.3 décrit les points de fonctionnements choisis à l'exécution lors d'un changement d'activité. La première colonne correspond aux deux chemins d'exécutions, tandis que les colonnes suivantes (de gauche à droite) représentent, respectivement, les GTC actifs, le point de fonctionnement choisi, le placement sur les clusters, la durée d'exécution et l'énergie consommée. Les cellules colorées correspondent au coût globale (en temps et en énergie).

[2]Noté *Case 1* et *Case 2* dans la Table 3.3

Table 3.3 – Choix des point de fonctionnement lors de l'exécution

| Case | Active GTC | Point N° | Mapping | Time | Energy |
|---|---|---|---|---|---|
| 1 | GTC1 (TC1, TC2, TC3, TC4) | 3 | 3, 2, 4, 4 | 150.1 | 429.05 |
| | GTC2 (TC5, TC6, TC7, TC8) | 6 | 3, 4, 3, 4 | 110 | 279.52 |
| | GTC4 (TC15, TC16) | 3 | 4, 3 | 66 | 207.14 |
| | | | | 326.1 | 915.71 |
| 2 | GTC3 (TC9, TC10, TC11, TC12) | 19 | 4, 3, 3, 4, 4, 3 | 252.7 | 490.48 |
| | GTC4 (TC15, TC16) | 3 | 4, 3 | 66 | 207.14 |
| | | | | 318.7 | 697.62 |

## 3.5 Conclusion

Tout au long de ce chapitre, nous vous avons exposé notre méthodologie. Composée de trois étapes, celle-ci permet de traiter différents aspects du placement/ordonnancement tel que l'exploitation et la mise en correspondance des hiérarchies applicative et architecturale ainsi que la prise en compte des différents paramètres intervenant dans les phases de conception et d'exécution.

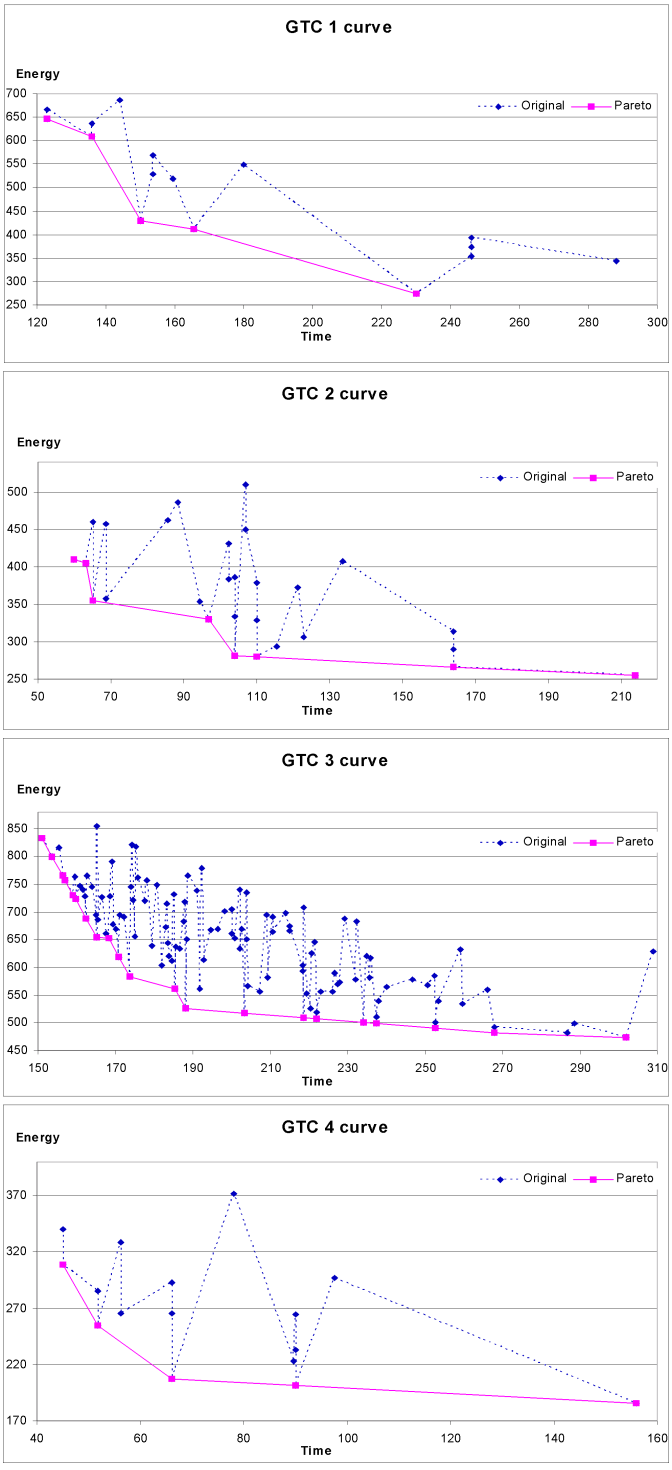



Figure 3.8 – Résultat d'exploration des différents GTC (GTC1 à GTC4)

Chapitre 4

# Placement/Ordonnancement Local hors-ligne

**Sommaire**

## 4.1 Introduction

Dans ce chapitre, nous présentons l'étape 1, qui a pour but de placer et d'ordonnancer les tâches élémentaires constituant une *Tâche Cluster* (TC). Dans un premier temps, nous proposons de résoudre le placement/ordonnancement d'une TC au sein d'un cluster, à l'aide de la Programmation Linéaire à coefficients entier (ILP : Integer Linear Programming) basée sur [118]. Nous décrivons deux formulations mathématiques (ILP) pour deux cas : avec et sans utilisation de pipeline logiciel. Cependant, le temps d'optimisation nécessaire aux méthodes ILP, est très important et devient vite prohibitif. C'est pourquoi, dans un deuxième temps, nous proposons l'utilisation d'heuristiques de listes afin d'accélérer le processus d'optimisation. De même, deux cas sont étudiés : avec et sans utilisation de pipeline logiciel. Mais avant cela, nous décrivons tout d'abord, le modèle de cluster utilisé tout au long de ce chapitre puis nous formulons le problème.

### 4.1.1 Modèle de Cluster

Le modèle de cluster utilisé est représenté en Figure 4.1. Il se présente sous la forme d'un système multiprocesseurs, où chaque processeur possède un lien de communication directe et dédié vers une mémoire locale. Chacune de ces mémoires est accessible par les autres processeurs via un bus partagé. L'accès à une mémoire peut être réalisé au travers du lien dédié d'une part et via le bus partagé d'autre part de façon simultanée. De plus, nous faisons l'hypothèse que les transferts de donnés entre les mémoires, via le bus partagé, sont réalisés par le biais d'un processeur de communication dédié (**DMA** : *Direct Memory Access*), afin de conserver la puissance de calcul des processeurs.

### 4.1.2 Formulation du problème

Le problème peut être formulé ainsi : étant donné un graphe de tâches et une architecture de cluster, réaliser le placement et l'ordonnancement des tâches et des communications de sorte que la durée totale d'exécution (de bout en bout) soit minimale. Le graphe de tâches de notre TC est constitué de $N$ tâches, notées $T_1, \ldots, T_N$. Associé à chaque tâche $T_i$, son temps d'exécution

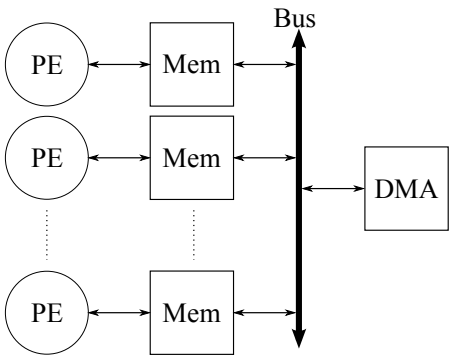
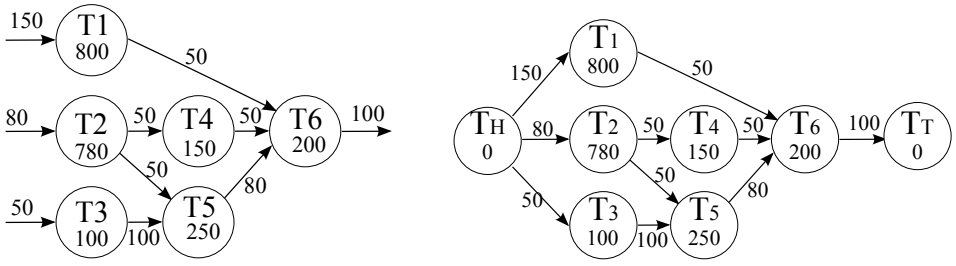



Figure 4.1 – Modèle de cluster utilisé

en nombre de cycles $w_i$. Les arcs $E_{i,j}$ représentent les communications entre les tâches $T_i$ et $T_j$. A chaque communication $E_{i,j}$ est associé $c_{i,j}$, la quantité de donnée à transférer. Le cluster est constitué de $M$ processeurs notés $P_1, \ldots, P_M$.

Une TC ne peut démarrer son exécution qu'une fois que toutes ses données d'entrée sont prêtes. De même, elle termine son exécution lorsque toutes ses données de sortie sont disponible. Ainsi, une tâche de durée d'exécution nulle, reliant toutes les tâches sans prédécesseurs, est ajoutée au graphe. De façon similaire, une tâche de durée d'exécution nulle, reliant toutes les tâches sans successeurs, est ajoutée au graphe. Ces deux tâches sont nommées, respectivement $T_H$ et $T_T$ (c.f. figure 4.2(b)).

Dans la suite, le graphe de la figure 4.2(a) est utilisé en guise d'illustration. Pour des raison de clarté et de simplicité, le cluster est supposé n'être constitué que de 4 processeurs.




(a) Graphe de TC. (b) Graphe de TC modifié.

Figure 4.2 – Exemple illustratif de graphe de TC.

## 4.2 Résolution par programmation linéaire

Dans cette section, nous présentons deux formulations permettant de résoudre notre problème. La première, nommée "Placement/Ordonnancement simple", optimise le temps d'exécution de bout en bout. La seconde, nommée "Placement/Ordonnancement avec pipeline", met en oeuvre l'utilisation de pipeline logiciel afin d'augmenter les possibilités de parallélisme et ainsi réduire le temps d'exécution.

### 4.2.1 Placement/Ordonnancement simple

L'objectif est de minimiser la durée d'exécution de la TC de bout en bout, c'est-à-dire minimiser la date de terminaison de la tâche $T_T$.

Soit la variable binaire de décision :

$$X_{i,j} = \begin{cases} 1 \text{ , si la tâche } T_i \text{ est placée sur le processeur} P_j \\ 0 \text{ , sinon} \end{cases} \tag{4.1}$$

$$\sum_{j=1}^{M} X_{i,j} = 1 \text{ , chaque tâche est allouée à un seul processeur} \tag{4.2}$$

Soit $start(T_i)$ et $end(T_i)$ les dates respectives de démarrage et de terminaison de la tâche $T_i$. Alors :

$$end(T_i) = start(T_i) + w_i \tag{4.3}$$

Soit $\Gamma^-(T_i)$ et $\Gamma^+(T_i)$, l'ensemble des tâches précédant, respectivement succédant, à la tâche $T_i$. La tâche $T_i$ ne peut commencer son exécution qu'après que l'ensemble de ses prédécesseurs aient terminé leur exécution (c-à-d : $\forall T_h \in \Gamma^-(T_i)$). De plus, si $T_i$ et $T_h$ sont placées sur des processeurs différents, alors $T_i$ doit attendre que le transfert de données venant de $T_h$ soit

terminé.

$$start(E_{h,i}) \geq end(T_h) + 1 \tag{4.4}$$

$$start(T_i) \geq end(E_{h,i}) + 1, \quad \forall h \text{ tel que } T_h \in \Gamma^-(T_i) \tag{4.5}$$

Lorsque deux tâches dépendantes sont placées sur le même processeur, alors le coût de communication est nul. Afin de refléter ceci, nous avons la contrainte supplémentaire :

$$end(E_{h,i}) = start(E_{h,i}) + L_{h,i} \times c_{h,i} - 1 \tag{4.6}$$

avec $L_{h,i}$ une variable binaire tel que $L_{h,i} = 1$ si et seulement si $T_h$ et $T_i$ sont placées sur des processeurs différents. Du fait que l'expression 4.6 n'est pas linéaire, nous linéarisons de la façon suivante :

$$\forall j \in [1, M] \quad L_{h,i} \leq 2 - X_{h,j} - X_{i,j} \tag{4.7}$$

$$\forall j \in [1, M] \quad \forall k \in [1, M],\ k \neq j \quad L_{h,i} \geq X_{h,j} + X_{i,k} - 1 \tag{4.8}$$

Les contraintes précédentes assurent que deux tâches dépendantes ne se chevauchent pas. Il est aussi nécessaire de garantir que deux tâches indépendantes, placées sur le même processeur, aient des exécutions disjointes. Ainsi, pour chaque paire de tâches indépendantes $T_i$ et $T_{i'}$, soit la variable binaire :

$$L_{i,i'} = \begin{cases} 1 \text{ , si } T_i \text{ et } T_{i'} \text{ sont placées sur des processeurs différents} \\ 0 \text{ , sinon} \end{cases} \tag{4.9}$$

Soit la variable binaire $B_{i,i'}$ tel que $B_{i,i'} = 0$ si $T_i$ et $T_{i'}$ sont placées sur le même processeur et que $T_{i'}$ s'exécute après $T_i$. De même, la variable binaire $B_{i',i} = 0$ si $T_i$ et $T_{i'}$ sont placées sur le même processeur et que $T_i$ s'exécute après $T_{i'}$. Nous assurons que deux tâches ont des

exécutions disjointes avec les contraintes suivantes :

$$B_{i,i'} + B_{i',i} - L_{i,i'} = 1 \tag{4.10}$$

$$start(T_i) \geq end(T_{i'}) - \infty \times B_{i,i'} + 1 \tag{4.11}$$

$$start(T_{i'}) \geq end(T_i) - \infty \times B_{i',i} + 1 \tag{4.12}$$

**Explication :** Dans l'Equation 4.11, lorsque $B_{i,i'} = 0$, alors le terme $\infty \times B_{i,i'}$ disparaît[1] et $start(T_i) \geq end(T_{i'}) + 1$, ce qui, correspond bien au fait que $T_{i'}$ s'exécute après $T_i$. Par contre, si $B_{i,i'} = 1$, alors $start(T_i) \geq -\infty$, signifiant simplement que $start(T_i)$ peut prendre n'importe quelle valeur. En effet, puisque $T_i$ et $T_{i'}$ sont indépendantes et allouées à des processeurs différents, aucune contrainte sur leur date de démarrage n'est nécessaire. Le même raisonnement est employé pour l'Equation 4.12.

Etant donné que toutes les communications ont lieu sur le bus partagé, nous devons garantir qu'elles ne se chevauchent pas les unes avec les autres. De façon similaire aux équations 4.10 à 4.12, pour toute paire de communications disjointes $E_{h,i}$ et $E_{f,g}$, nous avons les contraintes suivantes :

$$V_{h,i,f,g} + V_{f,g,h,i} = 1 \tag{4.13}$$

$$start(E_{h,i}) \geq end(E_{f,g}) - \infty \times V_{h,i,f,g} + 1 \tag{4.14}$$

$$start(E_{f,g}) \geq end(E_{h,i}) - \infty \times V_{f,g,h,i} + 1 \tag{4.15}$$

Avec $V_{h,i,f,g}$, la variable binaire telle que $V_{h,i,f,g} = 1$ si $E_{f,g}$ se situe après $E_{h,i}$, 0 dans le cas contraire. De même, la variable binaire $V_{f,g,h,i} = 1$ si $E_{h,i}$ se situe après $E_{f,g}$, 0 sinon. La figure 4.3 illustre le placement/ordonnancement optimal du graphe de TC donné en figure 4.2(a), en utilisant la formulation ILP avec quatre processeurs. Il est à noter, que seul deux processeurs sont utilisés dans cet exemple. En effet, utiliser plus de processeurs, impliquerait des communications supplémentaires, ce qui allongerait le chemin critique et donc la durée total d'exécution.

[1] Par convention : $\infty \times 0 = 0$ et $\infty \times 1 = \infty$.

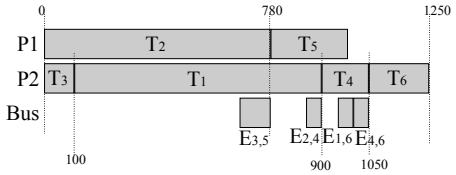



Figure 4.3 – Placement/ordonnancement simple

### 4.2.2 Placement/Ordonnancement avec pipeline

Dans cette section, nous étendons la formulation ILP précédente, afin de permettre l'utilisation de pipeline logiciel. Dans une exécution pipelinée synchrone, les tâches sont distribuées entre des étages de pipeline de longueur uniforme. L'intervalle d'initiation ou II (*Initiation Interval*) représente le temps d'exécution de l'étage de pipeline le plus long. L'objectif d'un ordonnancement pipeliné est de distribuer les tâches dans les étages, afin de minimiser l'intervalle d'initiation II, tout en respectant les dépendances entre les tâches et les contraintes de ressources. Chaque processeur peut être exploité dans un étage de pipeline séparé, car chacun s'exécute en parallèle. D'un autre côté, un étage peut exploiter plus d'un processeur, ce qui implique que le nombre maximale d'étage de pipeline ne peut être supérieur au nombre de processeur. La stratégie employée, est de placer et d'ordonnancer les tâches sur les processeurs de la même façon que pour un placement/ordonnancement sans pipeline (c.f. sec. 4.2.1), puis d'allouer les processeurs aux différents étages.

Soit la variable binaire de décision :

$$W_{j,s} = \begin{cases} 1 \text{ , si le processeur } P_j \text{ est alloué au } s^{ième} \text{ étage de pipeline} \\ 0 \text{ , sinon} \end{cases} \tag{4.16}$$

$$\sum_{s=1}^{M} W_{j,s} = 1 \text{ , chaque processeur est alloué à exactement un étage} \tag{4.17}$$

Il est à noter que le nombre d'étage de pipeline est implicitement définit à $M$, qui est le nombre maximal d'étage possible. Ceci est nécessaire afin de conserver une formulation linéaire. La solution peut, cependant, comporter des étages sans aucun processeur alloué. L'objectif est de minimiser l'intervalle d'initiation (II).

Soit $start(Stage_s)$ et $end(Stage_s)$, respectivement, les dates de début et de fin d'exécution de l'étage $Stage_s$. Alors

$$II \geq end(Stage_s) - start(Stage_s) + 1 \quad \forall s \in [1, M] \tag{4.18}$$

Les étages de pipeline ne doivent pas se chevaucher. Soit $B'_{s,t}$, la variable binaire tel que $B'_{s,t} = 1$, si l'étage $Stage_t$ s'exécute après $Stage_s$, 0 sinon. Similairement, la variable binaire $B'_{t,s} = 1$, si l'étage $Stage_s$ s'exécute après $Stage_t$ et 0 sinon. Ainsi, pour chaque paire d'étages $Stage_s$ et $Stage_t$, nous avons les contraintes suivantes :

$$B'_{s,t} + B'_{t,s} = 1 \tag{4.19}$$

$$start(Stage_s) \geq end(Stage_t) - \infty \times B'_{s,t} + 1 \tag{4.20}$$

$$start(Stage_t) \geq end(Stage_s) - \infty \times B'_{t,s} + 1 \tag{4.21}$$

La longueur de l'étage $Stage_s$ doit inclure la totalité de la durée d'exécution des processeurs alloués. Pour tous les étages $s \in [1, M]$ et tous les processeurs $j \in [1, M]$, il est nécessaire que :

$$start(Stage_s) \leq start(P_j) + \infty \times (1 - W_{j,s}) \tag{4.22}$$

$$end(Stage_s) \geq end(P_j) - \infty \times (1 - W_{j,s}) \tag{4.23}$$

où $start(P_j)$ et $end(P_j)$ sont les dates de début et de fin de l'exécution du processeur $P_j$. Celles-ci sont déterminées par la date de début de la première tâche et de fin de la dernière tâche placées sur le processeur. Par exemple, le temps d'exécution du processeur $P_4$ (c.f. figure 4.4(a)) commence avec le début de la tâche $T_3$ et se termine à la fin de la tâche $T_6$. Pour chaque processeur $P_j$, avec $j \in [1, M]$ et chaque tâche $T_i$, avec $i \in [1, N]$ :

$$start(P_j) \leq start(T_i) + \infty \times (1 - X_{i,j}) \tag{4.24}$$

$$end(P_j) \geq end(T_i) - \infty \times (1 - X_{i,j}) \tag{4.25}$$

Les communications doivent aussi êtres prises en compte dans les étages de pipeline. Contrai-

rement aux tâches, les communications prennent places sur le bus, qui sera utilisé dans tous les étages. Dans l'illustration de la Figure 4.4(b), les communications de différents étages de pipeline (c.-à-d. de différentes instances du graphe de TC) s'exécutent simultanément à l'intérieur d'un II. Les contraintes 4.13 à 4.15 assurent seulement que les communications à l'intérieur d'un étage (c.-à-d., d'une même instance de graphe) ne se chevauchent pas les unes avec les autres. Cependant, il est nécessaire de s'assurer que les communications ne se chevauchent pas entre les étages. Ceci est réalisé en normalisant l'intervalle d'exécution des communications. L'intervalle normalisé d'une communication est sa date de début, relativement à la date de début de l'étage de pipeline dans laquelle elle est allouée. Par exemple, l'intervalle de la communication $E_{2,4}$ en figure 4.4(a) est [800, 850]. Ce qui correspond à un intervalle normalisé de [0, 50] (relativement à la date de démarrage de l'étage 2).

$$F_{h,i,s} = \begin{cases} 1 \text{ , si } E_{h,i} \text{ est alloué à l'étage } Stage_s \\ 0 \text{ , sinon} \end{cases} \tag{4.26}$$

$$\sum_{s=1}^{M} F_{h,i,s} = 1 \text{ , chaque communication est allouée à exactement un étage} \tag{4.27}$$

Chaque communication est alors inclue dans l'intervalle de l'étage auquel elle est allouée.

$$start(Stage_s) \leq start(E_{h,i}) + \infty \times (1 - F_{h,i,s}) \tag{4.28}$$

$$end(Stage_s) \geq end(E_{h,i}) - \infty \times (1 - F_{h,i,s}) \tag{4.29}$$

Finalement, l'exclusion mutuelle pour chaque paire de communications $E_{h,i}$ et $E_{f,g}$ distinctes est requise.

$$\begin{aligned} (start(E_{h,i}) - start(Stage_s)) \geq \quad & (end(E_{f,g}) - start(Stage_t)) \\ & -\infty \times V'_{h,i,s,f,g,t} + 1 \end{aligned} \tag{4.30}$$

$$\begin{aligned} (start(E_{f,g}) - start(Stage_t)) \geq \quad & (end(E_{h,i}) - start(Stage_s)) \\ & -\infty \times V'_{f,g,t,h,i,s} + 1 \end{aligned} \tag{4.31}$$

où la variable binaire $V'_{h,i,s,f,g,t} = 0$ ($V'_{f,g,t,h,i,s} = 0$) si et seulement si $E_{h,i}$ est ordonnancée à l'étage $Stage_s$, $E_{f,g}$ est ordonnancée à l'étage $Stage_t$ et l'intervalle normalisé de $E_{h,i}$ est ordonnancé après (avant) l'intervalle normalisé de $E_{f,g}$.

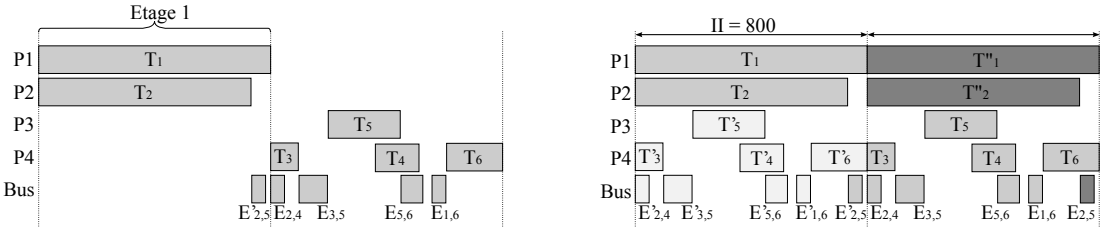
La linéarisation des équations 4.30 et 4.31 est réalisée de la façon suivante : $V'_{h,i,s,f,g,t} = 1$, ($V'_{f,g,t,h,i,s} = 1$) ssi $F_{h,i,s} = 1$ et $F_{f,g,t} = 1$ et l'intervalle normalisé de $E_{g,h}$ est ordonnancé après (avant) l'intervalle normalisé de $E_{h,i}$.

$$V'_{h,i,s,f,g,t} + (V'_{f,g,t,h,i,s} + F_{h,i,s} \geq 2 \tag{4.32}$$

$$V'_{h,i,s,f,g,t} + (V'_{f,g,t,h,i,s} + F_{f,g,t} \geq 2 \tag{4.33}$$

$$V'_{h,i,s,f,g,t} + (V'_{f,g,t,h,i,s} + F_{h,i,s} + F_{f,g,t} \leq 3 \tag{4.34}$$

$$\tag{4.35}$$




(a) Une instance d'exécution

(b) Pipeline en état quasi stationnaire.

Figure 4.4 – Ordonnancement pipeliné optimale de l'exemple de la figure 4.2(a)

## 4.3 Approximation par Heuristiques de Liste

Bien que les deux formulations ILP précédentes, garantissent une certaine optimalité, le temps nécessaire à l'optimisation est extrêmement long et devient très vite prohibitif. Nous proposons, donc, d'utiliser des heuristiques de listes afin d'accélérer le processus. Similairement à la résolution ILP, nous proposons de résoudre le problème de deux façons : avec et sans utilisation de pipeline logiciel.

### 4.3.1 Placement/Ordonnancement simple

Nous présentons dans la suite, deux algorithmes de listes couramment utilisés ; l'un basé sur des priorités statiques et l'autre basé sur des priorités dynamiques.

Les Algorithmes 4.1 et 4.2 présentent deux heuristiques de listes génériques. Ces deux formes permettent l'ordonnancement d'un graphe DAG $G$ sur un jeu de processeurs $P$. La différence, réside, principalement, dans la façon dont est constituée la liste de noeuds à ordonnancer. Dans le premier, nommé *Static List Scheduling*, l'ordre des noeuds est déterminé statiquement suivant une politique prédéterminée, alors que dans le second l'ordre est remis à jour dynamiquement, d'où le nom de *Dynamic List Scheduling*.

**Algorithme 4.1** Heuristique de liste statique générique

**Entrée:** A DAG $G = (V, E, w, c)$ and a set of processors $P$
**Entrée:** A schedule of $G$ on $P$
1: **procedure** SLSCHEDULING($G$, $P$)
   $SortedList \leftarrow$ SORT($G$, policy)
2: **for all** node $n \in SortedList$ **do**
3: $p \leftarrow$ SELECTPROCESSOR($P$, $n$)
4: Schedule $n$ on $p$
5: **end for**
6: **end procedure**

**Algorithme 4.2** Heuristique de liste dynamique générique

**Entrée:** A DAG $G = (V, E, w, c)$ and a set of processors $P$
**Entrée:** A schedule of $G$ on $P$
1: **procedure** DLSCHEDULING($G$, $P$)
2: $UnScheduledList \leftarrow V$
3: **while** $UnScheduledList \neq$ null **do**
4: $n \leftarrow$ CHOOSENODE($UnScheduledList$)
5: $p \leftarrow$ SELECTPROCESSOR($P$, $n$)
6: Schedule $n$ on $p$
7: Remove $n$ from $UnScheduledList$
8: **end while**
9: **end procedure**

Différentes méthodes permettent de constituer une liste de noeuds, afin de les ordonnancer à l'aide des précédents algorithmes de listes. Cependant, l'algorithme utilisé doit permettre de garantir que les différentes relations de précédences entre les tâches et les communications soient respectées. Parmi les algorithmes bien connues, on trouve le tri topologique, le tri par le calcul des *top level*, *bottom level* ou encore le *priority level* qui est en quelques sorte la somme des deux

autres. Ce dernier, revenant à choisir les tâches du graphe se trouvant sur le chemin critique. Nous ne détaillons pas plus ces algorithmes qui font partie des "standards" (c.f Annexe B).

### 4.3.2 Placement/Ordonnancement avec pipeline

Dans cette sous-section nous présentons brièvement notre approche, afin de résoudre le problème de placement/ordonnancement à l'aide de pipeline logiciel et d'un heuristique de liste.

Dû à "l'effet" pipeline, durant l'intervalle d'Initiation II, chaque processeur exécute une instance particulière d'une ou plusieurs tâches. Dans le mêmes temps, les communications (intra-cluster) entre les tâches d'une mêmes instance, prennent place sur le bus et ce pour plusieurs instances différentes. Ces communications peuvent potentiellement générer un trafic important. Afin d'en limiter les effets, le graphe de TC est au préalable partitionné en plusieurs parties, chacune devant être exécuté sur un même processeur. Chaque processeur, devient alors, un étage particulier du pipeline.

L'algorithme utilisé pour préparer les étages de pipeline est un algorithme de partitionnement de graphe multi-niveaux, le *k-way partitionning* [67], dont les différentes phases sont plus amplement détaillées en Annexe B.2. Une fois le partitionnement effectué, chaque processeur se voient allouer une partition pour être exécuter dans l'étage du pipeline. Afin d'ordonnancer les tâches et les communications à l'intérieur du pipeline, nous utilisons un heuristique de liste, inspiré de celui présenté dans [52]. Deux schémas de partitionnement sont utilisés : Le premier nommé *cut-partitionning* à pour objectif de minimiser le nombre de communications entre les différentes partitions. Le second, appelé *volume partitionning*, tente de minimiser le volume total des communications entre les partitions.

## 4.4 Conclusion

Dans ce chapitre, nous avons proposé deux approches permettant de résoudre le problème de placement/ordonnancement d'une TC sur un cluster de processeurs. Ces deux approches, avec et sans pipeline logiciel, ont été d'une part formulé mathématiquement sous forme ILP et d'autre part traité à l'aide d'heuristique de liste standard dont les résultats et la mise en oeuvre sont

présentés au Chapitre 7 suivant.

Chapitre 5

# Placement/Ordonnancement Global en ligne

**Sommaire**

## 5.1 Introduction

L'étape 2, comme nous l'avons énoncé dans le Chapitre 3, consiste à explorer l'espace des solutions de chaque groupe de TC (GTC) indépendamment les uns des autres.

Cette étape étant réalisée en ligne, des modules de mesures (capteurs) doivent être présent afin de prendre en considération les variations dû au procédé de fabrication. Nous faisons donc l'hypothèse que un ou plusieurs sous-systèmes matériel et/ou logiciel sont disponibles afin de mesurer et collecter les divers informations nécessaires tel que fréquence maximale de fonctionnemdictionnaireent, consommation d'énergie dynamique et consommation statique.

De plus, un sous-système DVFS/*Vdd-Hopping*([90], [69], [79]) présent sur chaque cluster, permet d'ajuster son couple tension d'alimentation/fréquence. Celui-ci possède plusieurs modes de fonctionnement, décrit en sous-section 5.3.1.1. Pour chaque mode, un sous-système de contrôle, permet d'appairer le couple tension d'alimentation ($Vdd$) et fréquence de fonctionnement. En effet, nous faisons l'hypothèse que la fréquence de fonctionnement est ajustée à la tension d'alimentation de sorte que la fréquence sélectionnée, soit la fréquence maximale possible pour un fonctionnement correct sans erreurs, ni rupture. Ainsi, chaque cluster à une fréquence maximale de fonctionnement et des caractéristiques de consommation énergétique qui lui sont propre et potentiellement différentes des autres clusters.

L'étape 2 est une phase exploratoire qui réalise une recherche multiobjectifs, dont les deux critères sont le temps d'exécution et l'énergie consommée. Nous avons choisi d'utiliser des algorithmes génétique, car ils sont relativement bien adaptés à la recherche multiobjectifs. En effet, leur capacité à "s'échapper" de minima locaux, leur permettent une meilleure exploration.

Dans la suite de ce chapitre, nous présentons dans un premier temps (c.f. Sec. 5.2), quelques éléments généraux sur l'optimisation multiobjectifs ainsi qu'une brève introduction aux algorithmes génétiques. L'Annexe C décrit plus en détails les algorithmes génétiques ainsi que divers techniques adaptées à la recherche multiobjectifs.

Dans un deuxième temps, nous présentons la recherche multiobjectifs tel qu'elle est utilisée dans l'étape 2. Les différents modèles de consommation d'énergie (c.f. Sec. 5.3.1) et de performance (c.f. Sec. 5.5) nécessaire à l'évaluation des fonctions objectifs y sont décrit. Enfin, nous

concluons ce Chapitre en Section 5.4

### 5.1.1 Formulation du problème

Le problème qui nous concerne est d'explorer l'espace des solutions de placement/ordonnancement d'un ensemble de TC, appelé Groupe de TC ou GTC sur la totalité de la plateforme. On veut pouvoir constituer un jeux de solutions afin de s'en servir pour prendre une décision ultérieurement. L'enjeu est double, d'une part obtenir une exécution minimale et d'autre part minimiser la consommation d'énergie. De l'exploration, un ensemble de solution est trouvé. Parmi celles-ci, un certains nombre ne sont pas "optimales" et donc, doivent être écartées. La résolution du problème est conforme à une optimisation multiobjectifs dans laquelle on cherche le front pareto.

## 5.2 Optimisation multiobjectifs et Algorithmes génétiques

Dans cette section,nous présentons brièvement quelques éléments de base concernant l'optimisation multiobjectifs. Tout d'abord, en donnant une définition puis,en expliquant les notions de dominance pareto et optimalité pareto. Enfin, une petite introduction aux algorithmes génétique est présentée.

### 5.2.1 Définition

De façon formelle, dans les problèmes d'optimisation multiobjectifs, un vecteur de valeur est affecté à un vecteur de variables de décision $X = [x_1, x_2, \ldots, x_z]$, ce qui constitue l'affectation. Chaque valeur est décrite dans un domaine donné, souvent appelé domaine de valeur. Une affectation est faisable, si elle répond à toutes les contraintes représentées par un vecteur $C = [c_1(X), \ldots, c_p(X)]$, elle est aussi appelée solution. Un vecteur de fonction métrique (ou fonctions objectifs) $M = [m_1, \ldots, m_q]$ est alors appliqué à la solution afin d'obtenir un point (ou valeur objectif) $[m_1(X), \ldots, m_q(X)]$.

### 5.2.2 Dominance pareto et optimalité pareto

Mathématiquement, les notions de dominance et d'optimalité sont formulées de la façon suivante :

**Définition 1** (Dominance pareto)**.** *Soit $a$ et $b$ deux solutions et $q$ le nombre de fonction métrique. Soit $u, v \in 1, \ldots, q$, les indices de deux fonctions métriques. Alors on dit que :*

- *$b$ domine $a$ relativement à la métrique $M$ :*
  $dom(b, a, M) \Leftrightarrow \forall u : m_u(b) \geq m_u(a)$ *et* $\exists v : m_v(b) > m_v(a)$
- *$b$ est égale à $a$ relativement à la métrique $M$ :*
  $equ(b, a, M) \Leftrightarrow \forall u : m_u(b) = m_u(a)$

**Définition 2** (Optimalité pareto)**.** *Soit $b$ et $a$ deux solutions. On dit que $b$ est une solution (pareto) optimale, si et seulement si aucune solution $a$, tel que $a$ domine $b$, n'existe. L'ensemble des solutions optimales est appelé front pareto.*

Dans notre cas, une solution étant caractérisée par son temps d'exécution et sa consommation d'énergie, il est tout naturel de la représenter comme un point dans un plan orthogonale, dont les axes des abscisses et des ordonnées représentent respectivement le temps d'exécution et la consommation d'énergie. L'espace des solutions possibles est donc inclus dans ce plan et il est ainsi possible de qualifier une solution par rapport à une autre en se basant sur les notions de dominance et d'optimalité (au sens pareto) d'une solution sur une autre.

La figure 5.1 illustre ceci, avec un système orthogonale où les axes des abscisses et des ordonnées représentent respectivement le temps d'exécution et la consommation d'énergie. Par exemple, considérant un point A de ce plan, il est possible de diviser le plan, relativement à A, en quatre quadrants, notés 1, 2, 3 et 4. Si B est dans le quadrant 2, alors $t_B > t_A$ et $e_B > e_A$ et le point B est dit dominé par A (ou bien A domine B). Réciproquement, si B se trouve dans le quadrant 4, alors A est dominé par B (où B domine A). Dans les cas où B est dans l'un des quadrants 1 ou 3, la domination d'un point par rapport à l'autre ne peut être déterminé, car les deux sont potentiellement des solutions intéressante, cela dépend juste de quel paramètre, du temps d'exécution ou de la consommation d'énergie, est préférable de privilégier.

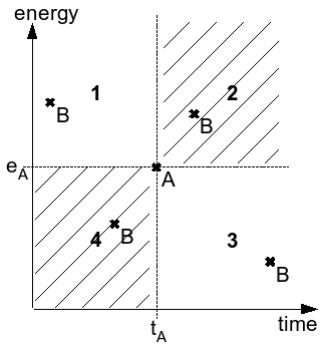



Figure 5.1 – Dominance des points de fonctionnements.

### 5.2.3 Brève introduction aux algorithmes génétiques

Les algorithmes génétiques ou *GA* ([46]) font partis d'une catégorie de méthode de recherche et d'optimisation imitant les principes de l'évolution des espèces et de la sélection naturelle. Les algorithmes génétiques travaillent sur une population d'individus, chacun représentant une solution au problème d'optimisation. Les GA font évoluer cette population afin d'en améliorer les individus à chaque nouvelle génération. En générale, un individu (solution) est représenté par une chaîne binaire appelée *chromosome* ou *génome*, cela correspond au *codage*. A l'initialisation, une population d'individus est générée, soit aléatoirement, soit à l'aide d'un heuristique, constituant ainsi la *population initiale*. Chaque chromosome de la population est alors évalué au travers de la fonction objectif, puis un facteur de qualité ou *fitness* lui est appliqué. La fonction *fitness* normalise les valeurs de la fonction objectif en les transformants en une mesure relative comprise en 0 et 1. Par exemple, la fonction fitness $F(x) = g(f(x))$ avec la fonction $g$ transformant les valeurs de la fonction objectif $f(x)$ en nombre non négatifs. Vient alors la phase de reproduction. Celle-ci se base sur la fonction fitness pour sélectionner, parmi la population, les "meilleurs" chromosomes qui se reproduiront. La reproduction est réalisée au moyen d'un opérateur de croisement ou *crossover* qui va échanger une ou plusieurs portions de deux chromosomes parents afin de créer un nouvel individu, potentiellement de meilleur qualité. Occasionnellement, une *mutation* est appliquée. Cet opération va venir altérer, le plus souvent aléatoirement, une par-

tie du patrimoine génétique de l'individu fraîchement créé. Ainsi, une nouvelle population est constituée et le processus est réitéré en réévaluant la population puis en appliquant les trois opérateurs (sélection, croisement, mutation) jusqu'à ce que le critère de terminaison soit atteint. La figure C.1 illustre les différentes phases typiques du processus d'exécution des algorithmes génétiques. Pour de plus amples informations, veuillez vous reporter en Annexe C, où un descriptif plus détaillé est présenté.

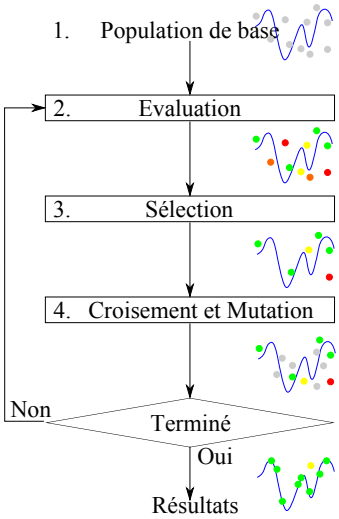



Figure 5.2 – Synoptique du flot de fonctionnement des algorithmes génétiques. L'algorithme commence par constituer une population initiale, puis évalue chaque individu de la population et sélectione les meilleurs individus afin de les reproduire en appliquant les opérateurs de croisement et de mutation. Ce processus se répète jusqu'à ce que le critère de terminaison soit atteint.

## 5.3 Exploration Multiobjectifs

Faisant l'hypothèse, que chaque cluster a la possibilité d'ajuster son couple tension d'alimentation/fréquence via un mécanisme DVFS/*Vdd-Hopping*, nous présentons les différents modes envisagés et décrivons une méthode de détermination de l'énergie résultante. Cette méthode permet, lors de l'exploration, d'évaluer l'énergie consommée.

### 5.3.1 Modèle de consommation d'énergie

Suite à l'étape 1 (Chapitre 4) sur chaque TC du groupe, le nombre de cycles nécessaire pour l'exécution de chaque TC est connu. Comme dit précédemment, nous faisons l'hypothèse que les différentes valeurs de consommations statique, dynamique, ainsi que la fréquence maximale de fonctionnement sont disponibles.

Ainsi, à chaque cluster est associé une fréquence maximale et deux coefficients de consommations d'énergie moyenne par cycle d'horloge. L'un pour la consommation statique, l'autre pour la consommation dynamique.

#### 5.3.1.1 Différents modes de fonctionnement

Nous présentons dans la Table 5.1 les différentes notations, qui seront utilisées dans la suite.

Table 5.1 – Liste des notations utilisées

| Notation | Description |
|---|---|
| $F_H, F_L, F_0$ | Fréquence High, Low, Clock-Off |
| $V_H, V_L, V_0$ | Tension d'alimentation High, Low, Power-Off |
| $P_H, P_L, P_0$ | Puissance instantanée à $(V_H, F_H)$, $(V_L, F_L)$, $(V_L, F_0)$ |
| $P_{DH}, P_{DL}$ | Puissance dynamique instantanée à $(V_H, F_H)$, $(V_L, F_L)$ |
| $P_{SH}, P_{SL}$ | Puissance statique instantanée à $V_H$, $V_L$ |
| $E_H, E_L, E_0$ | Energie totale consommée à $(V_H, F_H)$, $(V_L, F_L)$, $(V_L, 0)$ |
| $E_{DH}, E_{DL}$ | Energie dynamique consommée à $(V_H, F_H)$, $(V_L, F_L)$ |
| $E_{SH}, E_{SL}$ | Energie statique consommée à $V_H$, $V_L$ |
| $w_H, w_L$ | Nombre de cycles à $(V_H, F_H)$, $(V_L, F_L)$ |
| $\Delta_H, \Delta_L$ | Durée à $(V_H, F_H)$, $(V_L, F_L)$ |

Les formules génériques de détermination de la consommation statique et dynamique sont rappelées aux Equations 5.1 et 5.4. L'Annexe A, présente de façon plus détaillée la consomma-

tion dans les SoCs.

$$P_S \propto k_S.Vdd \tag{5.1}$$

$$E_S = P_S.\Delta t = k_S.Vdd.\Delta t \tag{5.2}$$

$$E_S(/cycle) = k_S.Vdd.T$$

$$= k_S.Vdd/f \ , T = 1/f \tag{5.3}$$

$$P_D \propto k_D.f.Vdd^2 \tag{5.4}$$

$$E_D = P_D.\Delta t = k_D.f.Vdd^2.\Delta t \tag{5.5}$$

$$E_D(/cycle) = k_D.f.Vdd^2.T$$

$$= k_D.Vdd^2 \ , T = 1/f \tag{5.6}$$

Comme nous l'avons suggéré précédemment, plusieurs modes de fonctionnement sont envisagés. Nous considérons, au sein de chaque cluster, les modes de fonctionnements suivant :

Table 5.2 – Les différents modes de fonctionnement

| mode | valeur couple $(V, F)$ | Energie(cpus) | mémoire |
|---|---|---|---|
| High | $(V_H, F_H)$ | $E_H = E_{SH} + E_{DH}$ | on |
| Low | $(V_L, F_L)$ | $E_L = E_{SL} + E_{DL}$ | on |
| Hopping | $\alpha F_H + (1-\alpha)F_L$ | $E_{hopping} = \alpha E_H + (1-\alpha)E_L$ | on |
| Idle | $(V_L, 0)$ | $E_{idle} = E_{SL}$ | on |
| Sleep | $(0, 0)$ | $E_{sleep} = 0$ | on |
| DeepSleep | $(0, 0)$ | $E_{deep} = 0$ | off |

#### 5.3.1.2 Détermination du mode de consommation

Après avoir placé une tâche sur un cluster, il est necéssaire d'en déterminer la durée d'exécution effective ainsi que la consommation d'énergie résultante. Pour se faire, le mode de fonctionnement doit être connu. Celui-ci est déterminé de la façon suivante :

Soit une tâche $T$, nécessitant $\omega$ cycles d'exécutions et soit $\Delta$, la durée d'exécution souhaitée.

Avec $\Delta_{min} = \frac{\omega}{F_H}$ et $\Delta_{max} = \frac{\omega}{F_L}$, repectivement, la durée minimale et maximale possible. Remarquons que $\Delta$ ne peut être $< \Delta_{min} = \frac{\omega}{F_H}$. Alors, deux cas de figure se présentent : (i) Soit $\Delta \geq \Delta_{max}$ ; (ii) Soit $\Delta < \Delta_{max}$.

#### 5.3.1.2.1 (Cas a) : $\Delta \geq \Delta_{max}$

Dans ce cas, le mode Idle est utilisé et $(\Delta_{eff},\ \Delta') = (\Delta_{max},\ \Delta - \Delta_{max})$. En effet, comme le

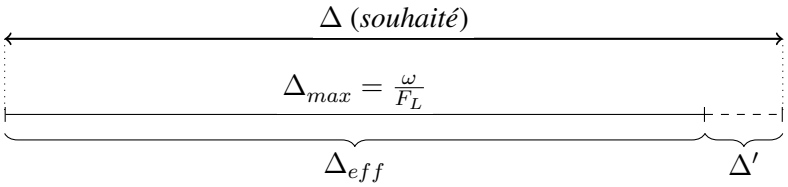



Figure 5.3 – Cas a : le mode Idle est utilisé

montre la Figure 5.3, la fréquence minimale de fonctionnement ne peut être inférieur à $F_L$, ce qui oblige à utiliser le mode Idle. La fréquence de fonctionnement est $F_L$ durant $\Delta_{eff}$ puis $F_0$ durant $\Delta'$. L'énergie consommée est alors :

$$E = P_L.\Delta_{eff} + P_0.\Delta' \quad (5.7)$$

$$= P_{DL}.\Delta_{eff} + P_{SL}.\Delta \quad (5.8)$$

$$E_{cyc} = \frac{E}{\omega} \quad (5.9)$$

$$= P_{DL}.\frac{\Delta_{eff}}{\omega} + P_{SL}.\frac{\Delta}{\omega} \quad (5.10)$$

$$= \frac{P_{DL}}{F_L} + \frac{P_{SL}}{\omega}.\Delta \quad (5.11)$$

#### 5.3.1.2.2 Cas b) : $\Delta < \Delta_{max}$

Dans ce cas, c'est le mode DVFS qui est utilisé et $(\Delta_{eff},\ \Delta') = (\Delta,\ 0)$ Dans le mode DVFS, le nombre de cycles exécutés $\omega$ peut-être divisé en deux, de la façon suivante :

$$\omega = \omega_H + \omega_L = \Delta.F_{moy} \quad (5.12)$$

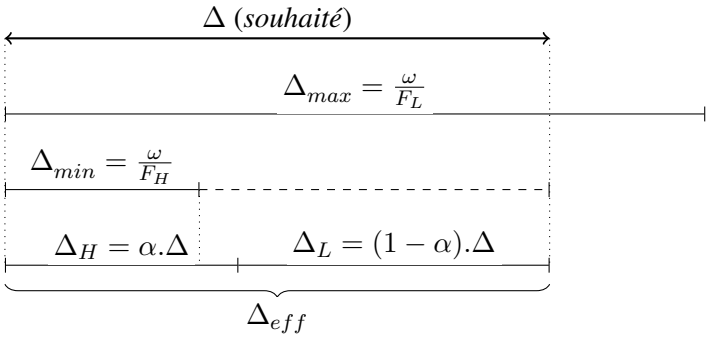



Figure 5.4 – Cas b : le mode DVFS est utilisé

, avec $F_{moy}$, la fréquence moyenne de fonctionnement sur $\Delta$. Chacun pouvant être obtenu par :

$$\omega_H = \Delta_H.F_H \tag{5.13}$$

$$\omega_L = \Delta_L.F_L \Rightarrow \Delta_L = \Delta - \Delta_H \tag{5.14}$$

Ainsi, on en déduit la fréquence moyenne :

$$\Rightarrow F_{moy} = \underbrace{\frac{\Delta_H}{\Delta}}_{\alpha}.F_H + \underbrace{\frac{\Delta - \Delta_H}{\Delta}}_{(1-\alpha)}.F_L \tag{5.15}$$

$$\rightarrow \alpha = \frac{1}{\Delta}.\frac{\omega - \Delta.F_L}{F_H - F_L} \tag{5.16}$$

D'où :

$$\Delta_H = \Delta.\alpha = \frac{\omega_H}{F_H} \tag{5.17}$$

$$\Delta_L = \Delta.(1-\alpha) = \frac{\omega_L}{F_L} \tag{5.18}$$

L'énergie consommée est alors :

$$E = P_H.\Delta_H + P_L.\Delta_L \tag{5.19}$$

$$= P_H.\alpha\Delta + P_L.\Delta.(1-\alpha) \tag{5.20}$$

$$= \Delta.(P_H.\alpha + P_L.(1-\alpha)) \tag{5.21}$$

$$E_{cyc} = \frac{E}{\omega} \quad (5.22)$$

$$= \frac{\Delta}{\omega}.(P_H.\alpha + P_L.(1-\alpha)) \quad (5.23)$$

$$= \frac{\Delta}{\omega}.(\alpha.(P_H - P_L) + P_L) \quad (5.24)$$

#### 5.3.1.3 Réseau sur puce

Nous faisons l'hypothèse d'un réseau en maille 2D à commutation de paquet de type *wormhole*. Un modèle d'énergie de router est proposé par Ye et al. dans [128]. Ainsi, nous nous basons sur modèle utilisé aussi dans [54], [24], [23]. L'énergie d'un bit ($E_{bit}$) est définit comme étant l'énergie dynamique consommée lorsqu'un bit de donnée traverse un router :

$$E_{bit} = E_{S_{bit}} + E_{B_{bit}} + E_{W_{bit}} \quad (5.25)$$

où $E_{S_{bit}}$, $E_{B_{bit}}$ et $E_{W_{bit}}$ représentent respectivement l'énergie dynamique consommée par le commutateur (switch) lui-même, les buffers et les fils d'interconnexion internes au switch. De plus, il convient de prendre en compte l'énergie consommée par les liens inter-routers. Ainsi, l'équation5.25 devient :

$$E_{bit} = E_{S_{bit}} + E_{B_{bit}} + E_{W_{bit}} + E_{L_{bit}} = E_{R_{bit}} + E_{L_{bit}} \quad (5.26)$$

$E_{L_{bit}}$ étant l'énergie dynamique consommée par un lien entre deux routers.

En conséquence, l'énergie moyenne consommée pour envoyer un bit de donnée d'un noeud $n_i$ à un noeud $n_j$ est :

$$E_{bit}^{n_i,n_j} = n_{hops}.E_{R_{bit}} + (n_{hops} - 1).E_{L_{bit}} \quad (5.27)$$

avec $n_{hops}$, le nombre de routers traversé par le bit. Ce qui donne

$$E_{flit}^{n_i,n_j} = nb_{bit/flit}.E_{bit}^{n_i,n_j} \quad (5.28)$$

pour un flit de donnée et une consommation moyenne pour un paquet de $k$ flits, égal à :

$$E_{paquet}^{n_i,n_j} = k.E_{flit}^{n_i,n_j} \tag{5.29}$$

Bien que ce modèle ne prenne pas en compte l'énergie consommée lors d'éventuelles contentions sur le réseau, il nous permet d'évaluer avec une relative précision l'énergie d'un message envoyé d'une source à une destination.

### 5.3.2 Modèle de performance

Soit deux tâches $T_a$ et $T_b$, communiquant $C_{ab}$ flits de $T_a$ vers $T_b$. Les poids de $T_a$ et $T_b$ sont respectivement $w_a$ et $w_b$. $T_a$ et $T_b$ sont "mappés" sur des noeuds ayant respectivement des fréquences de fonctionnement $f_a$ et $f_b$.

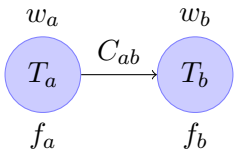



Figure 5.5 – Deux tâches $T_a$ et $T_b$, ayant une communication de $C_{ab}$ flits

#### 5.3.2.1 Liens homogènes – Sans congestion

Les durées minimales pour les tâches $T_a$, $T_b$ et la communication $C_{ab}$ sont données par :

$$\Delta_{C_{ab}} = \frac{C_{ab}}{B_l^{max}}, \quad \Delta_a = \frac{w_a}{f_a^{max}}, \quad \Delta_b = \frac{w_b}{f_b^{max}} \tag{5.30}$$

Avec $B_l^{max}$, $f_a^{max}$ et $f_b^{max}$, respectivement, la bande passante et les fréquences maximales de fonctionnement.

La durée minimale du triplé $T_a$, $T_b$, $C_{ab}$ sera de :

$$\Delta = \max\{\Delta_{C_{ab}}, \Delta_a, \Delta_b\} \tag{5.31}$$

D'où les fréquences de fonctionnement de $T_a$ et $T_b$ ainsi que la bande passante effectivement

utilisées par $C_{ab}$ :

$$B_{ab} = \frac{C_{ab}}{\Delta}, \quad f_a = \frac{w_a}{\Delta}, \quad f_b = \frac{w_b}{\Delta} \tag{5.32}$$

#### 5.3.2.2 Liens hétérogènes – Sans congestion

$$\Delta_{C_{ab}} = \frac{C_{ab}}{\min_{l \in path} \{B_l\}} \tag{5.33}$$

La bande passante globale du chemin entre $a$ et $b$ est limitée par le lien dont la bande passante est la plus faible.

#### 5.3.2.3 Liens hétérogènes – Avec congestion

$$\Delta_{com} = \max_{l \in path} \left\{ \frac{\sum_{C_{ij} \in l} C_{ij}}{B_l} \right\} \tag{5.34}$$

La bande passante globale du chemin sera limitée par le lien dont le rapport entre la quantité de données y transitant par sa bande passante est le plus grand.

$$\Delta = \max\{\Delta_{com}, \Delta_a, \Delta_b\} \tag{5.35}$$

### 5.3.3 Mise en oeuvre

Pour la mise en oeuvre de notre exploration des solutions, nous utilisons un algorithme génétique avec une implémentation de recherche multiobjectifs. Nous avons choisi l'algorithme NSGAII [35] qui est un algorithme réputé pour ses performances. Afin de garantir la diversité de la population, cet algorithme utilise la technique de *crowding distance*, dont nous détaillons le principe en Section C.4.2 de l'Annexe C. La fonction fitness se base sur l'approche "classement pareto" détaillé en Annexe C.3.3.

#### 5.3.3.1 Codage de la solution

Le codage de la solution est très classique, puisque nous avons opté pour un codage type "chaîne de bits". La solution est représentée par un tableau de chaîne de bits. Chaque chaîne de bits représente le *mapping* d'une tâche particulière. La chaîne de bits comporte deux parties x et y décrivant la position du noeud dans le réseau 2D mesh, sur lequel une tâche est mappée.

#### 5.3.3.2 Fonctions d'évaluations

Les fonctions objectifs (temps, énergie) sont évaluées séparément. La fonction objectif "temps" ou "performance" est évaluée au travers de l'algorithme 5.1 en utilisant le modèle décrit précédemment (c.f. Sec. 5.5). La fonction objectif "énergie" est évaluée de part l'algorithme 5.2 et utilise le modèle de la Section 5.3.1.

**Algorithme 5.1** Fonction d'évaluation de performance

```
1:  procedure EVALPERF
    /*Evaluation des performances des clusters.*/
2:      for all v in V do
3:          load(n_v) ← load(n_v) + w_v
4:      end for
5:      for all v in V do
6:          time(n_v) ← load(n_v)/F_H(n_v)
7:      end for
8:      max_node_time ← max_{v∈V}{time(n_v)}
    /*Evaluation des performances du NoC.*/
9:      for all e in E do
10:         p ← path(src(e), tgt(e))
11:         for all l in p do
12:             load(l) ← load(l) + w_e
13:         end for
14:     end for
15:     for all e in E do
16:         p ← path(src(e), tgt(e))
17:         for all l in p do
18:             time(l) ← load(l)/BW_l
19:         end for
20:         max_link_time ← max_{l∈p}{time(l)}
21:     end for

22:     expected_time ← max{max_node_time, max_link_time}
23:     return expected_time
24: end procedure
```

L'Algorithme 5.1 décrit le pseudo-code de la fonction d'évaluation de performance. Celui-ci

comporte deux parties qui évaluent, d'une part (ligne 1), la performance des clusters, et d'autre part (ligne 8), la performance du NoC. La complexité temporelle de cette algorithme est $O(V + EN)$, se décomposant comme suit : $O(V)$ pour la partie évaluation cluster et $O(NE)$ pour la partie NoC, où $V$ est l'ensemble des tâches, $E$ est l'ensemble des communications et $N$ est l'ensemble des noeuds. En effet, en utilisant une fonction de routage XY, le plus grand chemin possible est de longueur $N$ suivant X et $N$ suivant Y, donc $N$.

L'Algorithme 5.2 décrit le pseudo-code de la fonction d'évaluation de l'énergie. De la même façon, il comporte deux parties qui évaluent l'énergie des clusters (ligne 1) et l'énergie du NoC (ligne 10). La complexité temporelle de cette algorithme est $O(N)$.

**Algorithme 5.2** Fonction d'évaluation de l'énergie

```
1:  procedure EVALERNERGY
    /*Evaluation de l'énergie des clusters.*/
2:      for all n in N do
3:          Δ' ← expected_time − load(n/F_L(n))
4:          if Δ' ≥ 0 then
5:              node_energy ← node_energy + PT_H × load(n)/F_L + PS_L × Δ'
6:          else
7:              αΔ ← −Δ'/F_H − F_L
8:              node_energy ← node_energy + αΔ(PT_H − PT_L) + ΔPT_L
9:          end if
10:     end for
    /*Evaluation de l'énergie du NoC.*/
11:     for all l in L do
12:         link_energy ← link_energy + (P_flit(l) + PS(l))load(l)/BW(l)
13:     end for

14:     total_energy ← node_energy + link_energy}
15:     return total_energy
16: end procedure
```

## 5.4 Conclusion

Dans ce Chapitre, nous avons présenté les différents éléments constitutifs de l'étape 2 de notre méthodologie. Nous avons présenté les modèles de performance et de consommation d'énergie pour les noeuds et le réseau NoC. De plus, nous avons proposé une technique permettant la prise en compte de différents modes de fonctionnement (*Idle/DVFS*) au moment de la phase exploratoire, permettant ainsi de choisir s'il est plus avantageux de fonctionner en mode

*DVFS/Vdd-hopping* ou en mode *Idle*. Enfin, nous avons présenté la mise en oeuvre de la phase exploratoire en décrivant le codage des solutions ainsi que les fonctions objectifs sous forme de pseudo algorithmes. Les résultats d'expérimentation seront présentés au Chapitre 7 suivant.

Chapitre 6

# Placement/Ordonnancement à l'exécution

**Sommaire**

## 6.1 Introduction

La troisième étape, consiste à réaliser l'ordonnancement entre les différents GTC de l'application. Cette dernière étape est effectuée en fonctionnement, ce qui permet de s'adapter aux conditions d'exécution. En effet, les applications visées, possédant différents chemins d'exécution, (correspondant à des cas d'utilisations différents) les GTC la constituant ne seront pas tous actifs en même temps, voir même pour certains, pas du tout actif durant tout l'instance de l'application. Ainsi, suivant les différents choix qui pourront être fait, différents ordonnancements seront possibles. Suite aux étapes 1 et 2, nous disposons à présent d'un certain nombre (une par groupe de TC) de courbes pareto dont chaque point est associé à un ordonnancement particulier. L'étape 3 consiste donc à choisir de façon dynamique, au plus, un et un seul point parmi

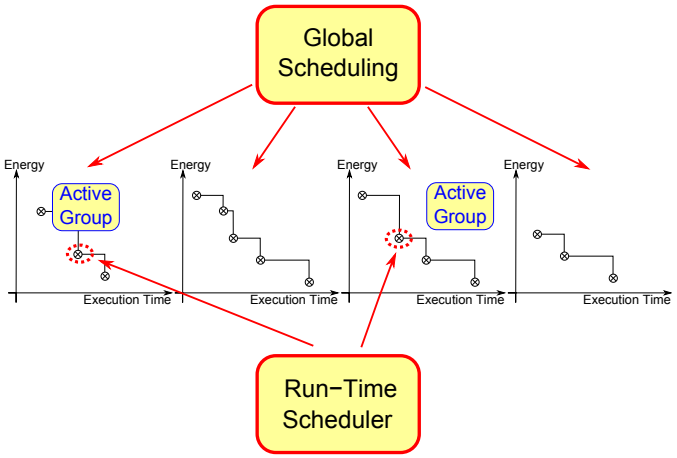



Figure 6.1 – Illustration de l'étape 3 : Choix des points de fonctionnements parmi les jeux de points des groupes actifs

chacune des courbes pareto des GTC actifs à un moment donné et à insérer les différentes communications nécessaires. Pour y parvenir, deux critères sont pris en considération. D'une part le temps d'exécution globale et d'autre part la consommation qui en découle. Ainsi, les points sont sélectionnés de sorte à respecter l'échéance globale tout en garantissant une consommation minimisée. La figure 6.1 donne un aperçu de l'étape 3, où quatre groupes de TC composent l'application. Chacun d'eux possède un jeu de point de fonctionnement pareto-optimale. Durant

l'exécution deux groupes sont actifs. Ainsi, un point de fonctionnement est choisi dans chacun des jeux des groupes actifs, de sorte que le temps d'exécution respecte l'échéance tout en minimisant la consommation d'énergie.

## 6.2 Formulation du problème

Le choix des points est un problème d'optimisation qui peut être formulé par un problème de "sac à dos à choix multiples" (MCKP : *Multiple Choice Knapsack Problem*) qui est connu pour être NP difficile. Celui-ci peut être formulé de la façon suivante :

$$\begin{aligned} maximize: & \sum_{i=1}^{k}\sum_{j=1}^{N_i} s_{ij}.x_{ij} \\ subject\ to: & \sum_{i=1}^{k}\sum_{j=1}^{N_i} t_{ij}.x_{ij} \leq D \\ & \sum_{j=1}^{N_i} x_{ij} \leq 1,\ 1 \leq i \leq k \\ & x_{ij} \in \{0,1\},\ i \in \{1,...,k\},\ j \in \{1,...,N_i\} \\ & s_{ij} = (e_{i0} - e_{ij}), s_{ij} \geq 0 \end{aligned} \tag{6.1}$$

Avec $D$ l'échéance globale, $k$ est le nombre de courbes pareto (égale au nombre de GTC actifs), $t_{ij}$ le temps d'exécution, $e_{ij}$ l'énergie du point $j$ du GTC $i$ et $N_i$ le nombre de point dans la courbe. $e_{i0}$ correspond à l'énergie du point le plus à gauche (dans la courbe) et donc au point où le temps d'exécution est plus faible et l'énergie la plus élevée. Ainsi, $s_{ij}$ peut être vue comme l'énergie préservée par rapport au point le plus consommant. $x_{ij} = 1$ dénote la sélection du point $j$ de la courbe $i$, donc $\sum_{j=1}^{N_i} x_{ij} \leq 1$ signifie qu'au plus un point de la courbe est sélectionné. Ainsi, le problème peut être décrit comme : *maximiser l'énergie sauvée en respectant l'échéance D*. Afin de minimiser le temps d'exécution de l'ordonnanceur en ligne, nous avons opté pour un algorithme décrit dans [127]. Celui-ci est un heuristique "glouton" relativement rapide pour prendre en considération de façon relativement efficace des applications de grande taille.

## 6.3 Heuristique de résolution

L'heuristique suivante (algorithme 6.1) est basée sur un algorithme glouton proposé dans [127] et [88]. Celui-ci propose un heuristique permettant de répondre au MCKP et de trouver une solution suffisamment bonne en un temps aussi faible que possible pour une taille de problème donné. De plus, l'algorithme proposé est constructif, améliorant la solution à chaque itération et peut donc être arrêté à tout moment, en cas par exemple de risque de dépassement du temps alloué à l'ordonnanceur en ligne. L'Algorithme 6.1 se décompose en deux phases, la phase d'initialisation et la phase itérative.

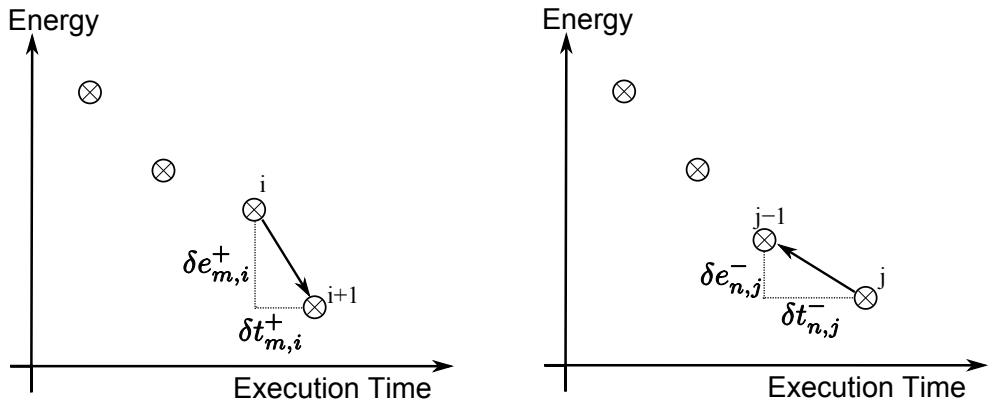



Figure 6.2 – Illustration de la phase d'initialisation et la phase itérative de l'Algorithme 6.1

Chaque point $i$, de la courbe pareto $m$, est représenté par deux paramètres : le temps d'exécution ($t_{m,i}$) et l'énergie consommée ($e_{m,i}$).Dans la phase d'initialisation, l'écart sur $t$ et $e$ est calculé suivant si le déplacement se fait sur la droite (du point $i$ à $i+1$) ou sur la gauche (du point $i$ à $i-1$), ainsi que la pente ($slope$) résultante. Les signes "+" et "-" en exposant indiquent la direction, respectivement, vers la droite ou vers la gauche. La solution initial est trouvée en ligne 12, 13 et assignée à la courbe, avec $s_m$, correspondant à une portion de *Deadline* $D$, proportionnelle au temps d'exécution de son point le plus à gauche. Ce qui garantit qu'une solution valide pourra toujours être trouvée pour la courbe. Lorsque la solution initiale est trouvée, une stratégie "à la volé" est utilisée. La différence,entre le temps assigné à la courbe $m$ et le

**Algorithme 6.1** Heuristique glouton

**procedure** INITIALISATION
  $slack = 0$
  **for all** curve $m$ **do**
    **for all** point $i$ in curve $m$ **do**
      $\delta e^+_{m,i} = e_{m,i} - e_{m,i+1}$
      $\delta e^-_{m,i} = e_{m,i-1} - e_{m,i}$
      $\delta t^+_{m,i} = t_{m,i+1} - t_{m,i}$
      $\delta t^-_{m,i} = t_{m,i} - t_{m,i-1}$
      $slope^+_{m,i} = \delta e^+_{m,i} / \delta t^+_{m,i}$
      $slope^-_{m,i} = \delta e^-_{m,i} / \delta t^-_{m,i}$
    **end for**
    $sm = t_{m,0} \frac{D}{\sum_{l=0}^{k-1} t_{l,0}}$
    search for maximal $j$ with $t_{m,j} \leq (sm + slack)$
    update $slack$
  **end for**
**end procedure**
**procedure** ITERATIVE IMPROVEMENT
  step 1 :
  sort $slope^+$ descendingly and $slope^-$ ascendingly
  **for all** curve $m$ in $slope^+$ **do**
    **for all** curve $n$ in $slope^-$ and $m \neq n$ **do**
      **if** $slope^+_m \leq slope^-_n$ **then**
        goto step 2
      **end if**
      **if** $\delta e^+_m > \delta e^-_n$ and $\delta t^+_m < \delta t^-_n + slack$ **then**
        change solution of curve $m$ from $i$ to $i + 1$
        change solution of curve $n$ from $j$ to $j - 1$
        update $slack$
        goto step 1
      **end if**
    **end for**
  **end for**
  step 2 :
  sort $slope^+$ descendingly
  **for all** curve $m$ in $slope^+$ **do**
    **if** $\delta t^+_m < slack$ **then**
      change solution of curve $m$ from $i$ to $i + 1$
      update $slack$
      goto step 2
    **end if**
  **end for**
**end procedure**

temps d'exécution courant de sa solution initiale, sera accumulée dans *slack* et ajoutée au temps disponible pour la courbe suivante. Après l'initialisation, les chances d'affiner la solution sont explorées en deux étapes. En *step 1*, il est testé la possibilité de déplacer le point de fonctionnement des courbes deux à deux, à droite pour l'une, à gauche pour l'autre. A la ligne 19, tout les

courbes sont rangées suivant la pente (*slope)* de leur solution courante. $slope^+$, dans le sens descendant et $slope^-$ dans le sens ascendant. Alors, l'algorithme essayera de trouver deux courbes $m$ et $n$, qui satisfassent la contrainte de temps et de consommation d'énergie en mêmes temps, lorsque la solution de $m$ est changée de $i$ à $i+1$ et la solution de $n$ de $j$ à $j-1$ (Figure 6.2). Lorsqu'il n'est plus possible de réaliser ce genre de réglages, l'algorithme passe à l'étape suivante. Le *step 2* effectue le réglage finale en trouvant tout courbe $m$ satisfaisant la contrainte de temps si sa solution courante est déplacée de $i$ à $i+1$. avec $k$ courbes de $l$ points, la complexité de la phase d'initialisation est $O(k.log\ l)$. Pour le *step 1*, chaque itération est au maximum de $O(k^2)$ opérations, alors que le *step 2* en demande $O(k)$.

## 6.4 Conclusion

Dans ce chapitre, nous avons présenté les éléments de mise en oeuvre de l'étape 3. Nous avons,dans un premier temps,formulé le problème, puis présenté et décrit un algorithme de résolution, basé sur l'heuristique de [127]. L'algorithme présenté n'est, malheureusement, pas tout à fait adapté lorsque les groupe de TC n'exploitent par entièrement la plateforme. En effet, pour choisir un point de fonctionnement dans les jeux de solutions des groupes actif, celui-ci ne prend en compte que les aspects consommation et temps d'exécution, en omettant complètement l'aspect spatiale, c'est-à-dire le placement des groupe les un par rapport aux autres. Par exemple, si on considère deux groupes, l'algorithme choisira deux points dont la somme des temps d'exécution est inférieur à l'échéance et consommant le minimum d'énergie. En réalité, si les couples de solutions choisies n'ont aucun cluster en commun, le temps d'exécution n'est plus la somme, mais le plus grand des deux. Il est à noter que cela ne change rien au calcul de l'énergie qui reste toujours égale à la somme des deux énergies. Cependant, lorsque la taille de l'application est du même ordre de grandeur que le nombre de noeud dans le réseau, cet algorithme permet de choisir efficacement les points de fonctionnement des groupes actifs.

Chapitre 7

# Expérimentation

**Sommaire**

## 7.1 Introduction

Nous présentons, dans ce chapitre, les différentes expérimentations que nous avons menées. Les étape 1 et 2 de la méthodologie sont expériementées séparement. Celles-ci sont décrites respectivement dans les sections 7.2 et 7.3, dans lesquels les détails de mise en oeuvres ainsi que les résultas sont présentés.

Mais, avant cela, nous détaillons les différents graphes utilisés, qui sont communs à l'ensemble des expérimentations. Tout d'abord, l'ensemble des graphes ont été créés à l'aide de l'outil TGFF ([37]). Ce dernier génère des graphes aléatoires et permet, aux travers de divers paramètres, d'en ajuster les caractéristiques. Nous avons "sélectionné" différents paramètres afin de générer différents types graphes (c.f. Table 7.1) que nous nommons comme suit : le nom des graphes est représenté par 1, 2 ou 3 lettres identiques parmi {a, b, c}. Trois lettres (***) signifie que le graphe exhibe un fort parallélisme, alors qu'à l'opposé une seule (*) signifie un graphe plutôt séquentie. Les lettres a, b, c informent des relations d'interdépendances entre les tâches, avec respectivement une interdépendance faible, moyen, forte.

Table 7.1 – Les différents type de graphes utilisés.

| Type | Parallélisme | Interdépendance |
|---|---|---|
| a | faible | faible |
| b | faible | moyen |
| c | faible | fort |
| aa | moyen | faible |
| bb | moyen | moyen |
| cc | moyen | fort |
| aaa | fort | faible |
| bbb | fort | moyen |
| ccc | fort | fort |

De plus, différentes valeurs de CCR (Computation to Communication Ratio[1]) typiques de 0.1, 1 et 10 et 100 représentant des cas de communications respectivement élevé, moyen, faible, très faible sont utilisées.

[1] L'inverse, c-à-d.*Communication to Computation Ratio* est aussi utilisé dans la littérature. Attention, cependant, à ne pas confondre.

L'Equation 7.1 décrit le calcul du CCR moyen sur un graphe comportant $V$ tâches et $E$ communications.

$$CCR = \frac{\sum_{v \in V} w(v)}{\sum_{e \in E} c(e)} \tag{7.1}$$

De plus, nous avons généré 3 groupes de graphe : "*small*", "*mid*", et "*big*", chacun composé d'environ 1000 graphes. A l'intérieur d'un groupe, pour un nombre de tâche donnée, plusieurs graphes sont générés avec des poids de tâche $w_v$ et de communication $c_e$ variant indépendamment dans une plage allant de 10 à 80%. Chacun des groupes est composé de graphes de la Table 7.1, dont le nombre de tâche est compris dans une certaine liste, décrite dans la Table 7.2.

Table 7.2 – Les différents groupes de graphes et leurs nombre de tâches.

| Désignation | Nombre de tâche par graphe |
|---|---|
| "*small*" | $5, 10, 20, 30$ |
| "*mid*" | $50, 100, 200, 300$ |
| "*big*" | $500, 1000, 2000, 3000$ |

## 7.2 Etape 1

Comme il a été dit en section 4, nous avons opté pour l'utilisation d'heuristiques de liste afin de réaliser le placement/ordonnancement d'une TC sur un cluster. Dans les sous-sections suivante, nous présentons la méthode d'expérimentation que nous avons suivie puis les résultats d'expérimentation obtenus.

### 7.2.1 Méthode d'expérimentation

Nous avons utilisé, au total, six combinaisons d'heuristiques différentes, 4 pour le placement/ordonnancement simple (c.f. Sec 4) et 2 pour le placement/ordonnancement avec pipeline logiciel. Tout d'abord, pour le placement/ordonnancement simple, nous avons expérimenté le *Static List Scheduling* ainsi que le *Dynamic List Scheduling* que nous avons couplé avec deux autres algorithmes qui sont le *top level* et le *priority level*. Concernant le placement/ordonnancement avec pipeline, nous avons utilisé le même heuristique de liste ([52]), ainsi

que l'algorithme de partitionnement [67], mais en utilisant les schémas "cut" et "volume" comme décrit en section 4. Dans les expérimentations,nous avons utilisé les sigles suivants : "bl+sl", "pl+sl", "bl+dl", "pl+dl", "cut+pipe" et "vol+pipe". La Table 7.3 récapitule ceci. Le nombre de coeur du cluster peut prendre les valeurs 2, 4, 8 et 16.

| Désignation | | *Static List Scheduling* | *Dynamic List Scheduling* | *Pipeline Scheduling* |
|---|---|---|---|---|
| | sigle | sl | dl | pipe |
| *bottom level* | bl | bl+sl | bl+dl | |
| *priority level* | pl | pl+sl | pl+dl | |
| *Edge cut partitionning* | cut | | | cut+pipe |
| *Volume partitionning* | vol | | | vol+pipe |

Table 7.3 – Associations des algorithmes et leurs notations

Deux mesures nous intéresses particulièrement : l'accélération ou *Speed-up* et l'efficacité. Le *Speed-up* est obtenu à partir des mesures du temps séquentiel $t_{seq}$ et du *makespan*, noté $MK$, puis par le ratio de ceux-ci (c.f eq. 8.7). L'efficacité est représentée à l'Equation 7.3, où $ncore$ est le nombre de coeurs.

$$SpeedUp = \frac{t_{seq}}{MK} \tag{7.2}$$

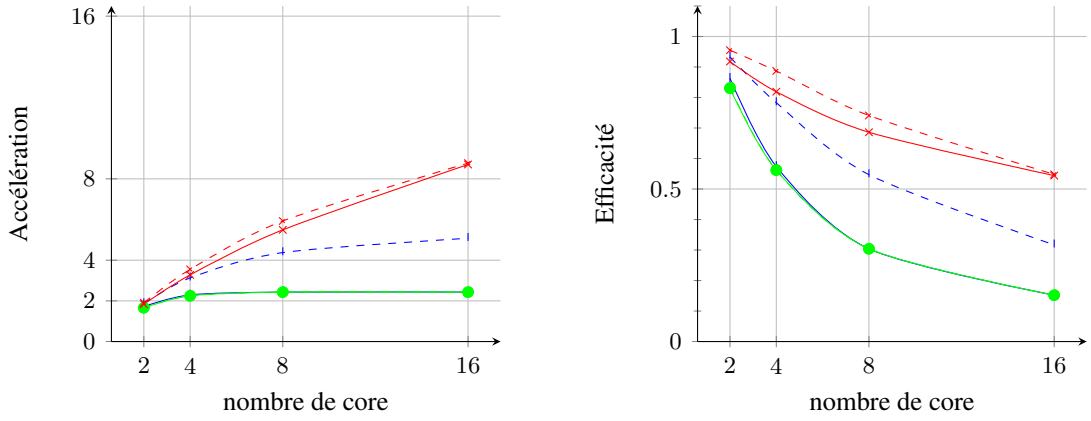
$$Eff = \frac{SpeedUp}{ncore} \tag{7.3}$$

### 7.2.2 Résultats

Les résultats sont organisés suivant les trois tailles de graphe définit dans la Table 7.2 (petit, moyen, grand) puis suivant le CCR utilisé (100, 10, 1 et 0.1).

#### 7.2.2.1 Graphe de petite taille

**7.2.2.1.1 CCR=100** L'accélération (speed-up) des divers algorithmes de *List Scheduling* (bl+sl, bl+dl, pl+dl) est très vite écrêté autour de 2, dès 4 processeurs. Leur efficacité diminue très rapi-



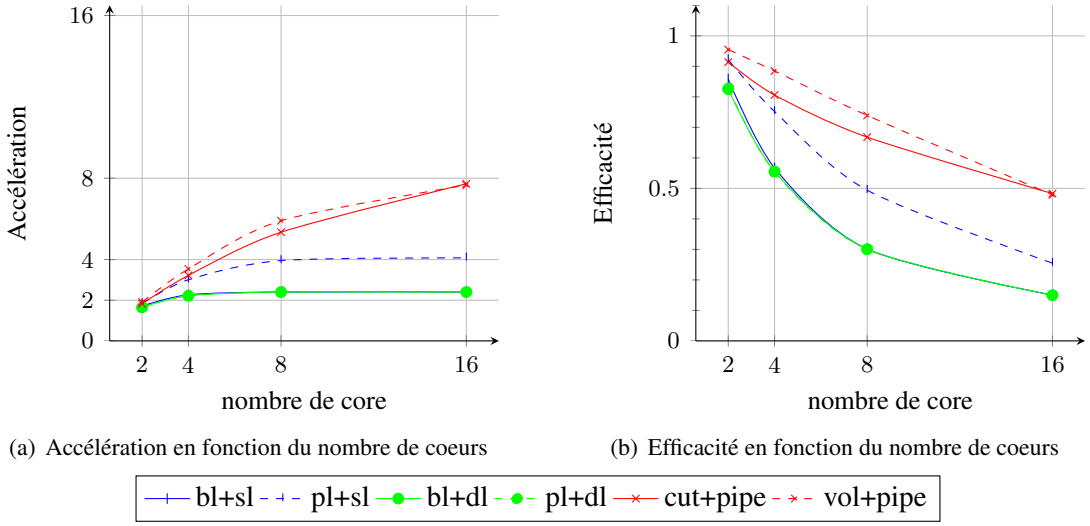

(a) Accélération en fonction du nombre de coeurs

(b) Efficacité en fonction du nombre de coeurs

Figure 7.1 – Graphe *small*, CCR : 100

dement et devient inférieur à 60% au delà de 4 processeurs. Avec un CCR de 100, les quantités de données communiquées sont extrêmement faible. Cependant, les relations de précédences reste présentent et doivent être respectées, ce qui dégrade les algorithmes de listes. Les modes pipelinés, bien que loin du cas idéal, conservent une accélération satisfaisante et présente une efficacité pas trop dégradé jusqu'à 16 processeurs. Ces résultats relativement médiocre sont en grande partie dû au fait que le parallélisme inhérent n'est pas suffisant pour que les divers algorithmes puissent en tirer partie.

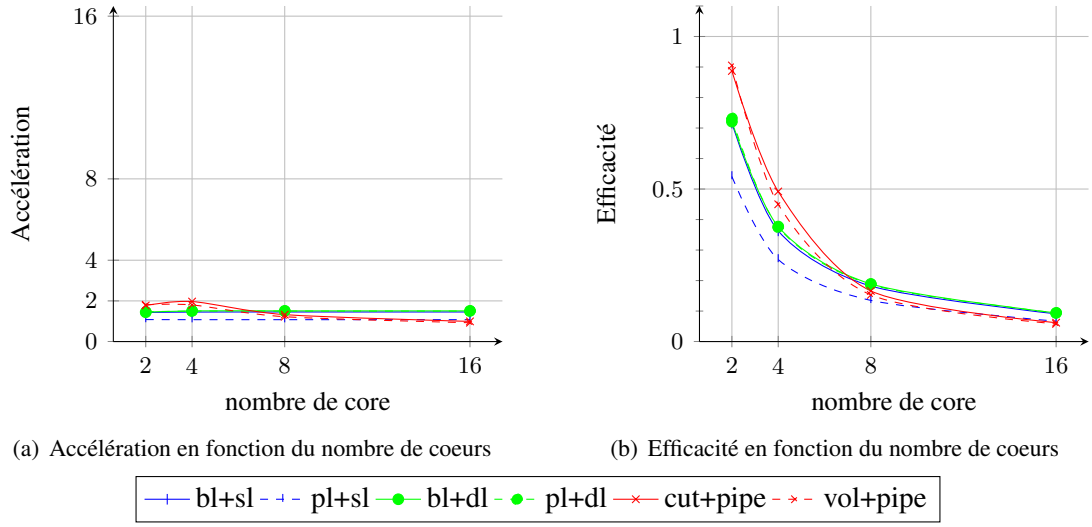
**7.2.2.1.2 CCR=10** Avec un CCR de 10, le même constat qu'avec un CCR de 100 peut être fait. Cependant, on notera une dégradation plus importante des algorithmes de pipelines.




(a) Accélération en fonction du nombre de coeurs (b) Efficacité en fonction du nombre de coeurs

Figure 7.2 – Graphe *small*, CCR : 10

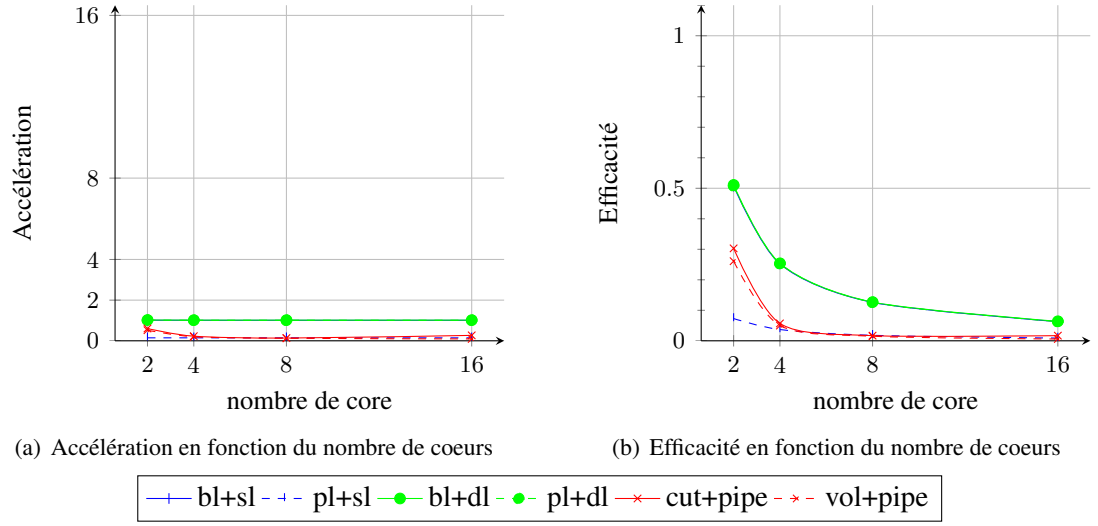
**7.2.2.1.3 CCR=1** Quelque soit l'algorithme utilisé, l'efficacité dégringole et l'accélération est inexistante. Ce résultat s'explique par le faible taux de parallélisme couplé à l'utilisation d'un bus partagé.




(a) Accélération en fonction du nombre de coeurs (b) Efficacité en fonction du nombre de coeurs

Figure 7.3 – Graphe *small*, CCR : 1

**7.2.2.1.4 CCR=0.1** Le constat est le même qu'avec un CCR de 1, si ce n'est que la dégradation est encore plus significative.



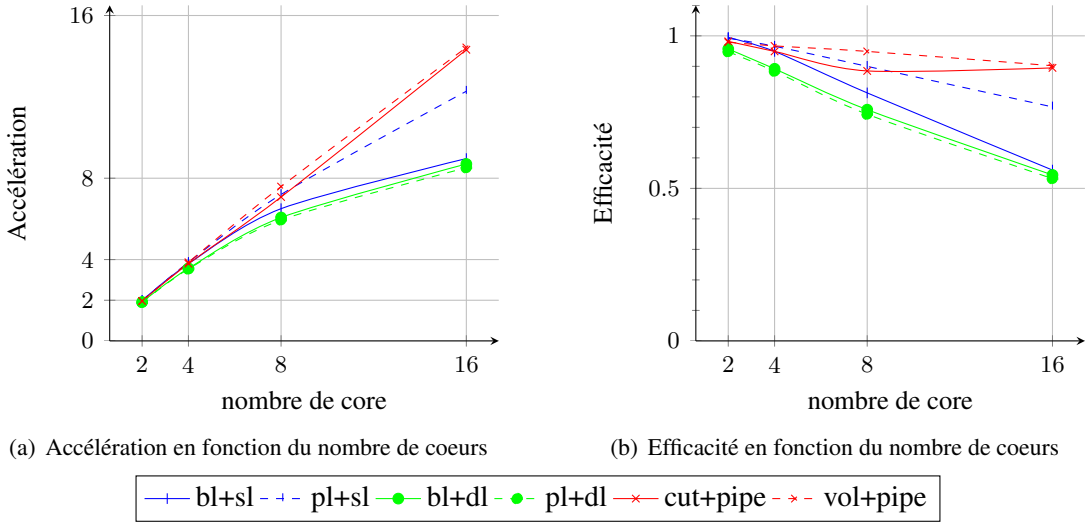

(a) Accélération en fonction du nombre de coeurs

(b) Efficacité en fonction du nombre de coeurs

Figure 7.4 – Graphe *small*, CCR : 0.1

### 7.2.2.2 Graphe de taille moyenne

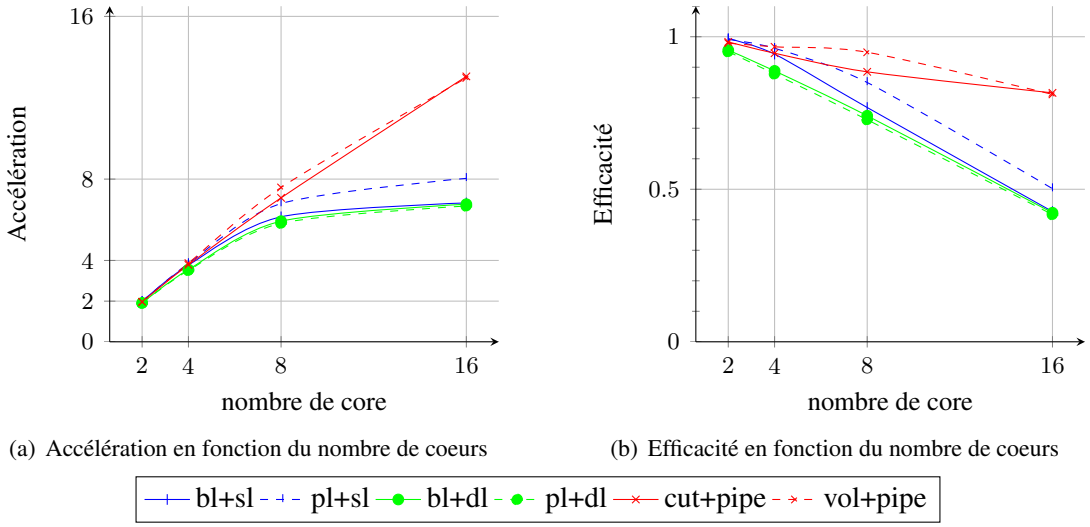
**7.2.2.2.1 CCR=100** Avec un parallélisme potentiel plus important, les divers algorithmes de liste présentent des performances, que ce soit au niveau de l'accélération ou bien de l'efficacité, plus importante. Cependant, mise à part pour pl+sl qui conserve une accélération et une efficacité relativement bonne, les autres algorithmes de liste se dégradent à partir de 8 processeurs. Les algorithmes de pipeline conservent des caractéristiques proche de l'idéal sur toute la plage (de 2 à 16 processeurs).




(a) Accélération en fonction du nombre de coeurs (b) Efficacité en fonction du nombre de coeurs

Figure 7.5 – Graphe *mid*, CCR : 100

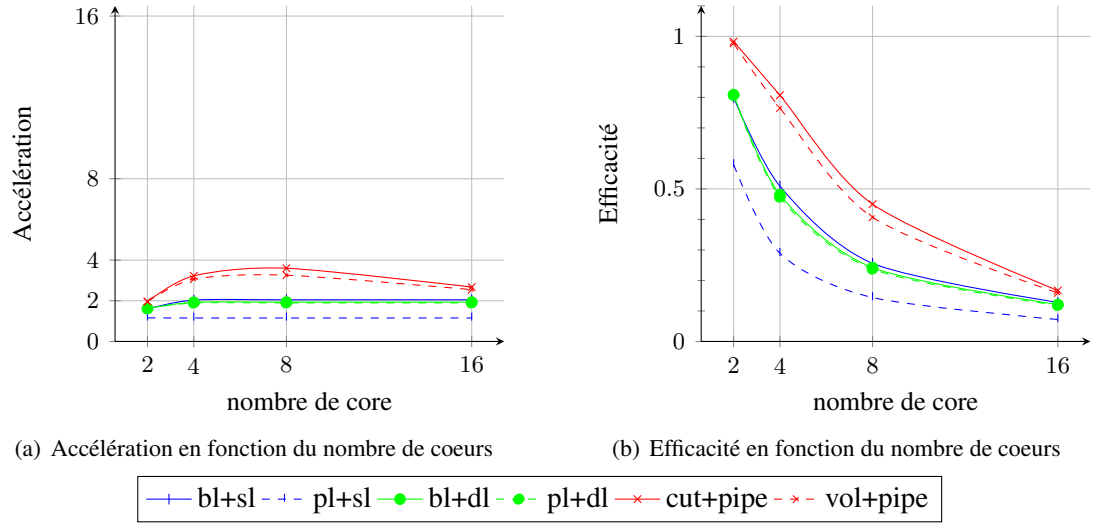
**7.2.2.2.2 CCR=10** Avec l'augmentation du taux de communication, les performance se dégradent. Seul les algorithmes de pipeline conservent de bonnes performances jusqu'à 16 processeurs.




(a) Accélération en fonction du nombre de coeurs (b) Efficacité en fonction du nombre de coeurs

Figure 7.6 – Graphe *mid*, CCR : 10

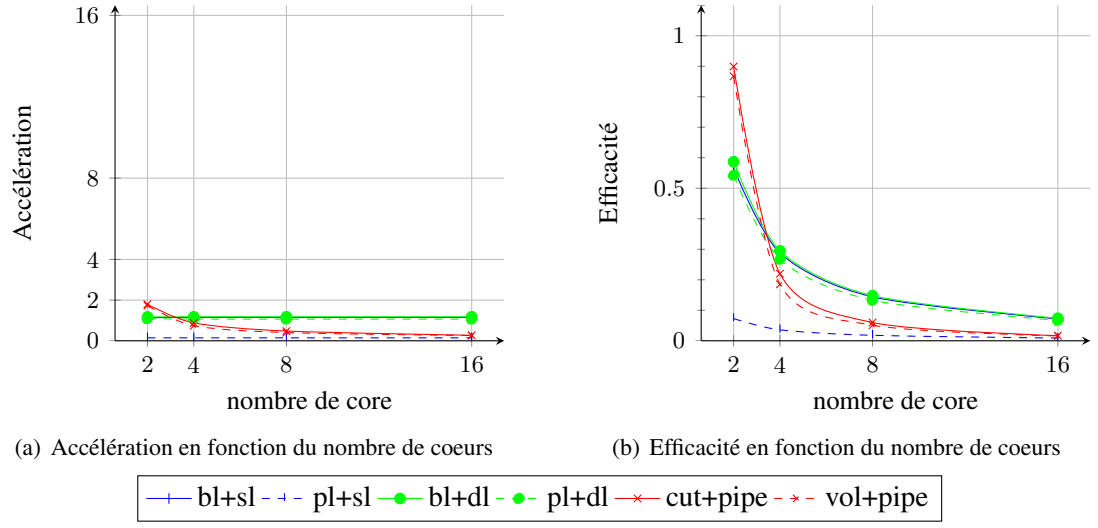
**7.2.2.2.3 CCR=1** Bien que le niveau de parallélisme soit suffisant (c.f. 7.2.2.2.1), le bus partagé limite les performances et ne permet pas une pleine utilisation des processeurs.




(a) Accélération en fonction du nombre de coeurs (b) Efficacité en fonction du nombre de coeurs

Figure 7.7 – Graphe *mid*, CCR : 1

**7.2.2.2.4 CCR=0.1** Même constat que précédemment (CCR=1), si ce n'est que les dégradations sont plus accentuées.

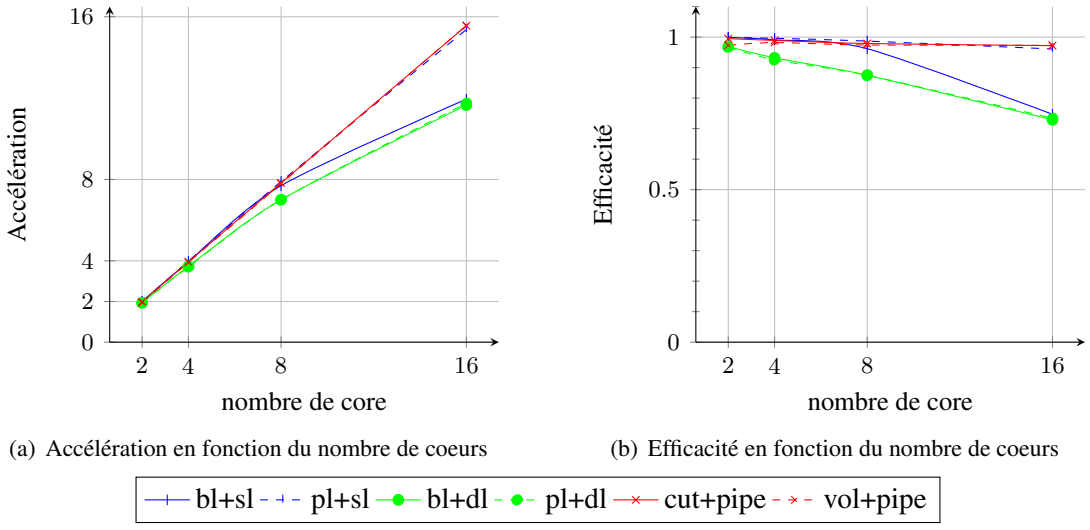



(a) Accélération en fonction du nombre de coeurs (b) Efficacité en fonction du nombre de coeurs

Figure 7.8 – Graphe *mid*, CCR : 0.1

#### 7.2.2.3 Graphe de grande taille

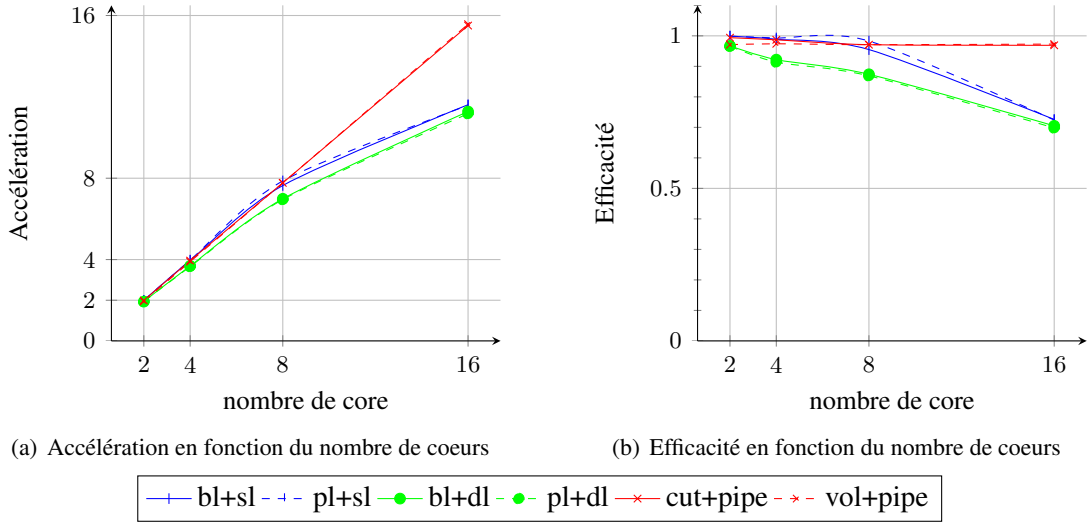
**7.2.2.3.1 CCR=100** Le parallélisme étant très important, les algorithmes de pipeline exhibent des performances quasi idéales sur tout la plage. Les algorithmes de liste présentent d'excellentes caractéristiques, mêmes si ils montrent une dégradation plus importante au delà de 8 processeurs.




(a) Accélération en fonction du nombre de coeurs (b) Efficacité en fonction du nombre de coeurs

Figure 7.9 – Graphe *big*, CCR : 100

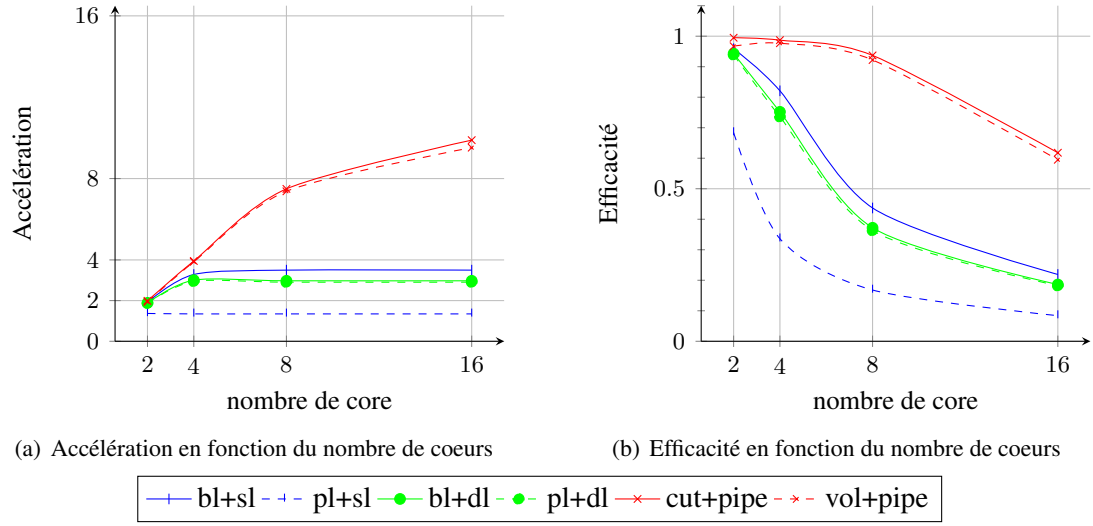
**7.2.2.3.2 CCR=10** Le constat est similaire au précédent, et même si le taux de communication augmente, ce n'est pas suffisant pour fortement dégrader les performances.




(a) Accélération en fonction du nombre de coeurs (b) Efficacité en fonction du nombre de coeurs

Figure 7.10 – Graphe *big*, CCR : 10

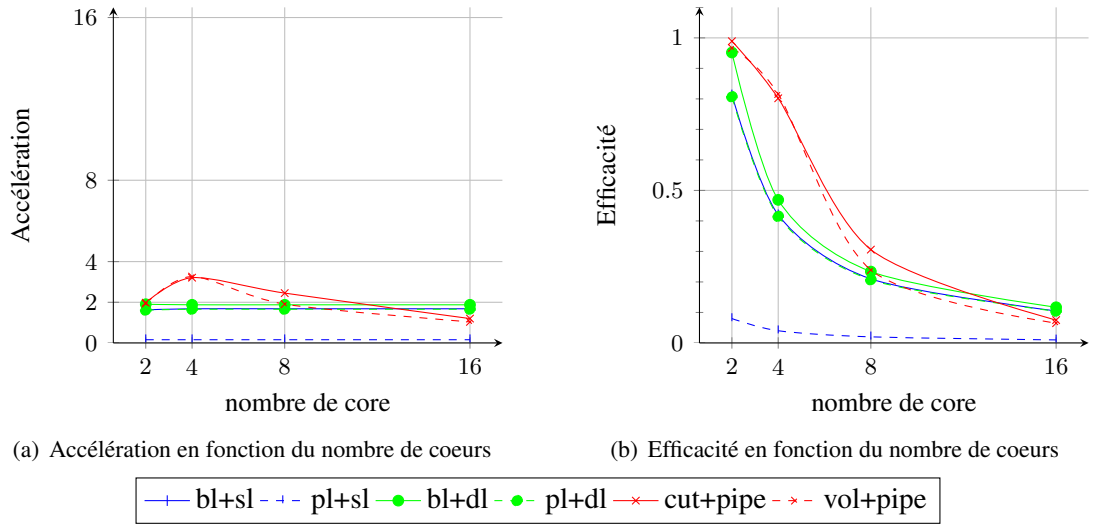
**7.2.2.3.3 CCR=1** Le parallélisme est important, mais le taux de communication limite les performances.




(a) Accélération en fonction du nombre de coeurs (b) Efficacité en fonction du nombre de coeurs

Figure 7.11 – Graphe *big*, CCR : 1

**7.2.2.3.4 CCR=0.1** Le taux de communication est trop important. Le bus partagé est le facteur limitant.




(a) Accélération en fonction du nombre de coeurs

(b) Efficacité en fonction du nombre de coeurs

Figure 7.12 – Graphe *big*, CCR : 0.1

### 7.2.3 Conclusion

Dans cette section, nous avons présenté les expérimentations et résultats menés sur l'étape 1 de notre méthodologie. Les différents algorithmes présentés dans le Chapitre 4 ont été mis en oeuvre et les résultats de mesures d'efficacité ainsi que d'accélération (speed-up) ont été présentés. Quelques soit l'algorithme utilisé, les performances (accélération et efficacité) sont relativement bonnes, à condition que le parallélisme du graphe soit suffisant et que le taux de communication ne soit pas trop important (c.f. paragraphes 7.2.2.3.1, 7.2.2.3.2, 7.2.2.2.1). En effet, l'utilisation d'un bus partagé, limite les performances des divers algorithmes (c.f. 7.2.2.3.3, 7.2.2.3.4, 7.2.2.2.2, 7.2.2.2.3, 7.2.2.2.4, 7.2.2.1.3, 7.2.2.1.4). Les algorithmes de pipelines ont montrés de bonnes performances, mêmes lorsque le parallélisme inhérent est relativement faible (c.f. 7.2.2.1.1, 7.2.2.1.2). Bien que l'utilisation d'un bus partagé limite les performances, les divers algorithmes présentés sont tout à fait envisageable pour une utilisation dans l'étape 1 de la méthodologie. Une adaptation des algorithmes à une interconnexion intra-cluster plus per-

formante (cross-bar, bus hiérarchique) devrait permettre des caractéristiques (accélération, efficacité) proche de l'idéale lorsque le niveau de parallélisme est suffisant. Pour une utilisation dont la cible applicative est périodique, l'utilisation des algorithmes de pipeline couplé à une interconnexion intra-cluster performante permettrait d'atteindre des caractéristiques quasi idéales quelque soit le niveau de parallélisme inhérent à l'application.

## 7.3 Etape 2

Nous présentons ici les expérimentations menées à l'étape 2. Cette étape réalise une exploration des solutions de placement/ordonnancement sur une architecture 2D mesh. Dans cette optique, celle-ci doit prendre en compte la variabilité du processus de fabrication, qui impacte la fréquence de fonctionnement des noeuds (cluster) ainsi que la consommation d'énergie qui en découle. Dans les sous-sections suivantes,nous présentons la méthode d'expérimentation utilisée puis les résultats expérimentaux obtenus.

### 7.3.1 Méthode d'expérimentation

Nous utilisons l'algorithme génétique NSGII auquel nous y avons implémenté nos fonctions d'évaluations (c.f. Chap. 5). L'ensemble des fonctions d'évaluation ainsi que l'algorithme génétique a été implémenté en C. Les paramètres de l'algorithme génétique sont les suivants :

- La taille de la population est de 100 individus
- Le nombre de génération est de 300
- La probabilité de croisement est de 0.9
- La probabilité de mutation est de 0.0125

Dans nos expérimentations, nous faisons varier plusieurs paramètres tel que la taille du réseau $N$, la variabilité (représentée par sont écart-type $\sigma$) et le ratio des puissances. Ce dernier, noté $SD$, est le ratio entre puissance statique et puissance dynamique. Il peut prendre les valeurs 0.001, 0.1, 0.3 et 0.5. Une valeur de 0.001 signifie que la puissance statique est très faible par rapport à la puissance dynamique, alors que par exemple à 0.3, la puissance statique représente 30% de la puissance dynamique. Il a été montré (c.f. Annexe A.2) que la variabilité suit une

loi de probabilité normale $\mathcal{N}(\mu, \sigma)$ de moyenne $\mu$ et d'écart-type $\sigma$. Afin de simuler une "map" de variabilité, nous avons généré des valeurs, suivant une loi normale et assigné aux différents noeud du mesh 2D. Nous avons fixé $\mu = 0$ et $\sigma$ peut prendre les valeurs 0.001, 0.03, 0.06 et 0.12. La taille du réseau 2D mesh est égal à 9 (3x3), 16 (4x4), 64 (8x8), 256 (16x16) ou 1024(32x32) noeuds.

Nous nous intéressons principalement à deux mesures. La première, dénommée "plage de performance" correspond à la plage sur laquelle il est possible de changer de point de fonctionnement afin d'accélérer/améliorer la performance (et donc de réduire le temps d'exécution). Celle-ci est déterminée par l'Equation 7.4, où $MaxTime$ et $MinTime$ sont les valeurs max et min trouvées lors de l'exploration. La seconde mesure nommée "plage d"énergie" est similaire et correspond à la plage sur laquelle il est possible de réduire la consommation d'énergie et est déterminée par l'Equation 7.5. Ces deux mesures permettent de répondre à la question : Par quel facteur est-il possible de réduire, d'une part, le temps d'exécution, et d'autre par, la consommation d'énergie.

$$plage\ de\ performance = \frac{(MaxTime - MinTime)}{MaxTime} \tag{7.4}$$

$$plage\ d'energie = \frac{(MaxEnergy - MinEnergy)}{MaxEnergy} \tag{7.5}$$

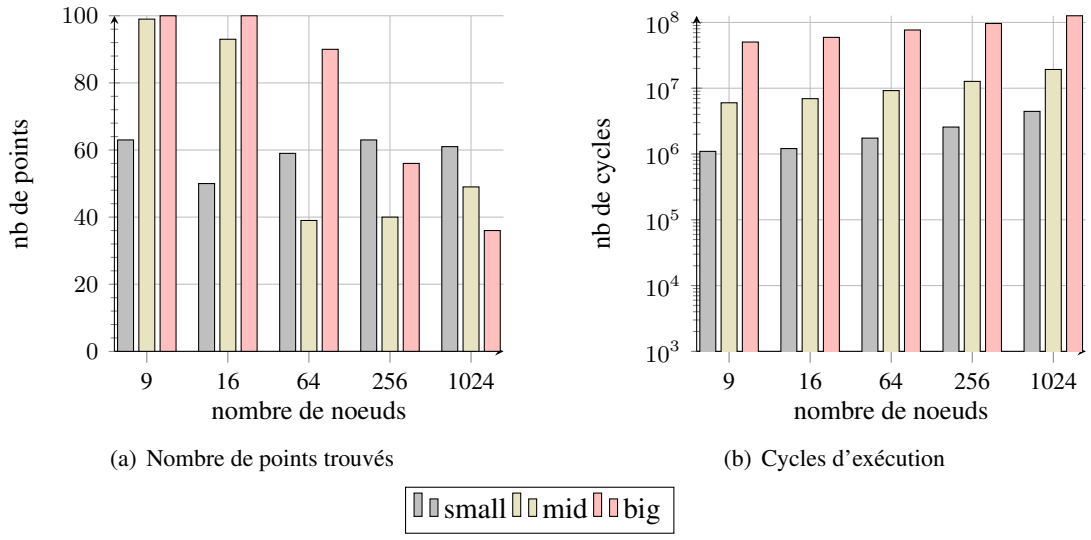
De plus, deux autre mesures ont été prisent en compte. D'une part, le nombre de points de fonctionnement trouvés lors de l'exploration et d'autre part le temps nécessaire à l'exploration.

### 7.3.2 Résultats

Les résultats d'expérimentation sont organisés de la façon suivante : La Section 7.3.2.1 présente une synthèse sur l'ensemble des valeurs de variabilité ($\sigma$) et de ratio de puissance ($SD$). Les Sections 7.3.2.2 et 7.3.2.3 étudient, respectivement, l'impacte de variabilité et du ratio de puissance.

#### 7.3.2.1 Résultats Globaux

On constate, d'après la Figure 7.13(a), que le nombre de point trouvés dépend à la fois de la taille du graphe et de la taille du réseau. En effet, plus le graphe est "grand" et plus le nombre de points est important. De même pour la taille du réseau. Cependant, on peut noter que même dans le pire cas (graphe *big* et 1024 noeuds), le nombre de points est de l'ordre de 35. Ce qui signifie, qu'avec une population de 100 individus (c.f. Sec. 7.3.1) 35% sont des solutions sur le front pareto.

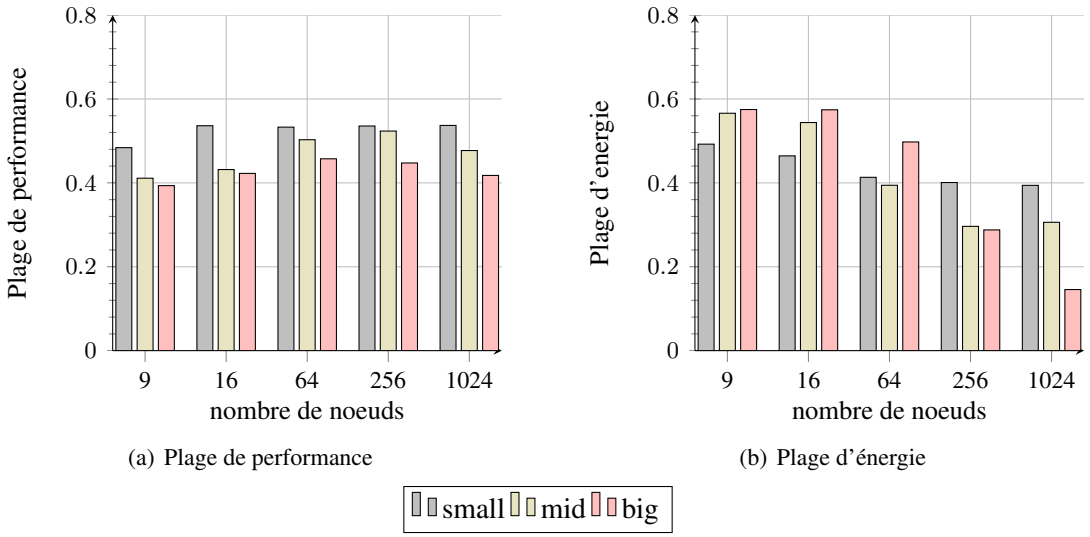



(a) Nombre de points trouvés

(b) Cycles d'exécution

Figure 7.13 – Mesure du nombre de point trouvé et du nombre de cycles nécessaire suivant trois tailles de graphes *small*, *mid*, *big*

La Figure 7.13(b) représente le temps nécessaire à l'exploration en nombre de cycles. Notez que ce dernier est représenter sur une échelle logarithme. Le temps d'exploration est bien entendu proportionnel à la taille du graphe et à la taille du réseau. Dans les Figures 7.14(a) et 7.14(b) représentent respectivement la plage de performance et la plage d'énergie. D'après ces deux figures, on constate d'une part, qu'il est possible de réduire le temps d'exécution de 40 à 55% et d'autre part de réduire l'énergie de 30 à 57% suivant les cas. Seul le cas où le nombre de noeuds est 1024 avec un graphe *big* ne permet de réduire que de 14% l'énergie, ce qui est malgré tout honorable.




(a) Plage de performance (b) Plage d'énergie

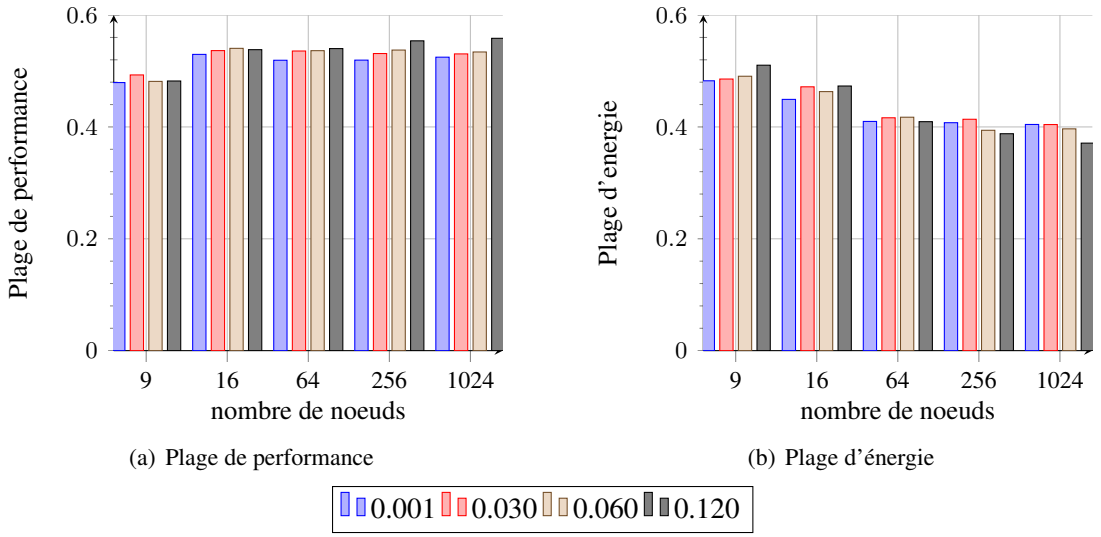
Figure 7.14 – Mesure de plages de performance et d'énergie suivant trois tailles de graphes *small*, *mid*, *big*

#### 7.3.2.2 Impacte de la variabilité

Dans la suite les mesures de plage de performance et de plage d'énergie sont décrites pour les trois tailles des graphes *small*, *mid* et *big*. Là,l'écart type $\sigma$ prend les valeurs 0.001, 0.03, 0.06 et 0.12. Le ratio de puissance étant fixé à 0.001, il ne joue aucun rôle. Que ce soit pour les petits, moyens ou grand graphes, on ne constate que peu d'impact sur la plage de performance, qui reste relativement stable (c.f. Fig.7.15(a), Fig.7.16(a), Fig.7.17(a)). De même, l'impact sur la plage d'énergie reste négligeable (c.f. Fig.7.15(b), Fig.7.16(b), Fig.7.17(b)). En effet, quelque soit la variabilité, l'algorithme génétique, de part l'exploration, va tenter de tirer partie des écarts de fréquences et de consommations (dynamique). Les faibles impacts constatés montrent les capacités d'adaptation de la technique.

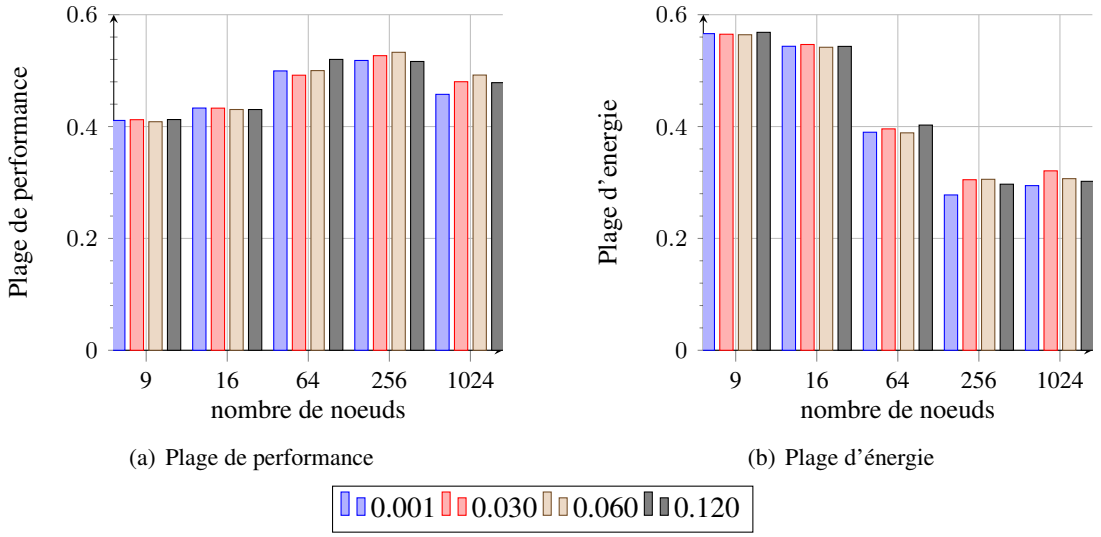
#### 7.3.2.3 Impacte du ratio de puissance

Dans la suite les mesures de plage de performance et de plage d'énergie sont décrites pour les trois tailles des graphes *small*, *mid* et *big*. Le ratio de puissance $SD$ prend les valeurs 0.001,




(a) Plage de performance (b) Plage d'énergie

Figure 7.15 – Mesure de plages de performance et d'énergie suivant quatre variabilités ($\sigma = \{0.001, 0.03, 0.06, 0.12\}$) pour un graphe *small* à $SD = 0.001$

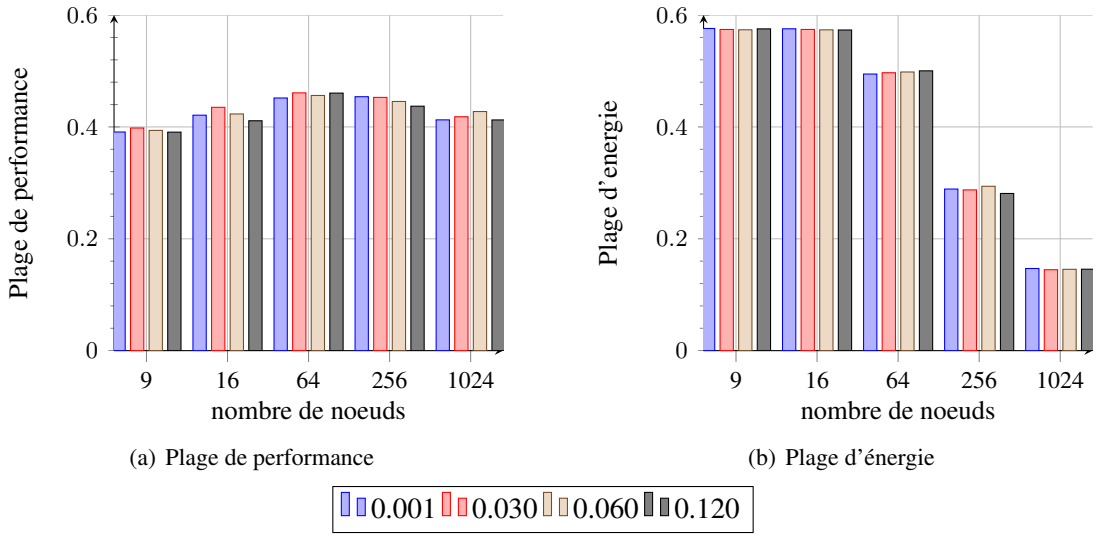



(a) Plage de performance (b) Plage d'énergie

Figure 7.16 – Mesure de plages de performance et d'énergie suivant quatre variabilités ($\sigma = \{0.001, 0.03, 0.06, 0.12\}$) pour un graphe *mid* à $SD = 0.001$

0.1, 0.3 et 0.5. L'écart type $\sigma$ étant fixé à 0.001, il ne joue aucun rôle.

Nous constatons de part les différentes figures ci-après, que quelque soit le graphe employé (*small*, *mid*, *big*), l'impact du ratio de puissance sur la plage de performance est négligeable (c.f.




(a) Plage de performance (b) Plage d'énergie

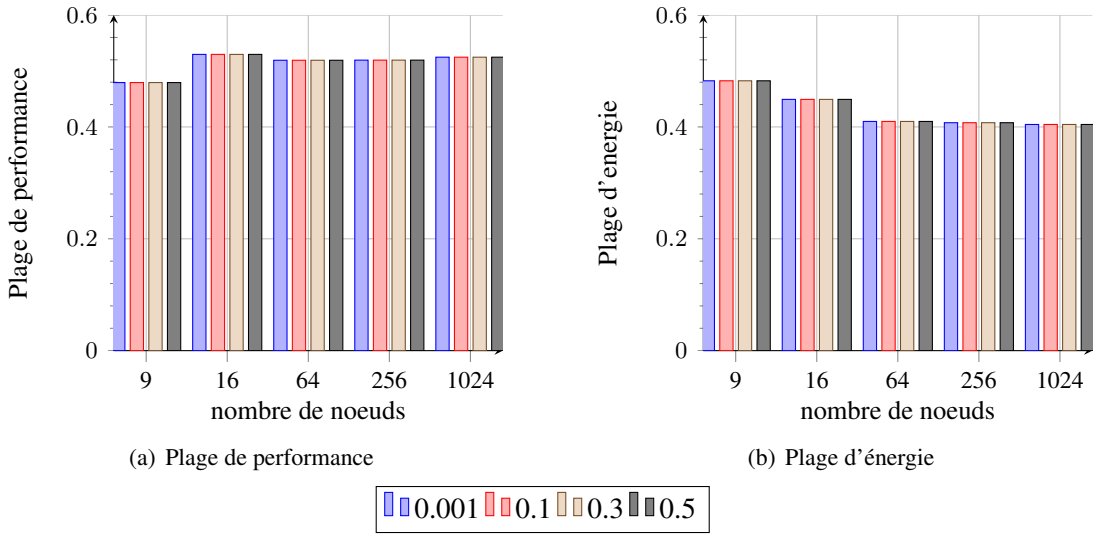
Figure 7.17 – Mesure de plages de performance et d'énergie suivant quatre variabilités ($\sigma = \{0.001, 0.03, 0.06, 0.12\}$) pour un graphe *big* à $SD = 0.001$

Fig.7.18(a), Fig.7.19(a), Fig.7.20(a)). Le même constant peut être fait quant à l'impact sur la plage d'énergie(c.f. Fig.7.18(b), Fig.7.19(b), Fig.7.20(b)).

De façon similaire à la Section 7.3.2.2, quelque soit le ratio de puissance, l'algorithme génétique permet de tirer partie des écarts de consommations (statique) et montre les capacités d'adaptation de la technique.

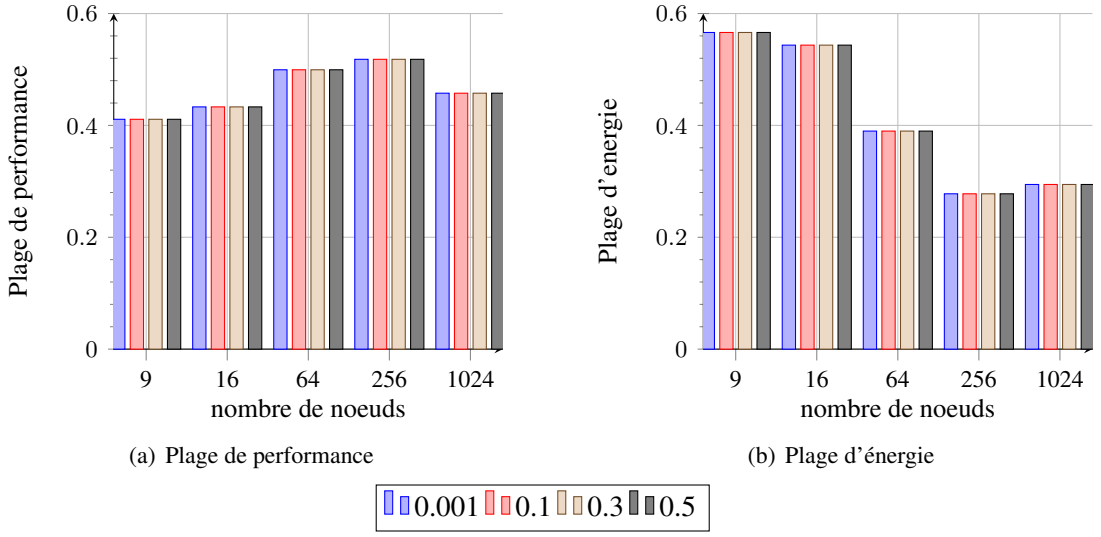
### 7.3.3 Conclusion

Dans cette section, nous avons présenté les différentes expérimentations menées sur l'étape 2. Nous avons, tout d'abord décrit la méthode employée ainsi que les différentes mesures effectuées. Nous avons présenté les résultats obtenus et montré que la technique employée est bien adaptée à notre problème. En effet, les différentes figures en Section 7.3.2.2 et 7.3.2.3 montrent que la technique employée s'adapte très bien aux variabilités dûes au processus de fabrication. De plus, nous avons montré qu'un gain potentiel tant au niveau de la performance (40 à 55%) que de l'énergie (30 à 57%) était possible quelque soit la taille de l'application ou du réseau 2D mesh.



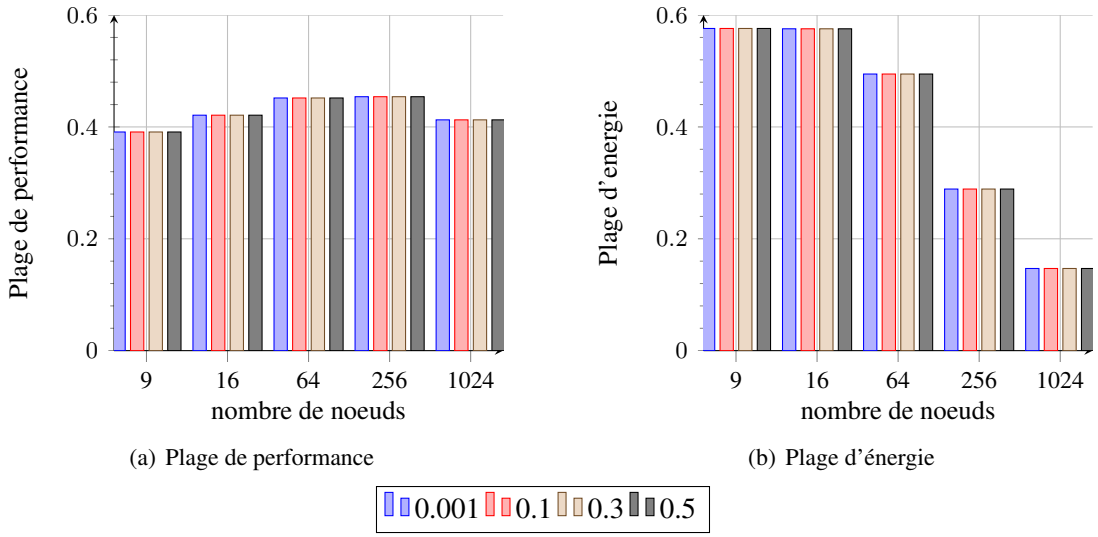

(a) Plage de performance (b) Plage d'énergie

Figure 7.18 – Mesure de plages de performance et d'énergie suivant quatre ratio SD ($SD = \{0.001, 0.1, 0.3, 0.5\}$) pour un graphe *small* avec $\sigma = 0.001$




(a) Plage de performance (b) Plage d'énergie

Figure 7.19 – Mesure de plages de performance et d'énergie suivant quatre ratio SD ($SD = \{0.001, 0.1, 0.3, 0.5\}$) pour un graphe *mid* avec $\sigma = 0.001$




(a) Plage de performance (b) Plage d'énergie

Figure 7.20 – Mesure de plages de performance et d'énergie suivant quatre ratio SD ($SD = \{0.001, 0.1, 0.3, 0.5\}$) pour un graphe *big* avec $\sigma = 0.001$

## 7.4 Conclusion

Dans ce chapitre, nous avons présenté les différentes expérimentations que nous avons menées. Dans un premier temps, les différents algorithmes présentés dans le Chapitre 4 ont été mise en oeuvre et les résultats de mesures d'efficacité ainsi que d'accélération (speed-up) ont été présentés. Nous avons montré que l'ensemble des algorithmes utilisés, donnaient des performances (accélération et efficacité) tout à fait acceptable lorsque l'application présente un parallélisme suffisant. D'autre part, les algorithmes de pipelines mises en oeuvres ont montrés de bonnes performances, mêmes lorsque le parallélisme inhérent était relativement faible. Bien que l'utilisation d'un bus partagé limite les performances, les divers algorithmes présentés sont tout à fait envisageable pour une utilisation dans l'étape 1 de la méthodologie. Lorsque l'application est périodique, l'utilisation des algorithmes de pipeline couplé à une interconnexion intra-cluster performante permettrait d'atteindre des caractéristiques quasi idéale quelque soit le niveau de parallélisme inhérent à l'application. Dans un deuxième temps, nous avons présenté les différentes expérimentations menées sur l'étape 2. Nous avons, tout d'abord décrit la méthode employé ainsi que les différents résultats obtenus. Nous avons montré que la technique

employée s'adapte très bien aux variabilités dûes au processus de fabrication et permet un gain potentiel de 40 à 55% en terme de performance et de 30 à 57% au niveau de l'énergie. Et ce quelque soit la taille de l'application ou du réseau 2D mesh.

# Troisième partie

## Placement d'application auto-adaptatif tolérant aux défaillances

**Sommaire**

Chapitre 8

# Technique Auto-adaptative aux défaillances

**Sommaire**

## 8.1 Introduction

### 8.1.1 Modèles : Système et Application

Le système ciblé est basé sur une réseau d'interconnexion ayant une topologie en maille 2D avec un grand nombre, fixé, de noeuds. Chaque noeud est une unité de calcul connectée à son router local par l'intermédiaire de sont interface réseau. Chaque noeud est potentiellement exposé à des défaillances. L'algorithme de routage assure, adaptativement, une service "Best Effort" en présence de défaillances de routers et de liens.

Les applications de *Streaming* peuvent êtres modélisées par un Graphe Dirigé Acyclique ou DAG (Directed Acyclic Graph) [82].

Nous considérons, ici, des applications de *Streaming* respectant une structure fork-join ([135, 81]). Le DAG Fork-join est une forme restreinte de DAG, où à chaque opération *fork* correspond une opération *join*.

De plus, la possibilité d'une structure imbriquée à l'intérieur du DAG est admise, tel qu'il est représenté en Figure 8.1, dans laquelle le flot d'exécution n'est pas connue à priori. La Figure 8.1 décrit un exemple simple d'application, où $S_{ijk}$ représente la procédure principale (fonction *main*). $S_{ijk}$ créé deux tâches indépendantes $S_{ij}$ et $S_k$, puis attend leur terminaison. Après quoi, $S_{ij}$ démarre deux autre tâches $S_i$ et $S_j$, qui créent, respectivement $\{T_{i1}, T_{i2}\}$ et $\{T_{j1}$ à $T_{j3}\}$. $S_k$ créé quatre tâches parallèles $T_{k1}$ à $T_{k4}$ qui ont le même code mais travaillent sur des données différentes, et de façon similaire les taches $T_i$s et $T_j$s. Il est à noter que chaque *fork* (création) doit être suivit par un *join* (waiting).

### 8.1.2 Formulation du Problème

Afin d'être efficace, une stratégie tolérante aux fautes au niveau système, nécessite des techniques de bas niveaux permettant de saisir le comportement en détails.

Au sein de l'architecture ciblée, le sous-système de routage doit aussi être tolérant aux fautes. Ainsi, la stratégie proposée se repose sur un algorithme adaptatif tolérant aux fautes, capable d'adapter le chemin de routage à la topologie dynamique du réseau, dû aux défaillances de

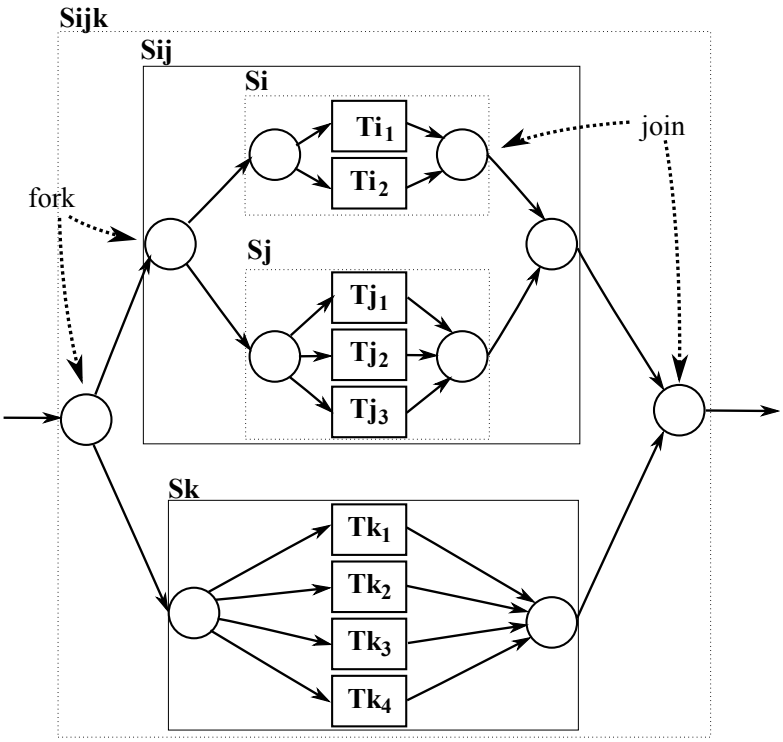



Figure 8.1 – Exemple illustratif d'application

noeuds, de routeurs et de liens. En conséquence, la politique de gestion des défaillances (*Failure Management Policy*) doit garantir la terminaison de l'application. Dans la suite, les noeuds peuvent se trouver dans trois états différents : (i)*Idle* : le noeud est fonctionnel et n'a pas de travail alloué. (ii) *Busy* : le noeud est fonctionnel, mais un travail lui est déjà alloué. (iii)*Fail* : le noeud n'est pas fonctionnel.

Se reportant à l'algorithme de routage adaptatif précédemment cité, et considérant des applications pouvant créer dynamiquement des tâches , notre technique est capable, durant l'exécution, de garantir sans "coupure", la terminaison de l'application et la délivrance du résultat attendu, en dépit de multiples défaillances de noeuds et de liens dans un réseau sur puce de topologie en maille 2D.

## 8.2 Placement et Rétablissement autonome de tâches applicatives

### 8.2.1 Organisation Hiérarchique et obligations

La structure imbriquée de l'application ainsi que son flot d'exécution inconnu, conduit clairement à une organisation hiérarchique directe, dans laquelle chaque niveau a ses propres responsabilités. Les relations entre tâches parents et tâches enfants, ainsi que leurs obligations respectives, sont représentées par une organisation hiérarchique. Dans cette hiérarchie, un *Stream Leader* est une tâche qui crée d'autres tâches, des tâches enfants. La tâche n'ayant pas de parents est appelé "racine" ou *root*. Chaque *Stream Leader* est responsable des ses enfants, de leur placement sur les noeuds à leur achèvement. Les enfants doivent, par conséquent, rendre compte à leur *Stream Leader*. Par exemple, en Figure 8.1, le *Stream Leader Sj* doit placer ses tâches enfants *Tj1* à *Tj3* et s'assurer qu'elles s'exécuteront jusqu'au bout, mêmes en cas de défaillance. Lorsqu'un enfant doit créer une nouvelle tâche, devenant ainsi un *Stream Leader*, la procédure à suivre consiste en deux phases. Premièrement, parmi toutes les ressources disponibles, les ressources requises doivent être trouvées et réservées. Deuxièmement, le code (programme) de la tâche crée est transféré vers la destination allouée. Après ces deux étapes, la tâche nouvellement crée est prête et le notifie à son *Stream Leader*. Alors, ce dernier peut envoyer les données et l'exécution de la tâche enfant commence. Une fois terminée, chaque tâche enfant envoie son résultat à son *Stream Leader*

### 8.2.2 Détection des pannes durant l'exécution

Durant l'exécution, des défaillances peuvent apparaître et rendre certains noeuds, accueillant une tâche enfant, inatteignables. Ainsi, le *Stream Leader* ne pourra obtenir le résultat escompté et dû fait de la non-terminaison de cette tâche, l'application tout entière pourrait "planter". Se basant sur l'hypothèse de *fail-silent*, chaque tâche enfant doit envoyer périodiquement, un message "*I am alive*" ("Je suis en vie"), afin de notifier son *Stream Leader* qu'elle est toujours en cours d'exécution. De l'autre côté, le *Stream Leader*, ayant la responsabilité de ses enfants, vérifie qu'ils sont toujours "en vie" en contrôlant la réception des messages *"I am alive"*. Si l'un

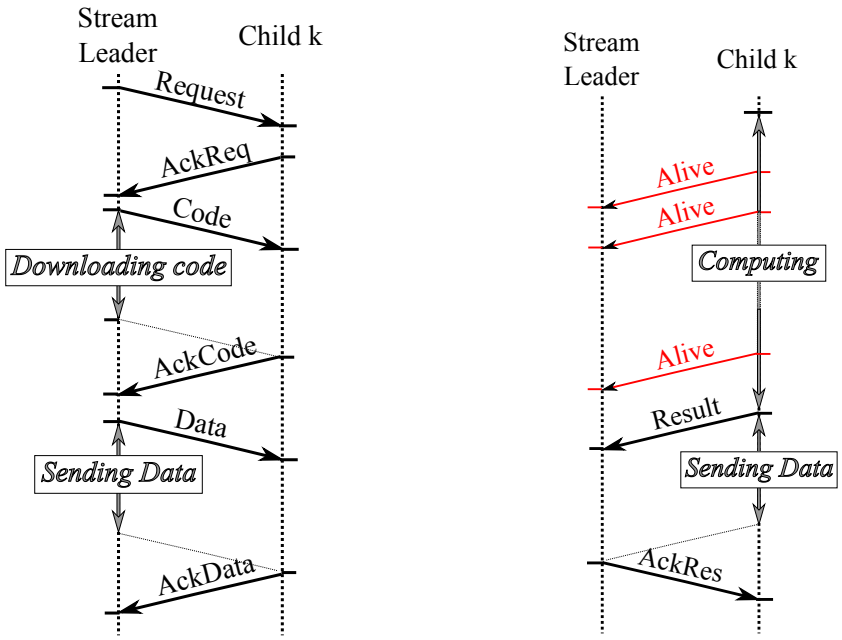
de ces messages n'a pas été reçu, le *Stream Leader* redémarre la tâche non-terminée sur un autre noeud. Lorsque une défaillance intervient sur un *Stream Leader*, tous ses enfants directs sont perdus ainsi que tous les sous arbres en résultants. Afin de maintenir la cohérence au sein de l'application, tous les sous arbres perdus doivent êtres réassigné sur d'autres noeuds non - défaillants. Ainsi, lorsqu'un enfant envoie un message *"I am alive"* à son *Stream Leader* défaillant, il recevra un acquittement négatif (*Nack*), car son *Stream Leader* n'est pas joignable. Il s'en suit, que l'enfant se détruit lui-même et le noeud retourne dans un état non-actif (*Idle*). La fréquence avec laquelle un enfant envoi les messages *"I am alive"* dépend de l'application (durée des tâches, taille des messages échangées), mais intervient au moins une fois durant l'exécution de la tâche. En fait, la durée séparant deux messages *"I am alive"* consécutifs est la latence minimale entre l'apparition de la défaillance et sa détection.

### 8.2.3 Stratégie de recherche Tolérante aux défaillances

Lorsqu'un *Stream Leader* créé de nouvelles tâches, il doit tout d'abord trouver des noeuds non-défaillants et inexploités afin de les allouer. Etant donné l'absence de mécanisme de centralisation de l'état du système, le *Stream Leader* doit parcourir séquentiellement les noeuds existants afin de déterminer leur disponibilités. Ainsi, suivant une stratégie de "recherche à proximité" (*Nearest Neighbor search*), un *Stream Leader* teste chacun de ses voisins du plus proche au plus éloigné, jusqu'à ce qu'un noeud disponible soit trouvé.

**Définition 3** (Stratégie de recherche à proximité)**.** *Soit un ensemble de noeuds $\mathcal{S}$ dans un espace à métrique Manhattan $\mathcal{M}$. Soit un noeud demandeur $r \in \mathcal{S}$. Trouver un noeud non-exploité $q \in \mathcal{S}$ aussi proche (relativement à $\mathcal{M}$) que possible de $r$, en présence de défaillances.*

Le fonctionnement général est le suivant : Le *Stream Leader* envoie un message de requête à un des ses voisins les plus proches, commençant par les noeuds à un saut. Chaque noeud libre recevant cette requête, répondra par un message d'acquittement *AckReq* (Figure 8.2(a)) et sera automatiquement réservé. Chaque noeud déjà occupé retournera un acquittement négatif (*NackReq* or *Nack*). De même, si le noeud destination est défaillant, les éléments de routage répondront par un message *Nack*, signifiant que la destination ne peut être atteinte. Du point




(a) Initialisation du noeud k.

(b) Exécution de l'enfant k.

Figure 8.2 – Messages échangés entre le *Stream Leader* et un enfant k

de vue du *Stream Leader*, la réception d'un message *AckReq* lui assure la réservation du noeud destination. Chaque noeud non-défaillant acquittera les requêtes dans l'ordre *FCFS*(*First Come, First Served*), c'est-à-dire premier arrivé, premier servi. Dû fait d'un trafic potentiellement dense ou de défaillance du réseau de communication, une latence importante peut exister avant que le *Stream Leader* ne reçoive une réponse. En conséquence, après un certains temps (time-out), le *Stream Leader* considérera que le noeud destination est temporairement injoignable. En fait, lorsqu'un message *NackReq* est reçu ou bien que le time-out est expiré, le *Stream Leader* passe au noeud suivant et réitère la procédure.

Une fois un noeud libre et non-défaillant réservé, le code programme et les données de la nouvelle tâche sont chargés, tel que représenté en Figure 8.2(a). De plus, si le time-out se produit avant que le noeud réservé ne reçoit le code et les données, celui-ci retourne dans l'état inactif, ce qui permet d'éviter d'attendre indéfiniment en cas de défaillance du *Stream Leader*. Suivant le système, la taille de l'application et le nombre de noeuds défaillants, il se peut qu'il n'y ai plus de noeuds disponible. Dans ce cas, le *Stream Leader* arrête la recherche et exécute la nouvelle

tâche lui-même. Une fois le code et les données transférés, le processus enfant peut démarrer. Lorsque la tâche enfant est terminée, le résultat est envoyé à son *Stream Leader*. Si, durant l'exécution, un noeud enfant est défaillant, alors son *Stream Leader* en sera informé du fait de l'absence du message *"I am alive"*. Alors, suivant la *Nearby Search Strategy*, le *Stream Leader* va "replacer" la tâche défaillante sur un autre noeud (si il existe). Il est à noter que les données à traiter, doivent être sauvegardées jusqu'à la terminaison de la tâche enfant.

**Définition 4** (Re-mapping)**.** *Lorsqu'un Stream Leader est défaillant, tous ses enfants sont perdus. Ce fait est récursif, ce qui implique que chaque sous-branche du Stream Leader défaillant sont aussi perdues et doivent, par conséquent, êtres replacées.*

Partant de l'utilisation d'un DAG ayant une structure fork-join hiérarchique (tel que représenté en Figure 8.1), l'hypothèse est faite que chaque tâche fork et chaque tâche dual join sont toutes deux placées sur le même noeud (c-à-d, le *Stream Leader*), dû fait de la caractéristique de barrière du join. Ainsi, après qu'une tâche fork ait terminée son exécution, une phase de "search & map" se déroule itérativement (un pas pour chaque enfant) pour tous les enfants. Alors, les données à traiter sont transmises. Après quoi, la tâche join démarre et attend les données entrantes (résultats) issues des enfants. Durant chaque pas de "search & map", un *Stream Leader* pourrait être amené à essayer de nombreux noeuds défaillant ou occupé, avant d'en trouver un disponible (idle).

**Définition 5** (Search Step)**.** *Un "step" est la procédure permettant de trouver un noeud disponible. Potentiellement, de nombreux essais infructueux ($unsucc\_tries$) sur des noeuds défaillants ou occupés peuvent être nécessaire avant qu'un noeud libre soit trouvé.*

Le nombre d'essais pour atteindre $n$ enfants est donné par :

$$\sum tries \; = \; \sum unsucc\_tries + n \tag{8.1}$$

En Section 8.2.4, l'Algorithme 8.1 permettant le *Nearby Search&Map* est proposé, suivit par un exemple en Fig. 8.3. Les Figure 8.3(a) et Figure 8.3(b) décrit le placement, respectivement avant et après qu'une défaillance se produise, alors que le nombre d'essais pour la recherche

et le replacement sont représentés en Figure 8.3(c) et Figure 8.3(d). Une des conséquences des défaillances est que les distances entre des enfants et leurs *Stream Leader* peuvent augmenter et ainsi dégrader les performances du système. Cependant, dans le cas d'une maille 2D, le nombre de noeuds disponible augmente rapidement avec le nombre de sauts (hops) tel que décrit en Définition 6, ce qui limite la dégradation.

**Définition 6.** *Soit un domaine en maille 2D non-borné, c-à-d avec un très grand nombre de noeuds. Considérant une recherche dans les quatre directions à partir du noeud $n_{(i,j)}$ et à une distance de (hop) hops autour de celui-ci, alors, le noeud atteignable est donné par l'équation 8.2 et le nombre de hop moyen pour atteindre ceux-ci est donnés en équation 8.3.*

$$n\_node \;=\; \sum_{i=1}^{hop} (4 \cdot i) \;=\; 2 \cdot hop \cdot (hop + 1) \tag{8.2}$$

$$av\_hop \;=\; \frac{\sum_{i=1}^{hop} (4 \cdot i \cdot i)}{\sum_{i=1}^{hop} (4 \cdot i)} \;=\; \frac{2 \cdot hop + 1}{3} \tag{8.3}$$

### 8.2.4 Placement de DAG en présence de défaillances

Afin d'implémenter la stratégie tolérante au défaillance, précédemment décrite, nous proposons l'Algorithme 8.1. Celui-ci prend en paramètre d'entrée le nombre de noeuds ($n\_node$) à trouver, la liste des voisins à tester ($neighborsList$) et fournit en sortie la liste des noeuds trouvés ($nodeList$). Chaque entré de liste peut être marquée comme étant BUSY et/ou CHILD. Démarrant avec la première entrée de la liste, si l'adresse destination n'est pas marquée BUSY (ligne 5), une requête est envoyée (ligne 6). Si l'état retourné est *Idle* (ligne 7), l'adresse du noeud destination est ajoutée à la liste $nodeList$ (ligne 10) puis marquée BUSY et CHILD dans $neighborsList$ (ligne 8-9). Si l'état retourné est BUSY (ligne 12-13), l'entrée correspondante de la liste $neighborsList$ est marquée BUSY. Sinon, l'entrée de liste est retirée de $neighborsList$ (ligne 15), signifiant que le noeud est défaillant ou bien injoignable. Ce processus est réitéré avec la prochaine entrée de $neighborsList$ jusqu'à ce que tous les noeuds aient étés visités ou que

**Algorithme 8.1** Nearby Search Strategy

```
1:  procedure SEARCH(n_node, neighborsList )
2:      nodeList ← {∅}
3:      destAddress ← FIRSTENTRY(neighborsList)
4:      while n_node > 0 and destAddress ∈ neighborsList do
5:          if isBusy(destAddress) = False then
6:              status ← SEND(REQ, destAddress)
7:              if status = AckREQ then /*destination is Idle*/
8:                  SET(BUSY, destAddress)
9:                  SET(CHILD, destAddress)
10:                 PUSH(childrenList, destAddress)
11:                 n_node ← n_node − 1
12:             else if status = NAckREQ then/*destination is Busy*/
13:                 SET(BUSY, destAddress)
14:             else/*destination is unreachable*/
15:                 REMOVE(neighborsList, destAddress)
16:             end if
17:         end if
18:         destAddress ←NEXTENTRY(neighborsList)
19:     end while
20:     return nodeList
21: end procedure
```

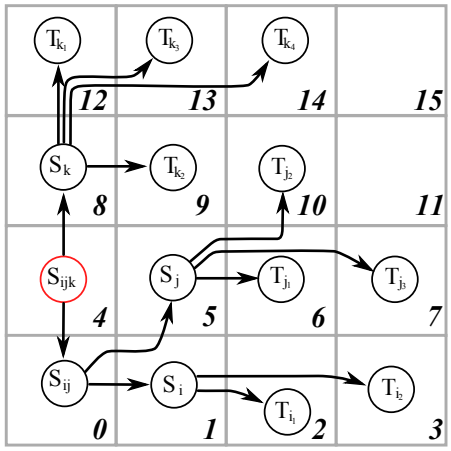
$n\_node$ noeuds aient étés trouvés. Alors, la liste des noeuds libres trouvés $nodeList$ (ligne 20) est retournée. La liste $neighborsList$ est construite de la façon suivante : les voisins sont ajouté un à un du plus proche au plus éloigné, ce qui est déterminé par le nombre maximale de hops sans défaillance de liens.

En guise d'exemple, la Figure 8.3 représentant l'application de la Figure 8.1 placée sur une maille 2D mesh de taille 4 par 4. La Figure 8.3(a) montre un possible placement résultant de *Nearby Strategy*, alors que la Figure 8.3(b) décrit le résultat d'un replacement suite à la défaillance du noeud 5. En Figure 8.3(a) et Figure 8.3(b), les flèches représentent des communications bidirectionels entre les *Stream Leader*s et leurs enfants, suivant le modèles fork-join. La direction des flèches donne la direction des dépendances des *Stream Leader*s vers leurs enfants. Par exemple, le *Stream Leader* $S_j$ communique les données traitées à son propre *Stream Leader*, $S_{ij}$, duquel sont issue les données à traitées. Similairement, $S_j$ envoi des données à ses trois enfants $T_{j_1}, T_{j_2}$ et $T_{j_3}$, qui renvoi les données traitées. Imaginons que le noeud 5 soit un défaillant. Les Figure 8.3(c) et Figure 8.3(d) montrent respectivement le nombre d'essais pour replacer $S_j$, et $T_{j1}$ à $T_{j3}$ après la défaillance du noeud 5. Le *Stream Leader* $S_{ij}$ détecte que son enfant $S_j$ (node 5) est défaillant de part l'absence du message *"I am alive"*. Dans ce cas, il

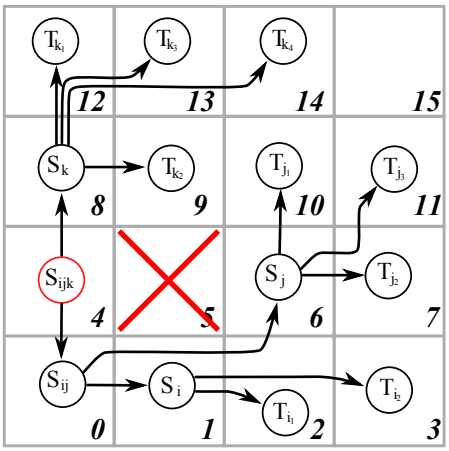
démarre la recherche de noeuds disponible (non-défaillant et libre) en utilisant *Nearby Search Strategy*. Entre temps, les noeuds 6, 7 and 10, de part leur dernier message *"I am alive"* acquitté négativement (*Nack*), se détruisent eux-mêmes et deviennent alors disponibles (*Idle*). Dans la Figure 8.3(c), $S_{ij}$ (noeud 0) connaît les noeuds 4 et 1, puisqu'ils sont respectivement son *Stream Leader* et un de ses enfants. Ainsi, il commence la recherche sequentiellement par les noeuds les plus proches 8, 2, 12, 9, qui sont tous occupés. Finalement, le noeud libre 6 est trouvé et $S_j$ est placé sur celui-ci (le code et les données stocké par $S_{ij}$ sont envoyés), comme en Figure 8.3(b). Après quoi, $S_j$ démarre sa recherche suivant la même voie (Fig. 8.3(d)) et re-map les enfants $T_{j_1}, T_{j_2}$ et $T_{j_3}$. Parce que chaque *Stream Leader* est responsable de ses enfant et donc de tous leurs descendants, lorsqu'une défaillance apparaît, seul la partie de l'application relative à la défaillance est stoppée. Le reste de l'application peut continuer à s'exécuter. Cependant, il doit être noté que les *Stream Leader*s hiérarchiquement proche du noeud root sont plus sensibles aux défaillances, car plus d'enfants en dépendent. Néanmoins, notre technique peut garantir que l'application continuera à s'exécuter sans intervention externe et ce même en cas de défaillance.

### 8.2.5 Analyse du placement en présence de défaillances multiples

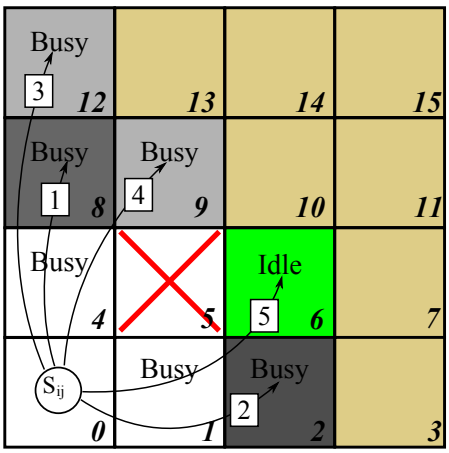
Afin d'analyser la complexité des messages d'un *Stream Leader*, la défaillance de noeuds déjà mappé durant la phase de search & map n'est pas considérée. Une autre hypothèse inhérente est que le *Stream Leader* considéré ainsi que tous ses ancêtres soient exempte de défauts. En effet, si un de ces noeuds est défaillant, c'est son propre *Stream Leader* qui devra procéder au re-mapping en accord avec l'Algorithme 8.1. Chaque fois qu'un enfant est défaillant, son *Stream Leader* doit chercher un autre noeud disponible. Afin de mapper avec succès, $n$ enfants, $n + k$ essais seront nécessaire, $k$ étant le nombre d'essais infructueux. Il est supposé que seul le *Stream Leader* est mappé. Nous nous intéressons au nombre d'essais supplémentaire $k$, nécessaire tel que $n$ essais soient un succès. Soit $X$ la variable aléatoire donnant le nombre de noeuds défaillant $k$ visité durant une recherche pour $n$ noeuds et $P_s$ la fiabilité d'un noeud. En théorie des probabilités, la loi binomiale négative est une distribution discrète du nombre de succès avant qu'un nombre donné d'échec n'apparaissent dans une séquence d'expérience de Bernoulli.



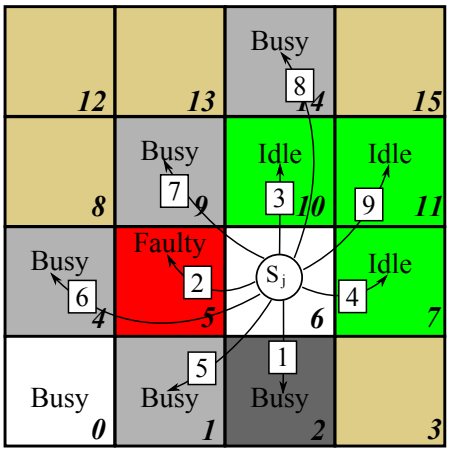

(a) Résultat de placement




(b) Résultat de Re-mapping




(c) Recherche pour le re-mapping de $S_j$




(d) Recherche pour le re-mapping de $T_{j1}$ à $T_{j3}$

Figure 8.3 – Exemple de mapping et re-mapping de DAG suivant la stratégie *Nearby Search*

Dans la suite, nous faisons l'hypothèse que la variable aléatoire $X$ suit une distribution binomial négative ([38]), c-à-d $X \sim \mathcal{NB}(n, P_s)$. Le Tableau 8.1 montre une analyse des probabilités suivant la loi binomial négative. Pour chaque probabilité de non-défaillance ($P_s$=0.9, 0.99, 0.999) et chaque nombre d'enfant ($n$=2, 10, 50, 200, 500), ce tableau donne le nombre maximale de défaillance $k$ avec une probabilité $c$ égale à 99, 95 et 65 %. Par exemple, considérant l'application illustrative de la Figure 8.1, le *Stream Leader* $S_i$ a 2 enfants $T_{i1}$ et $T_{i2}$ à mapper (c-à-d au moins deux essais). Si chaque noeud a un taux de fiabilité de 0.9, alors suivant le Tableau 8.1, il est garantit à 99% que pas plus de 2 noeuds défaillants seront visités. En conséquence, avec

Table 8.1 – Nombre de noeuds défaillant visité avec une probabilité $c$, $n$ enfants et un taux de fiabilité $P_s$

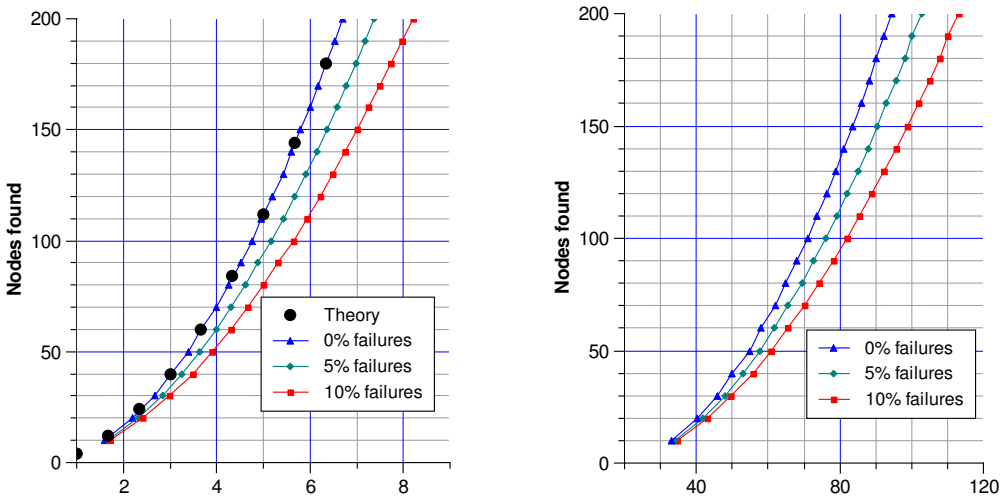
| $c$ | 99% | | | 95% | | | 65% | | |
|---|---|---|---|---|---|---|---|---|---|
| $P_s$ | 0.9 | 0.99 | 0.999 | 0.9 | 0.99 | 0.999 | 0.9 | 0 .99 | 0.999 |
| $n$=2 | 2 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| $n$=10 | 4 | 1 | 0 | 3 | 1 | 0 | 1 | 0 | 0 |
| $n$=50 | 11 | 2 | 0 | 10 | 2 | 0 | 7 | 1 | 0 |
| $n$=200 | 34 | 6 | 1 | 30 | 4 | 0 | 26 | 3 | 0 |
| $n$=500 | 75 | 11 | 2 | 68 | 9 | 2 | 62 | 7 | 1 |

au plus 4 essais, les 2 enfants pourront être mappés avec succès. Suivant le tableau 8.1, la stratégie de placement Neighbor Search garantit qu'avec un taux de fiabilité réaliste ($P_s > 0.9$), le nombre de noeud défaillant visité, et donc d'essais supplémentaire, sera négligeable, et ce même avec un grand nombre d'enfants. De plus, l'exécution de l'application est garantit même si des noeuds tombent en panne. En effet, notre technique permet de trouver efficacement les noeuds disponibles afin de re-mapper les tâches sans arrêter l'application.

### 8.2.6 Validation avec l'algorithme de routage tolérant aux fautes

Afin de valider les résultats théorique précédemment décrits, des simulations ont été réalisées sur la base du simulateur et des algorithmes de routage présentés dans [31, 6, 19, 20]. Dans [19], 3 algorithmes de routage adaptatifs sans deadlock sont présentés. Basé sur l'utilisation de 4 canaux virtuels, ces algorithmes combinent 2 réseaux virtuel ayant des restrictions dans l'ordre de liens afin d'éviter les deadlocks et de préserver l'adaptabilité. Ces algorithmes nécessitent que chaque routers soient au courant de l'état (c-à-d, défaillant ou non) des autres routers à 1 hop uniquement, ce qui est réalisé au travers de messages *I am Alive Messages*. Le premier algorithme, *Variant A*, sélectionne la plus haute direction dans la hiérarchie des directions, qui dépend de la relation entre la destination du message et le noeud courant. L'absence de deadlock et les boucles infinies sont garantis par 2 canaux virtuel. L'algorithme *Variant B* a, au prix d'une complexité plus importante, la possibilité de transférer un message d'un réseau à un autre. En effet, ce dernier nécessite le "poinçonnage" des noeuds afin de contrer les boucles infinies. Enfin, l'algorithme *Variant C* rend possible le mode écho fournissant une plus grande tolérance aux défaillances. En effet, si il existe au moins un chemin entre un noeud source et un noeud desti-

nation, l'algorithme *Variant C* garantit que le message arrivera à sa destination. Nous utilisons l'algorithme de routage *Variant A* du fait de sa plus faible latence et de sa moindre complexité. De plus, pour les taux de défaillances considérés ici (up to 10%) et se référents au Tableau 8.1 ainsi qu'à la Figure 8.4, l'algorithme *Variant A* procure une tolérance aux défaillances suffisante et permet de couvrir tous les chemins possible avec une grande probabilité. Cependant, si les défaillances deviennent plus importantes (ex. 20% 40%), un algorithme plus complexe (ex. *Variant C*) permettant une meilleur tolérance pourrait être aussi utilisé. Afin de simuler une maille 2D de $32 \times 32$ noeuds, un simulateur "cycle accurate" (précis au niveau cycle) a été développé, ce qui permet une simulation relativement précise. Les noeuds défaillant sont aléatoirement choisis parmi l'ensemble des noeuds (ex. 102 noeuds pour 10% de défaillance), en prenant garde de ne pas partitionner le réseau. Durant la simulation, le noeud central de la maille est utilisé pour mapper 200 noeuds ($n = 200$) en utilisant la stratégie *Nearby Search Strategy*. Il émet sequentiellement des requêtes à chacun de ses voisins en utilisant l'algorithme de routage *Variant A* ; les noeuds destination étant rangé par ordre croissant de leur distance Manhattan au noeud centrale. La simulation s'arrête après avoir reçu 200 acquittements positifs. La Figure 8.4 montre le résultat moyen de plus de 60 patterns différents pour chaque taux de défaillances. Pour tous les patternes, l'algorithme réussi à trouver 200 noeuds non-défaillant, en respectant le Tableau 8.1. La Figure 8.4(a) montre le nombre de hops moyen nécessaire pour différents $n$ et taux de défaillance $(1 - P_s)$. De plus, on peut constater que les jeux de points $(av\_hop, n\_nodes)$ (courbe bleu) résultant de la simulation (0% de défaillance), sont très proches de ceux d'analyses, obtenue des Equations 8.3 et 8.2 (cercles noirs). La Figure 8.4(b) montre le résultat de simulation concernant la latence moyenne (2-way) pour atteindre les noeuds, très proche des résultat théorique (Fig. 8.4(a)). Pour $n = 200$, les latences moyennes sont de 94.4, 103 and 113 [ns] et le nombre de hop moyen de 6.7, 7.4 et 8.2, pour respectivement 0%, 5% et 10% de défaillance. Par conséquent, même avec l'utilisation de l'algorithme *Variant A* qui est relativement simple, la dégradation de performance de notre solution reste raisonnable. Cependant, il est à noter qu'une tolérance plus importante peut être atteinte, si nécessaire, en utilisant un algorithme de routage plus complexe. De plus, ces simulations montrent comment notre technique permet de prendre en compte les incertitudes sur l'inter-connecte et au niveau application,




(a) Nombre moyen de hops

(b) Latence moyenne en *ns*

Figure 8.4 – Impact de la défaillance des noeuds dans une maille 2D de $32 \times 32$, utilisant la stratégie *Nearby Search* et l'algorithme de routage *Variant A*

et ce sans aucune connaissance de l'état du système. Notre solution est plus efficace que les technique basées sur le broadcast, qui nécessitent la collecte, le stockage et le maintien d'information de l'ensemble des noeuds et qui, par conséquent, devient prohibitif pour un réseaux de grande taille. A contrario, notre proposition consiste simplement en un certain nombre de step de recherche tel que décrit en Définition 5. Dans un large système multicoeurs dont le contexte change sans cesse, notre technique permet une bonne adaptabilité de l'application à l'état du système (Fig. 8.4) et ne requiert que peu de trafic et de puissance de calcul. En Section 8.2.4, nous avons présenté une technique de mapping, transparente à l'application, permettant de tolérer des patternes de défaillance variées. L'algorithme proposé est capable de mapper des tâches sur un nombre arbitraire de noeuds $n$ sans aucune connaissance sur l'état du réseau (Fig. 8.3(a)). En outre, notre technique supporte le re-mapping de tâches dont le noeud est défaillant, tel que décrit en Figure 8.3(b), Figure 8.3(c) et Figure 8.3(d). Enfin, en Section 8.2.6, a été présenté un algorithme de routage permettant d'implémenter efficacement notre solution.

## 8.3 Etude de cas : application Mjpeg-2000

Afin de démontrer la faisabilité de notre technique, nous l'avons analysée au travers d'une étude de cas sur une application réelle de décompression vidéo Mjpeg2000 [1], qui présente un fort potentiel de parallélisme. Avant de décrire l'application Mjpeg2000 ainsi que le détails des résultats obtenus, nous décrivons la méthode d'estimation employée.

### 8.3.1 Méthode d'Estimation

Sur la base de la Figure 8.2, les durées d'échange de messages entre un *Stream Leader* et ses enfants sont composées de quatre parties.

La première partie, donnée en Figure 8.4(b), représente la durée moyenne par noeud consacré pour la recherche d'un noeud libre non-défaillant. Par exemple, avec un taux de défaillance de 10%, pour trouver 100 noeuds non-défaillants, la latence moyenne par noeud nécessaire est de 82ns. Les deuxième et troisième partie, sont les durée de transfert du programme (code) et de ses données d'entrée. La quatrième, représente la durée nécessaire au transfert du résultat des enfants au *Stream Leader*.

Dans la suite, la phase d'*Initialization* ($Init$) est égale à la somme des trois premières parties alors que la phase *Result* ($Res$) correspond à la dernière. Les durées des phases d'*Initialization* ($\Delta_{Init}$) et de *Result* ($\Delta_{Res}$) sont présentées dans les Eq. 8.4 et 8.5.

$$\Delta_{Init} = \Delta_{search} + \Delta_{flit} * (codeSz + dataInSz) \quad (8.4)$$

$$\Delta_{Res} = \Delta_{flit} * dataOutSz \quad (8.5)$$

Où, $\Delta_{flit}$ est durée moyenne de transfert d'un flit, $\Delta_{search}$ est la latence moyenne pour trouver un noeud non-défaillant disponible, $codeSz$, $dataInSz$ et $dataOutSz$ sont respectivement la taille du code programme, la taille des données à traiter et la taille du résultat, tous exprimés en nombre de flits.

Le CCR (Computation-to-Communication Ratio), représenté à l'Equation 8.6, dérive des Equations 8.4 et 8.5. $CT$ correspond au temps de calcul d'un enfant.

$$CCR = \frac{CT}{\Delta_{Init} + \Delta_{Res}} \tag{8.6}$$

De plus, le facteur de réduction du temps d'exécution $RR$ (*Execution Time Reduction Ratio*) est calculé en divisant le temps de calcul séquentiel (*Sequential Execution Time*) par le temps d'exécution total (*Total Execution Time*), qui est la somme du temps d'exécution $CT$ d'un enfant avec les durées des phases d'*Initialization* $\Delta_{Init}$ et de *Result* $\Delta_{Res}$. Le temps d'exécution séquentiel peut être approximé par le temps de calcul d'un enfant $CT$, multiplié par le nombre d'enfants $n$.

$$RR = \frac{n.CT}{CT + \Delta_{Init} + \Delta_{Res}} \tag{8.7}$$

### 8.3.2 Mjpeg-2000 : Présentation

Du point de vue de l'encodeur ([1]), l'entrée est partitionnée de divers façon. Toutes les images indépendantes et chacune de leurs composantes (par exemple : Y,U,V qui sont aussi indépendantes) est décomposée en grille régulière de tuiles (*tile*) (notez que le nombre de tuile peut être égale à 1). Chaque tuile est décomposée en sous-bandes par l'IDWT (*Inverse Discret Wavelet Transform*), dépendant du nombre de niveaux de décomposition définit par l'utilisateur. Les sous-bandes sont, de plus, organisées en blocs réguliers appelés *code-blocks* afin d'être traités par l'encodeur tier-1. Puis, l'encodeur tier-2 encapsule les données dans une *stream*. En suivant [1], une image de taille $W$ par $H$, respectivement la largeur et la hauteur de l'image. Le nombre de tuile ($numXtiles$, $numYtiles$) est calculé sur la base de la taille de tuile définit par l'utilisateur, respectivement $XTsiz$ et $YTsiz$ dans les directions X et Y. Pour chaque tuile , il y a $N_r$ niveaux de résolution différents ($r$) (définit par l'utilisateur), résultant de nombreuses sous-bandes. La taille des code-block est aussi définit par l'utilisateur. Conséquemment, étant

donné la taille $x_{cb}$ et $y_{cb}$ d'un code-block ainsi que $XTsiz$ et $YTsiz$ la taille d'une tuile, le nombre correspondant de code-blocks $numXcblk$ et $numYcblk$ dans une tuile peut être calculé. La taille d'un code-block est restreinte par le standard et doit être dans $[2^2, 2^6]$ suivant chaque direction. De faite, le maximum de code-block est obtenu lorsque $x_{cb}$ et $y_{cb}$ sont tous deux égaux à $2^2$.

Soit une image CIF, de taille 352x288 pixels, avec une seule composante (par exemple la luminance) et une seule tuile. Considérant la plus petite taille de code-block possible, chacun d'eux sera de 16 éléments (*sample*) ($x_{cb} = y_{cb} = 2^2$). Alors, le nombre de code-block dans une image est $\frac{352}{2^2} \times \frac{288}{2^2} = 6336$. Dans ce cas, le code-block peut être vu comme de "grain fin" (seulement 16 samples).

Le précédent exemple, montre le potentiel de Mjpeg2000 pour un parallélisme très important. Ce parallélisme est principalement du parallélisme de donnée, ce qui est adéquation avec le modèle de DAG Fork-Join, même pour les petites tailles d'image. La partie décodage est principalement composée de 3 bloc fonctionnelles qui s'exécutent séquentiellement. En premier vient la fonction "dé-encapusaltion" qui est réalisée par le décodeur tier-2. Puis, le décodage arithmétique est effectuer par le décodeur tier-1, suivie par la transformé inverse en ondelette (IDWT) qui génère les tuiles décodées.

Le standard [1] définit comment les données sont traitées et organisées en *stream*. Ce document expose le grand potentiel de parallélisme, principalement de donnée, de partie décodeur du Mjpeg2000.

La Figure 8.5 montre une représentation détaillée du DAG fork-join d'une partie de décodage d'une tuile. Tous les termes utilisés peuvent être retrouvés dans le standard ([1]). *Habituellement, le bloc de décodage tier-1 est la partie du décodeur Mjpeg2000 qui est le plus consommateur en terme de temps de calcul. C'est pourquoi, notre analyse concernant la parallélisation du décodeur Mjpeg2000 sera se restreinte à cette partie. Les résultats sont donnés dans la Table 8.2.*

La durée du décodeur Tier-1, qui consomme le plus de temps de calcul, a été estimé expérimentalement. Les expérimentations ont été réalisées sur une station de travail équipé d'un processeur Intel à 2.4GHz et de 3GByte de mémoire, pour la séquence vidéo "waterfall" au format CIF à 30i/s (image par seconde). La Table 8.2 présente le temps de calcul moyen par

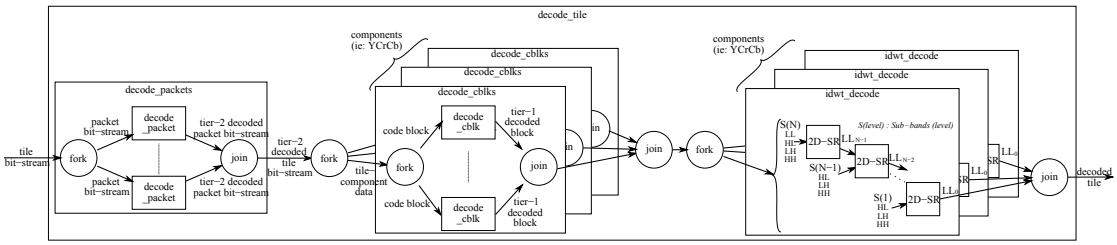



Figure 8.5 – Partie du décodeur Mjpeg 2000 représenté comme un DAG

code-block en colonne 3. La colonne 1 donne le nombre de *samples* par code-block (c-à-d la taille d'un code-block), allant de 16 (4x4) à 4096 (64x64).*samples*. Le nombre de code-block par image est représenté en Colonne 2. Les *samples* sont soit des entiers soit des réels (flottant simple précision) dépendant du choix fait durant l'encodage. Par exemple, avec une taille de code-block de 32x32, il y a 99 code-blocks par image. Qui plus est, c'est 99 tâches peuvent être exécutées simultanément, chacune d'elles ayant 1024 *sample* à traiter (exmpl. 4KB avec des entiers).

La précédente sous-section 8.3.1, nous permet d'estimer les différents paramètres clé pour le décodeur Tier-1. Un des paramètres clé est le temps de communication (*Communication time*), dérivé des Equ. 8.4 et Equ. 8.5, ainsi que de la Table 8.2 des colonnes 4 à 6 avec 0, 5 and 10% de défaillances. De l'implémentation du bloc Tier-1, la taille du code programme a été estimée autour de 4k flits sans aucune optimisation du compilateur. Les données à traitées sont inconnues à priori, car elles dépendent des paramètres de l'encodeur et de la vidéo originale. Cependant, considérant que l'application est une application de compression/décompression vidéo, un ratio au moins égale à $\frac{3}{2}$ est prise en compte. Alors, la taille des données ç décompresser est plus petite d'au moins $\frac{2}{3}$ par rapport à la taille de l'image originale (notée $Sample$ dans la Table 8.2 colonne 1). En conséquence, la durée totale de transfert (Communication time) entre le Stream Leader et tous ses enfants, est égale à $\Delta_{comm} = \Delta_{Init} + \Delta_{Res}$. Les Equations 8.4 et 8.5 amènent à $\Delta_{comm} = \Delta_{search} + \Delta_{flit} * (codeSz + \frac{2}{3} * sample + sample)$, où $sample$ est le nombre de sample par code-block, comme décrit dans la Table 8.2 colonne 1. La taille du code programme ($codeSz$) est égale à 4k flits, et le temps pour trouver un noeud libre non-défaillant $\Delta_{search}$) est donné par Figure 8.4(b). La durée d'un flit est (arbitrairement) fixé à 1ns. Le taux computation-

Table 8.2 – Tier-1 : estimation du temps de calcul et de communication pour un processeur à 2.4GHz

| Sample / cblk | cblk / frame | Comp. time / cblk ($\mu s$) | Comm. time($\mu s$) ($\Delta_{comm}$) at % of failure | | | Comp. to Comm. ratio ($CCR$) at % of failure | | |
|---|---|---|---|---|---|---|---|---|
| | | | 0% | 5% | 10% | 0% | 5% | 10% |
| 16 | 6336 | 5.06 | 5,04 | 5,23 | 5,65 | 1 | 0,967 | 0,896 |
| 64 | 1584 | 20.2 | 4,45 | 4,52 | 4,62 | 4,54 | 4,47 | 4,37 |
| 256 | 396 | 76.75 | 4,48 | 4,5 | 4,51 | 17,1 | 17,1 | 17 |
| 1024 | 99 | 253.99 | 5,19 | 5,19 | 5,2 | 48,9 | 48,9 | 48,8 |
| 4096 | 24 | 744.24 | 8,23 | 8,24 | 8,24 | 90,4 | 90,4 | 90,3 |

to-communication ratio ($CCR$) est estimé de l'Equ. 8.6 et reporté dans la Table 8.2, colonnes 7 à 9.

Suivant les colonnes 4 à 6 de la Table 8.2, le temps de communication est très faiblement impacté par le temps necéssaire à trouver un noeud libre non-défaillant ($\Delta_{search}$). Il ne représente qu'une petite portion du temps de communication, même à 10% de défaillance. De plus, pour les code-blocks dont la taille est supérieur à 256, le $CCR$ est plus élevé, allant de 17 à 90. Enfin, le taux de réduction du temps d'exécution $RR$ (*Execution Time Reduction Ratio*) obtenu est proche du cas idéal (c-à-d proche $n$). Par exemple, pour un code-block de 4096 (Table 8.2-dernière ligne), le temps de communication à 10% de défaillance (colonne 6) est de 8.24$\mu s$ et le temps d'exécution (colonne 3) est de 744.24$\mu s$. Le temps d'exécution séquentiel est obtenu en multipliant le nombre de code-blocks par le temps d'exécution d'un code-block ($24 \times 744.24\mu s$), ce qui donne 17.9$ms$. Alors, faisant l'hypothèse, que tous les messages peuvent être envoyés et reçus en parallèle, le $RR$ est de 23.7, ce qui est très proche du cas idéal de 24. Se basant sur les résultats obtenus, la stratégie proposée procurera un cadre d'exécution continu pour les applications parallèles sur un chip NoC multi-coeurs en présence de multiple noeuds et liens défaillants.

## 8.4 Conclusion

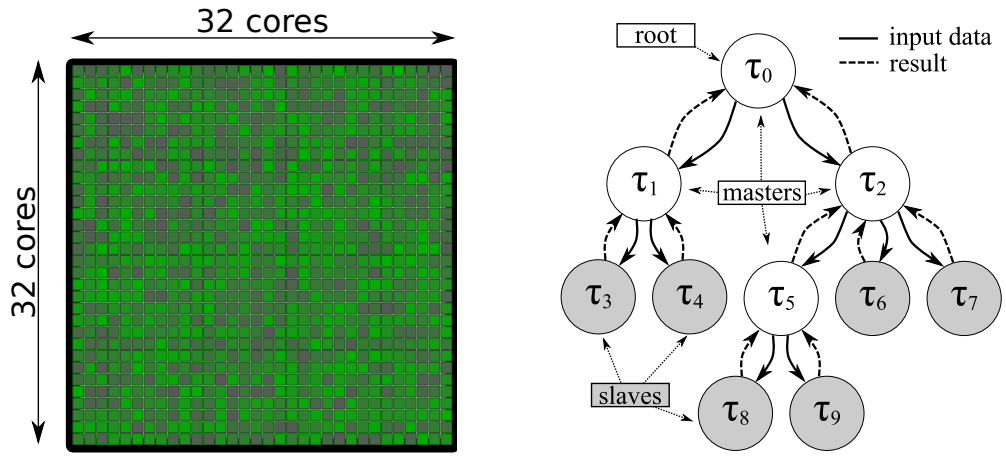
Dans ce chapitre, la stratégie d'auto-recouvrement proposée a été détaillée et supportée par différentes simulations et analyses théoriques. Premièrement, une application a été décomposée en DAG Fork-Join de tâches, qui garantie la résistance de notre approche en présence de

défaillance, comme décrit en Section 8.2.4. Par la suite, la Table 8.1 montre que la stratégie *Nearby Search* proposée ne nécessite que peu de step supplémentaires en présence de noeuds défaillants et supporte efficacement la stratégie d'auto-recouvrement. En Figure 8.4, les résultats de simulation confirment (c.f Sec. 8.2.6.) que la stratégie proposée est faisable en se basant sur l'algorithme de routage *Variant A*. Enfin, en Section 8.3, la stratégie proposée a été appliquée au décodeur MJpeg 2000. En Table 8.2, l'efficacité de notre approche est représentée en terme de ratio : temps de communication sur temps d'exécution. Il a été démontré que la stratégie proposée permet au décodeur MJpeg2000 d'être parallélisé et que le temps d'exécution en découlant est significativement réduit, pour approcher le cas idéal, c-à-d 24. De plus, il a été montré que l'application MJpeg2000 peut être représentée comme un DAG Fork-Join, ce qui permet une implémentation efficace de la stratégie d'auto-recouvrement d'applications parallèles en présence de défaillance dans un large Network on Chip, de plus de 1000 coeurs.

Chapitre 9

# Gestion de la consommation et des variabilités

**Sommaire**




(a) Présentation du système many-cores ciblé

(b) Exemple illustratif d'application master-slave hiérarchique

Figure 9.1 – Modèles d'Architecture et d'Application

## 9.1 Introduction

### 9.1.1 Modèle d'Architecture

Dans ce chapitre nous considérons un processeur many-core, tel que représenté en Figure 9.1(a), composé d'un millier de coeurs (ou noeuds) fabriqué dans une technologie dite agressive, 32nm et en deçà. Dans ce contexte, la variabilité et les défauts des différents composants posent de sérieux problèmes. Le processeur dépend d'un réseau d'interconnexion en maille 2D et dont la fréquence de chaque noeud peut être ajustée indépendamment suivant la puissance de calcul nécessaire ainsi que des variabilités observées. Pour les technologies deca-nanomètrique, la variabilité est un problème majeur. Les variations du processus de fabrication affectent principalement la tension de seuil $V_{th}$ des transistors, dont la moyenne est $\mu_V$ est généralement constituée de variations systématiques et de variations aléatoires. Que ce soit les variations systématiques ou aléatoires, toutes deux suivent une loi normale de moyenne zéro et d'écart type $\sigma_{rand}$ et $\sigma_{syst}$, respectivement. De plus, les variations systématique de deux transistors différents sont corrélées par facteur $\rho$, dépendant de leur distance relative. L'Equation 9.1 représente les variation de $V_{th}$

suivant les variabilités qui lui sont appliquée.

$$V_{th} = \mu_V + \mathcal{N}(0, \sigma_{rand}) + \mathcal{N}(0, \sigma_{syst}, \rho) \tag{9.1}$$

En outre, du fait d'une variabilité importante, vieillissement des transistors, température, etc..., des fautes permanente ou temporaire peuvent entraver l'exécution de l'application. Ainsi, la puce devra, nécessairement embarquer une circuiterie de test en ligne afin de détecter les fautes. Afin de simplifier, lorsqu'un coeur, un router ou un lien est défectueux, il devient silencieux. Il est possible d'ajuster la tension d'alimentation de chaque coeur indépendamment au moyen de sous système de $V_{dd}$-hopping, tel que proposé dans in [8]. Ainsi, la tension d'alimentation d'un coeur est positionnée soit à $V_h$ (valeur haute), soit à $V_l$ (valeur basse). Lorsqu'un coeur est inactif, le mode power-gating est utilisé. Le contrôleur local au coeur, change la tension d'alimentation afin d'offrir la puissance de calcul nécessaire avec le minimum d'énergie possible. Du fait des variabilités, les fréquences permises à tension basse ($f_l$) et à tension haute ($f_h$) varient d'un coeur à l'autre. Ainsi, une technique de détection et d'ajustement de celles-ci, permettant d'éviter les défauts de timming, est nécessaire.

L'énergie consommée par un noeud quelconque $n_i$ est donnée par l'Equation 9.2. $\alpha_1$, $\alpha_2$ et $\alpha_3$ sont des paramètres d'ajustement englobant les effets du facteur d'activité au niveau des gates, la capacitance globale, ainsi que des effets parasites additionnelles.

$$P(n_i, f) = \underbrace{\alpha_1 \cdot V_{dd} \cdot e^{\alpha_2 \cdot V_{dd}}}_{\text{leakage}} + \underbrace{\alpha_3 \cdot V_{dd}^2 \cdot f}_{\text{switching}} \tag{9.2}$$

Ainsi, l'énergie moyenne par cycle pour un noeud $n_i$ fonctionnant à une fréquence $f$ est donnée par l'Equation 9.3.

$$E_{n_i@f} = \begin{cases} \frac{\{f_h - f\}P(n_i,f_l) + \{f - f_l\}P(n_i,f_h)}{\{f_h - f_l\}f} & \text{if } f > f_l \\ \frac{P(n_i,f_l)}{f} & \text{if } f_l > f > 0 \\ 0 \text{ (power-gating mode)} & \text{if } f = 0 \end{cases} \tag{9.3}$$

Le réseau d'interconnexion est réseau en maille 2D asynchrone, utilisant un algorithme de routage adaptatif. En outre, l'impact de la variabilité sur le réseau d'interconnexion est considéré comme étant négligeable. Assumant qu'il existe peu de contention au niveau des routers, l'énergie consommée par l'interconnexion est alors proportionelle à la latence du message tout entier. Par conséquent, l'énergie consommée pour la transmission d'un flit est donnée par l'Equation 9.4, où $d_M(n_i, n_j)$ est la distance Manhattan entre le noeud $n_i$ et le noeud $n_j$. $\alpha_4$ inclut l'énergie requise pour la réception, le stockage, l'arbitration et l'émission vers le lien suivant d'un flit

$$E_{flit}(n_i, n_j) \quad = \alpha_4 \cdot d_M(n_i, n_j) \tag{9.4}$$

### 9.1.2 Modèle d'Application

Les applications de streaming peuvent être modélisées par un Graphe Acyclique Dirigé ou DAG ([82]), dans lequel les sommets représentent les tâches et les arcs, les communications intervenant entre les tâches. Les classes d'applications considérées sont périodiques avec des contraintes temps réel souple. Elles sont décrites par une structure DAG se conformant au modèle master-slave ([17]). Ce modèle est l'une des plus simple façon de paralléliser une application et de ce fait l'un des plus populaire en pratique. De façon générale, une tâche maître (master) décompose son travail en de plus petites tâches, les distribues parmi un ensemble d'esclaves (slave) et attend les résultats [109]. Chaque slave réalise son travail sur les données qu'il reçoit, puis retourne le résultat vers le maître qui réuni et assemble les résultats partiels afin de produire le résultat final escompté. Afin de surmonter le goulot d'étranglement de la centralisation vers le maître, un pattern master-slave hiérarchique est pris en compte [99], tel que décrit dans l'exemple illustratif de la Figure 9.1(b). Dans ce modèle, le maître au sommet de la hiérarchie est appelé tâche racine ou root. Le root partitionne les données d'entrées en destination des maîtres de niveau inférieur et ainsi de suite jusqu'au slaves qui traitent directement les données reçus. Dans la suite, les "sous" maîtres sont simplement nommés master et le maître au sommet de la

hiérarchie, simplement root. Les termes master et tâche parent d'une part, slave et tâche enfant d'autre part sont s de façon interchangeable. En outre, le flot d'exécution exacte est à priori inconnu et les masters ou sous-master peuvent créer de nouvelles tâches esclaves dynamiquement durant l'exécution. Au moment de leur création, la charge de travail moyenne $w_i$ (en nombre de cycles) de chaque tâche est connue. Entre une tâche maître $\tau_i$ et une de ses tâche esclave $\tau_j$, $c_{i,j}$ et $c_{j,i}$ sont la quantité de flits transmis, respectivement en entré (données à traiter) et en sortie (résultat du calcul). De plus, la vitesse de calcul $f_{app}$ est la fréquence à laquelle tout les tâches de l'application doivent être exécutées. La fréquence d'une tâche est définit par l'Equation 9.6 et représente le nombre de cycles de calcul requit par la tâche durant une période d'une seconde. Enfin, la fréquence minimale $f(n_i)$ d'un coeur $n_i$ est exprimé dans l'Equation 9.7 et dépend de la charge de travail totale des tâches qu'il lui sont allouées.

$$W_{n_i} = \sum_{\forall \tau_k \in n_i} w_k \tag{9.5}$$

$$f(\tau_k) = f_{app} w_k \tag{9.6}$$

$$f(n_i) = f_{app} W_{n_i} \tag{9.7}$$

### 9.1.3 Formulation du Problème

Etant donné un processeur constitué de $N \times N$ noeuds de calculs, inter-connectés par un réseau en maille 2D et connaissant la variabilité des coeurs (c-à-d,$(f_l, f_h)$, $\forall n_i$), placer les tâches d'une application en respectant ses exigences et de sorte à minimiser l'énergie consommée. L'énergie consommée inclue les contributions du réseau d'interconnexion ainsi que celui des noeuds de calculs (voir Section 9.1.1). Basée sur les Equations 9.3 et 9.4, l'Equation 9.8 représente l'énergie totale consommée par l'application. Dans la suite, le noeud de calcul où la tâche $\tau_i$ est placée, est noté $n(\tau_i)$.

$$E = \sum_{n_i} W_{n_i} E_{n_i @ f(n_i)} + \sum_{\tau_i, \tau_j \in app} c_{i,j} E_{flit}\left(n(\tau_i), n(\tau_j)\right) \tag{9.8}$$

De plus, l'Equation 9.9 représente le fait que la fréquence maximale du noeud $n_i$ est une contrainte pour les tâches applicatives.

$$\sum_{\forall \tau_k \in n_i} f(\tau_k) \quad \leq \quad f_h, \forall n_i \tag{9.9}$$

## 9.2 Stratégie de Placement Dynamique tenant compte de la Variabilité

Comme expliqué en Section 9.1.3, le but est de réduire l'énergie consommée. $\hat{E}(\tau_m, \tau_k, n_i)$ décrit dans l'Equation 9.10, représente l'estimation de l'énergie consommée si la tâche $\tau_k$ serait placée sur le noeud $n_i$, étant donnée la tâche master $\tau_m$.

$$\begin{aligned} \hat{E}(\tau_m, \tau_k, n_i) &= E_{with\ \tau_k} - E_{without\ \tau_k} + E_{comm\ \tau_k} \\ &= (W_{n_i} + w_k) \cdot E_{n_i @ (f(n_i)+f(\tau_k))} \\ &\quad - W_{n_i} \cdot E_{n_i @ f(n_i)} \\ &\quad + c_{m,k} \cdot E_{flit}(n(\tau_m), n_i) \\ &\quad + c_{k,m} \cdot E_{flit}(n_i, n(\tau_m)) \end{aligned} \tag{9.10}$$

Le sur-coût d'énergie, dû au placement de $\tau_k$ sur $n_i$, dépend de la charge de travail $W_{n_i}$ du noeud, ainsi que de l'augmentation de fréquence, nécessaire au maintien de la puissance de calcul requise.

### 9.2.1 Critère de recherche Adaptatif

Durant l'étape de recherche, permettant de minimiser l'énergie, chaque noeud est visité un par un. Afin d'éviter la visite de tous les noeuds du SoC, un critère d'arrêt est définit en Equation 9.11. Comme expliqué en Section 9.1.1, la distribution des caractéristiques des noeuds de calcul peuvent être approximées par une loi normale. Par conséquent, l'énergie minimum d'un noeud $E_{n_i @ f_l}$ (ou facteur énergétique) suit une loi normale $\mathcal{N}_E(\mu_E, \sigma_E^2)$, où les paramètres $\mu_E$ et $\sigma_E^2$ sont respectivement la moyenne et la variance de l'énergie par cycle pour le plus basse couple

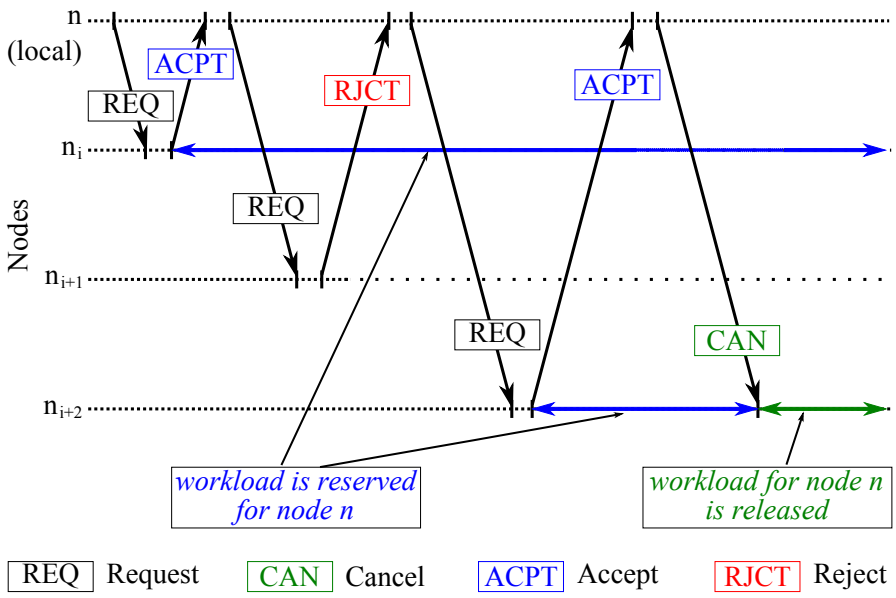



Figure 9.2 – Exemple de messages échangés durant la procédure de *Search & Map* (Cherche et Place).

tension/fréquence parmi tous les coeurs. Basé sur l'observation que $P[E_{n_i@f_l} > \mu_E - 3\sigma_E] \approx 99.73\%$, la borne inférieur de l'énergie consommée par une tâche est donné par l'Equation 9.11.

$$\begin{aligned} \hat{E}_{min}(\tau_m, \tau_k, n_i) &= w_k \cdot (\mu_E - 3\sigma_E) \\ &+ c_{m,k} \cdot E_{flit}(n(\tau_m), n_i) \\ &+ c_{k,m} \cdot E_{flit}(n_i, n(\tau_m)) \end{aligned} \tag{9.11}$$

Basé sur cette borne inférieur, la recherche de noeuds s'arrêtera lorsque l'énergie minimale possible du noeud suivant à visiter est plus grande que celle du meilleur noeud déjà trouvé.

### 9.2.2 Algorithme de placement Dynamique

L'Algorithme 9.1 est proposé afin de placer dynamiquement les tâches applicatives sur les noeuds du processeur. Il s'agit d'une forme générique, dans le sens que le "facteur de qualité" ($Q$) ainsi que le "facteur de seuil" peuvent êtres virtuellement remplacés par n'importe quel critère. Dans ce travail, le facteur de qualité générique $Q$ est remplacé par $\hat{E}$, issu de l'Equation 9.10 et

le facteur de seuil $Q_{th}$ par le critère énergétique $\hat{E}_{min}$ de l'Equation 9.11. La liste $listToVisit$ est la liste de tous les noeuds, ordonnés suivant leur proximité (du plus proche au plus éloigné). Cependant, un autre ordre peut être choisi, par exemple pour explorer un espace spécifique de la puce. Dans un souci de clarté, les caractéristiques d'une tâche $\tau_k$ sont dénommées par $TC_k$, où sont regroupées la charge de calcul ($w_k$), la charge de communication vers la la tâche parente (master) $\tau_m$ ($c_{m,k}, c_{k,m}$) et la fréquence moyenne ($f(\tau_k)$) requise. L'algorithme commence par vérifier si les besoins de la tâche peuvent être atteints sur le noeud local (line 5). Puis, cherche le premier noeud disponible parmi la liste de noeud $listToVisit$. C'est-à-dire, un noeud non-défaillant capable d'atteindre les besoins de la tâche. Tout d'abord, une requête est envoyée au noeud suivant de la liste (lignes 10 et 11). Si le noeud courant est à la fois, capable de répondre aux exigences (line 12) et permet de diminuer l'énergie estimée (line 13), alors il devient le meilleur noeud ($n_{best}$) (line 15), et le noeud précédent est "résilier" (line 14). Sinon, le noeud en cours de visite est résilier (line 18). La boucle en ligne 9 est répétée jusqu'à ce que tous les noeuds aient été visités ou bien que le meilleur noeud trouvé ait atteint la valeur du seuil. A la fin, si aucun noeud n'a été trouvé (ligne 26), la tâche est placée localement.

La procédure *onRequest* représentée par l'Algorithme 9.2 est exécutée par chaque noeud recevant un message de requête. Tout d'abord, les exigences de la tâche son vérifiées (line 2) et le facteur de qualité résultant est calculé (line 3). Puis, un message est envoyé en retour, afin de notifier si les exigences sont atteintes (line 5), ou non (line 7). Si un message *Accept* est envoyé (line 5), alors le noeud est réservé jusqu'à ce qu'un message *Cancel* (Algorithm 9.1, ligne 14) ait été reçu. Enfin, la tâche est démarrée dès que le code et les données entrante ont étés reçu de la tâche parent.

## 9.3 Expérimentation

La technique proposée a été simulée sur un modèle de simulation de haut-niveau spécifique. La simultanéité des messages de placement ainsi que l'algorithme de routage adaptatif (Variant A from [20]) ont été émulés. Basé sur la littérature, les valeurs des paramètres adéquates des Equations 9.2 et 9.4 ont été fixées approximativement. Durant la simulation, des applications

**Algorithme 9.1** DynamicMapping

**Entrée:** Task Characteristics $TC_k$, composed by the computation ($w_k$), the communication payload ($c_{m,k}, c_{k,m}$) and the required average frequency ($f(\tau_k)$).
**Entrée:** $\zeta$ is the "threshold relaxation factor" ($\in [0, 1]$)
**Entrée:** $listToVisit$ the list of nodes to visit

1: **procedure** SEARCH&MAP($TC_k$, $listToVisit$, $\zeta$)
2: $n_{best} \leftarrow \varnothing$
3: $Q_{best} \leftarrow \infty$

/*Compute the threshold factor and Check the requirement locally*/
4: $Q_{th} \leftarrow$ COMPUTETHRESHOLDFACTOR($TC_k$)
5: **if** CHECKREQUIREMENT($TC_k$) $= true$ **then**
6: $n_{best} \leftarrow s$
7: $Q_{best} \leftarrow$ COMPUTEQUALITYFACTOR($TC_k$)
8: **end if**

9: **while** $listToVisit \neq \varnothing$ and $\zeta \cdot Q_{best} > Q_{th}$ **do**
10: $n_i \leftarrow$ POP($listToVisit$)
11: $(status, Q) \leftarrow$ SENDREQUEST($TC_k$, $n_i$)
12: **if** $status =$ Accept **then**/*requirements are achievable*/
13: **if** $Q < Q_{best}$ **then**
14: SENDCANCEL($n_{best}$)/*only if $n_{best} \notin \{local, \varnothing\}$*/
15: $n_{best} \leftarrow n_i$
16: $Q_{best} \leftarrow Q$
17: **else**
18: SENDCANCEL($n_i$)
19: **end if**
20: **else if** $status =$ Reject **then**/*rqm are not achievable*/
21: **else**/*Destination node is unreachable.*/
22: /*Status is "NoAnswer" or "NAcK" (node, routers or links are faulty).*/
23: **end if**
24: $Q_{th} \leftarrow$ COMPUTETHRESHOLDFACTOR($TC_k$)
25: **end while**

26: **if** $n_{best} = \varnothing$ **then**
27: $n_{best} \leftarrow local$
28: **end if**
29: **return** $n_{best}$
30: **end procedure**

avec un nombre de tâches allant de 100 à 700, ont été placées sur le processeur constitué de 1024 coeurs. Les graphes d'application, ainsi que leurs caractéristiques on été générés aléatoirement, avec un ratio calcul/communication (CCR : Computation to Communication Ratio) proche de 1. Différents scénario de variabilité ont été utilisés (Tableau 9.1) afin de refléter d'éventuelles évolutions dans les futurs technologies CMOS. Les défaillances ont été injectées au niveau des noeuds, avant que le placement des tâche ne commence.

**Algorithme 9.2** onRequest

**Entrée:** Task Characteristics $TC_k : w_k, c_{m,k}, c_{k,m}, f(\tau_k)$
**Entrée:** Master task node $n_m$

1: **procedure** ONREQUEST($TC_k$, $n_m$)
2: **if** CHECKREQUIREMENT($TC_k$) $= true$ **then**
3: $Q \leftarrow$ COMPUTEQUALITYFACTOR($TC_k$)
4: mark myself as reserved for $TC_k$, assigned to node $n_m$
5: SENDACCEPT($Q$, $n_m$)
6: **else**
7: SENDREJECT($\varnothing$, $n_m$)
8: **end if**
9: **end procedure**

| Scenario | Sc. 1 | Sc. 2 | Sc. 3 |
|---|---|---|---|
| $\sigma_{rand}/\mu_V$ | 6 | 24 | 24 |
| $\sigma_{syst}/\mu_V$ | 6 | 0 | 12 |

Table 9.1 – Scénario de Variabilité

La Figure 9.3 décrit l'énergie totale consommée par l'application pour différente quantité de tâches et différents facteur de relaxation. Avec le scénario de variabilité 2, en l'abscence de défaillance, l'énergie consommée par l'application est significativement réduite du fait de l'utilisation de la stratégie *Self-Adaptive Nearest Neighbor* ($\zeta = 0.6, 1$) comparé à une utilisation de pure Nearest Neighbor ($\zeta = 0$). De plus, le facteur de relaxation $\zeta$ permet un compromis entre éfficacité énergétique et le coût en terme de communication nécessaire au placement.

La Figure 9.4 montre la relation entre énergie consommée, variabilité et facteur de relaxation. Globalement, plus la variabilité est importante et plus le facteur de relaxation doit être grand pour garantir l'efficacité de l'application. Cette même tendance est aussi observée lorsque le nombre de défaillances injectées augmente. En effet, lorsque les différences entre les noeuds augmentent, le meilleur noeud à utiliser peut être très distant, ce qui oblige à visiter plus de noeuds moins efficace avant de trouver le meilleur noeud.

Enfin, la Figure 9.5 donne le détail de l'énergie consommée par l'application, sous différents scénario de variabilité et de défaillance. L'énergie globale de l'application augmente avec la variabilité et le nombre de défaillance. Cependant, la stratégie de placement permet de limiter les dégradations en équilibrant l'énergie consommée durant le calcul et celle consommée durant

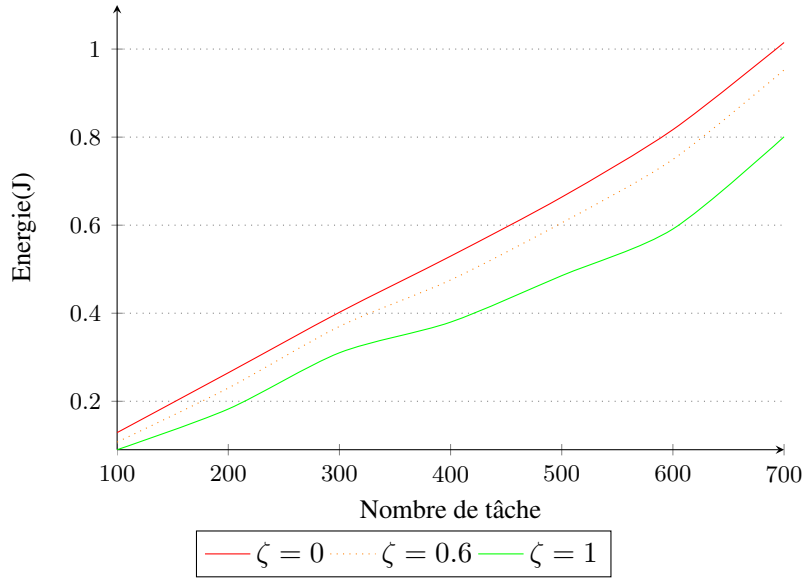



Figure 9.3 – Energie consommée vs. charge de travail (en nombre de tâche), pour différents facteurs de relaxation, suivant le scénario de variabilité sc.2

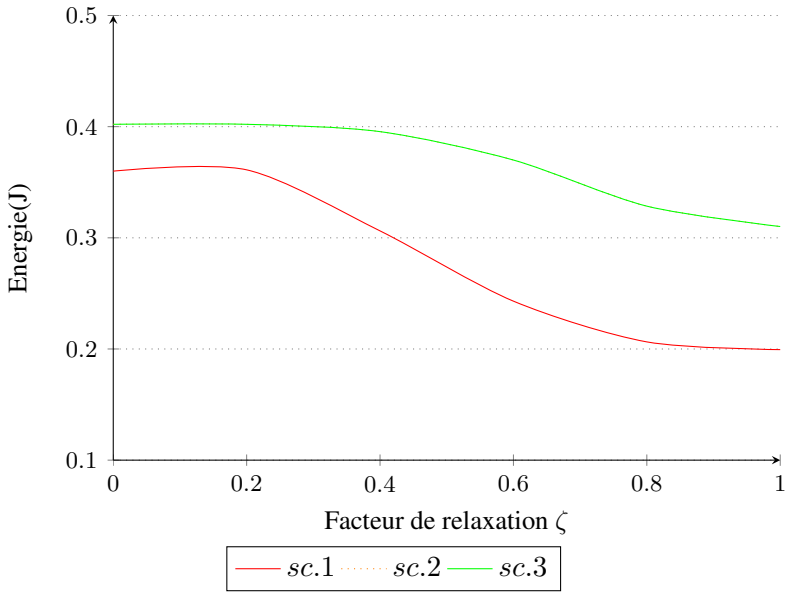



Figure 9.4 – Energie consommée vs. facteur de relaxation $\zeta$, pour différents scénario de variabilité. (un petit $\zeta$, relâche "l'agressivité" énergétique.)

les communications.

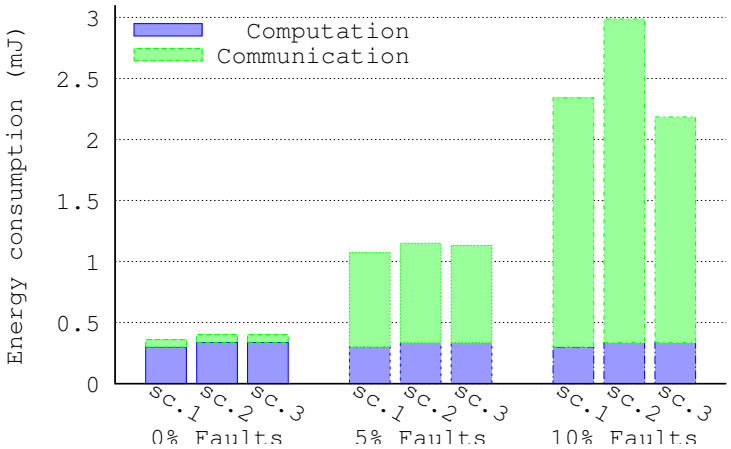



Figure 9.5 – Energie consommée par l'application sous différentes conditions processeur (taux de défaillance, variabilité)

## 9.4 Conclusion

Dans ce chapitre, une technique permettant d'exécuter des applications parallèles dans les futurs processeurs many-core peux fiables a été présentée. Celle-ci est capable de placer dynamiquement les tâches applicatives que ce soit à la demande ou au démarrage de l'application ou après une défaillance et permet de réduire la consommation du fait de la variabilité des coeurs. Basée sur les simulations exposées en Section 9.3, la technique proposée permet de réduire efficacement la consommation d'énergie en présence de variabilité dû au processus de fabrication. De plus, le facteur de relaxation $\zeta$ peut être affiné afin d'ajuster l'énergie préservée et permettre de contre balancer la variabilité du processus de fabrication. En outre, cette technique permet de limiter la dégradation d'énergie dûe aux défaillances, comme il a été montré en Figure 9.5. Cette approche est entièrement générique et pourra être améliorée, afin de prendre en compte d'autres aspects, tel que la température, la bande passante maximale des noeuds ou des routers ou encore la congestion des messages.

# Quatrième partie

## Conclusion Générale et Perspectives

**Sommaire**

Chapitre 10

# Conclusion

Dans cette thèse, trois parties ont été présentées Partie I, Partie II et Partie III. Dans la Partie I, il a été introduit le contexte et les motivations dans lequel se placent ces travaux. Puis, nous avons brièvement décrit le SoC ARAVIS, SoC devant permettre de répondre aux exigences de performances, de fiabilité et de robustesse mais aussi en terme de consommation énergétique. Enfin, pour clore la première partie, un état de l'art en relation avec notre contexte à été présenté. La Partie II a été consacrée à l'exposé d'une proposition de méthodologie permettant le placement et l'ordonnancement de tâches d'application sur l'architecture 2D mesh de cluster du SoC ARAVIS. Nous avons, dans cette partie, tout d'abord présenté la méthodologie dans ses grands principes en survolant les 3 étapes qui la composent. Puis, pour chaque étape, nous avons consacré un chapitre les présentant plus en détails. Plus particulièrement, les Chapitres 4, 5 et 6 détails respectivement l'étape 1 de *Placement/Ordonnancement Locale hors-ligne*, l'étape 2 de *Placement/Ordonnancement Globale enligne* et l'étape 3 de *Placement/Ordonnancement à l'exécution*. Enfin, le dernier Chapitre (7) de cette partie, a été dédié à l'expérimentation des différentes étapes, chacune de façon séparée.

La méthodologie (c.f. Chap. 3) que nous avons proposée devrait permettre de traiter différents aspects du placement/ordonnancement tel que l'exploitation et la mise en correspondance des hiérarchies applicative et architecturale ainsi que la prise en compte des différents paramètres intervenant dans les phases de conception et d'exécution.

Afin de résoudre l'étape 1, nous avons proposé deux approches pour réaliser le placement/ordonnancement au sein d'un cluster. Ces deux approches, avec et sans pipeline logiciel, ont été d'une part formulées mathématiquement sous forme ILP et d'autre part traitées à l'aide de multiples heuristiques de liste usuels (c.f. Chap. 4). Leur mise en oeuvre, lors d'expérimentations (c.f. Chap. 7.2), ont permis d'obtenir des résultats satis-

faisants, tant au niveau de l'accélération que de l'efficacité. En effet, il a été montré que l'ensemble des algorithmes utilisés, donnaient des performances tout à fait acceptables lorsque l'application présente un parallélisme suffisant. D'autre part, il a été montré que les algorithmes de pipelines mises en oeuvres montraient de bonnes performances, mêmes lorsque le parallélisme inhérent était relativement faible. De plus, ces derniers, couplés à une interconnexion intra-cluster plus performante permettraient d'atteindre des caractéristiques quasi idéales lorsque l'application est périodique.

L'étape 2 a été résolue à l'aide de techniques d'optimisation multiobjectifs (c.f. Chap. 5). Se basant sur un algorithme génétique, les différents modèles de performance, de consommation ainsi que les fonctions objectifs ont été définis. De plus, une technique permettant la prise en compte de différents modes de fonctionnement (Idle/DVFS) a été proposée et intégré aux modèles. La mise en oeuvre de cette étape a permis, lors de l'expérimentation, d'obtenir de bon résultats (c.f. Chap.7.3). En effet, il a été montré une très bonne adaptabilité aux variabilités dû au processus de fabrication avec des gains en performance de 40 à 55% et des gains en énergie de 30 à 57%.

Le rôle de l'étape 3 est de sélectionner, durant l'exécution, une combinaison de point de fonctionnement parmi les courbes des différents groupes actif. Cette combinaison, doit permettre de respecter les contraintes temporelles tout réduisant le plus possible la consommation d'énergie. La formulation de ce problème a été décrite puis résolue par l'utilisation d'un algorithme heuristique issue de [127] (c.f. Chap. 6). Il a été souligné que ce dernier ne permettait pas de résoudre pleinement notre problème. En effet, celui-ci ne permet pas de prendre en compte l'aspect spatiale d'un placement de tâche sur une cible architecturale de type 2D mesh. Ce qui a, pour conséquence, une mauvaise estimation du temps d'exécution globale. En réalité, cela se traduit par une surestimation

systématique lorsque l'ensemble des noeuds du SoC ne sont pas tous utilisés. Cependant, il a été noté que lorsque la taille du couple architecture - application est appairée, cet algorithme est tout à fait satisfaisant.

Les différents Chapitres composant la Partie II ont permis de démontrer la faisabilité de notre proposition de méthodologie. Il a été montré que cette dernière permettait de prendre en compte les aspects performance et consommation dans un environnement MpSoC massivement parallèle sujet aux variabilités du processus de fabrication. De plus, sa capacité à traiter des applications complexes et de (relativement) grande taille a été mis en évidence. Cependant, dans cette partie, l'aspect fiabilité reste absent.

Dans la Partie III, nous avons voulue, d'une part palier à ce manque et d'autre part apporter une réponse plus dynamique aux problème de placement/ordonnancement. Dans cette partie deux nouvelles propositions ont été faites. D'une part, sur l'aspect fiabilité, avec la proposition au Chapitre 8 d'une stratégie tolérante aux défaillances des noeuds et liens du réseau 2D mesh. D'autre part, en l'étendant aux aspects performance, consommation et variabilité avec une proposition au Chapitre 9 d'algorithmes distribués prenant en compte ces aspects dynamiquement.

Une stratégie d'auto-recouvrement (c.f. Chap. 8) a été proposée, détaillée et supportée par différentes simulations et analyses théoriques. La décomposition DAG Fork-Join d'une application a été réalisée, garantissant, ainsi, la résistance de notre approche en présence de défaillance. De plus, il a été montré que la stratégie *Nearby Search* proposée ne nécessite que peu de "step" supplémentaire en présence de noeuds défaillants et supporte efficacement la stratégie d'auto-recouvrement. Enfin, la stratégie a été appliquée au standard MJpeg 2000, une application réelle de décompression vidéo. L'efficacité de notre approche a été mise en évidence et a permis de démontrer que la stratégie proposée

permet au décodeur MJpeg2000 d'être parallélisé avec un temps d'exécution significativement réduit. Et ce en présence de défaillance dans un réseau 2D mesh de plus de 1000 coeurs.

Afin d'étendre aux aspects performance, consommation et variabilité la stratégie d'auto-recouvrement, une technique permettant de placement dynamique a été proposée (c.f. Chap. 9). Celle-ci permet de placer dynamiquement les tâches applicatives, que ce soit à la demande ou au démarrage ou bien encore après une défaillance. Permettant, ainsi, de réduire la consommation dû aux variabilité des coeurs. Il a été montré, au travers de simulations, que la technique proposée permettait de réduire efficacement la consommation d'énergie en présence de variabilité dû au processus de fabrication.

De plus, à l'intérieur des algorithmes distribués qui ont été proposés, le facteur de relaxation $\zeta$ permet d'ajuster le niveau d'énergie préservée et permet de contre balancer la variabilité du processus de fabrication. Enfin, il a été montré que cette technique permettait de limiter la dégradation d'énergie dû aux défaillances.

Chapitre 11

# Perspectives

Les perspectives de ce travail sont multiples et diverses. Dans la suite, nous présentons les différentes perspectives envisagées et envisageables suivant leur échéances, du court terme à plus long terme.

Tout d'abord, concernant l'étape 1. Nous avons présenté divers algorithmes, avec et sans pipeline logiciel, ciblant un cluster ayant un bus partagé comme interconnexion locale. La première perspective envisagée concerne la synchronisation intra-cluster. En effet, lors du placement/ordonnancement, l'ensemble des paramètres du processeur alloué, date de début, date de fin sont déterminé. Cependant, aucun mécanisme logiciel et/ou matériel ne permet de garantir la synchronisation des différentes tâches et communications entre les processeurs. Ainsi, il est envisageable d'étudier plus en profondeur divers mécanismes matériel et/ou logiciel permettant de réaliser ceci. De plus, la mise en oeuvre des machines à états permettrait d'en assurer tout ou partie. D'autre part, comme il a été signifié, le bus partagé est un goulot d'étranglement pour les applications dont les communications sont importantes. Il serait, donc, judicieux d'étendre les algorithmes avec d'autres interconnexions locales, tel que cross-bar, bus hiérarchique ou NoC. Enfin, la prise en compte de l'utilisation mémoire doit faire l'objet d'investigation afin de pouvoir permettre une gestion plus dynamique et garantir le bon fonctionnement.

Durant l'étape 2, l'exploration des solutions est réalisée séparément pour chaque groupe de TC sur l'ensemble du SoC. La probabilité que deux placements, issus de deux groupe différents, aient au moins un noeud ou un lien différent est très forte. Le problème se situe durant l'exécution de l'étape 3, qui ne prend pas en compte l'aspect spatiale et ne permet donc pas de faire une estimation correcte des performances. En effet, une simple somme des temps d'exécution n'est pas une mesure correct lorsque, par exemple, les points de fonctionnement choisis pour les groupes actifs n'utilisent

aucun noeud et/ou lien en commun. Ainsi, une investigation dans ce sens sur l'étape 2 et/ou sur l'étape 3 est nécessaire, afin de traiter ce problème.

L'architecture ciblée est un SoC composé de clusters de processeurs, organisé en maille 2D autour d'un NoC. Chaque cluster possède, suivant le nombre de coeurs, au plus, quelques centaines de kilo-octets de mémoires. Au niveau du SoC, la mémoire disponible est alors, suivant le nombre de noeuds, de l'ordre de quelques mega-octets. Par exemple, avec 16KB de mémoire de donnée par processeur, 8 processeurs par cluster et 16 (4x4) clusters, c'est 2Mo de mémoire (rapide) pour l'ensemble du SoC. Cependant, bon nombre d'applications, auront besoin d'une quantité de mémoire plus importante, ce qui nécessite d'utiliser une ou plusieurs mémoire de masse, typiquement DDR. Il serait intéressant d'explorer et d'intégrer cet aspect à notre méthodologie.

Une autre perspective à cours terme est la fusion des différentes étapes, afin de permettre des expérimentations sur une ou plusieurs application réelles. De même, le couplage des propositions de la méthodologies avec celles de la troisième partie, offrirait plus de robustesse tout en y ajoutant plus d'aspect dynamiques, ce qui augmenterait les capacités d'adaptations à l'environnement d'exécution.

# Annexe

## Sommaire

Annexe A

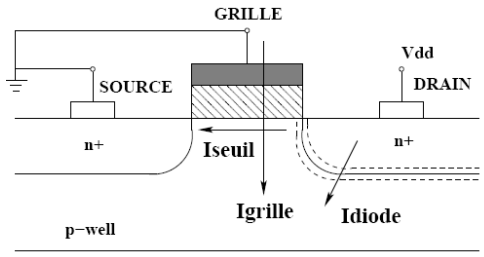
# Les SoCs

**Sommaire**

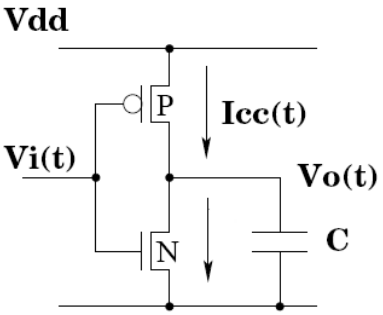
# A.1 La Consommation d'énergie dans les SoCs

## A.1.1 La technologie CMOS

La consommation des circuits CMOS inclut deux composantes : statique et dynamique. La puissance statique ($P_{leak}$) est dûe aux caractéristiques technologiques de fabrication et aux imperfections (intrinsèques) des matériaux utilisés pour la fabrication des transistors. Elle se caractérise à température ambiante, essentiellement au travers de trois courants qui sont : le courant sous le seuil $I_{subn}$ (*subthreshold current*), le courant de polarisation de diode en inverse $I_j$ (*reverse bias PN junction current*) et le courant à travers la grille $I_{gate}$ (*gate leakage current*). La figure A.1(a) représente un transistor en coupe avec les différents courants précédemment cités. Le modèle de consommation de transistor utilisé est décrit dans [89]. Celui-ci est aussi exploité dans d'autre travaux tel que [131], [91] et [4]. Dans la suite, ce modèle est succinctement explicité.




(a) Principaux courants de fuites dans un transistor CMOS




(b) Courant de court-circuit d'un inverseur CMOS

Figure A.1 – Consommation statique et dynamique des transistors CMOS

### A.1.1.1 Tension de seuil

La tension de seuil d'un transistor MOSFET à canal court dans le modèle BSIM ([22], [84]) est :

$$V_{th} = V_{th0} + \gamma(\sqrt{\Phi_s - V_{bs}} - \sqrt{\Phi_s}) - \theta_{DIBL}V_{dd} + \Delta V_{NW} \quad \text{(A.1)}$$

où $V_{th0}$ est la tension de seuil à polarisation zéro (*zero-bias threshold voltage*), $\Phi_s$, $\gamma$, $\theta_{DIBL}$ sont des constantes dépendant de la technologie, $V_{bs}$ est la tension appliquée entre la source et le corps du transistor, $\Delta V_{NW}$ est une constante modélisant les effets "faible distance", et $V_{dd}$ la tension d'alimentation. Si $|V_{bs}| \approx \Phi_s$, alors $\sqrt{\Phi_s - V_{bs}} - \sqrt{\Phi_s}$ peut être linéarisé en $k.V_{bs}$ et l'Equation A.1 devient :

$$V_{th} = V_{th1} - K_1.V_{dd} - K_2.V_{bs} \tag{A.2}$$

où $K_1$, $K_2$ et $V_{th1}$ sont constantes.

### A.1.1.2 Fréquence de fonctionnement

Le délai d'une porte est une fonction de la tension d'alimentation et de la tension de seuil des transistors la constituant. Puisque le délai de portes complexes reste proportionnelle au délai d'un inverseur standard, le délai d'un chemin peut être modélisé comme étant un modèle *alpha-power* d'un inverseur [102], [15] tel que :

$$t_{inv} = \frac{L_d.K_6.V_{dd}}{(V_{dd} - V_{th})^{\alpha}} \tag{A.3}$$

où $L_d$ est la longueur du chemin logique [48], $K_6$ est constante pour une technologie donnée et $\alpha$ mesure la vélocité de saturation. En substituant A.2 dans A.3 nous obtenons la fréquence de fonctionnement :

$$f = (L_d.K_6.V_{dd})^{-1}.((1 + K_1).V_{dd} + K_2.V_{bs} - V_{th1})^{\alpha} \tag{A.4}$$

### A.1.1.3 Puissance Consommée

La puissance totale consommée est la somme des puissances dynamique $P_{AC}$, statique $P_{DC}$ et de court circuit $P_{cc}$.

$$P = P_{AC} + P_{DC} + P_{cc} \tag{A.5}$$

La puissance de court-circuit $P_{cc}$ n'étant consommée que lors de très court instant (unique-

ment pendant transition du signal), elle ne participe que très faiblement et est, par conséquent, négligeable ([123]).

La puissance dynamique $P_{AC}$ est égale à :

$$P_{AC} = C_{eff}.V_{dd}^2.f \tag{A.6}$$

où $C_{eff}$ est la capacitance moyenne de commutation par cycle et $f$ est la fréquence d'horloge.

La majorité du courant statique dans un inverseur standard, provient de la conduction sous le seuil ([48], [114]), mais aussi du courant de polarisation inverse de la jonction, qui peut être une contribution significative ([71], [21]). La puissance statique $P_{DC}$ est égale à :

$$P_{DC} \approx V_{dd}.I_{subn} + |V_{bs}|(I_{jn} + I_{bn}) \tag{A.7}$$

où $I_{subn}$ est le courant de fuite sous le seuil, $I_{jn}$ et $I_{bn}$ sont les courants de fuites entre d'une part le drain, d'autre part la source et la jonction au substrat dans les transistors NMOS.

Le courant de fuite sous le seuil est modélisé par :

$$I_{subn} = (\frac{W}{L})I_S[1 - e^{\frac{-V_{dd}}{V_T}}].e^{\frac{-(V_{th}+V_{off})}{n.V_T}} \tag{A.8}$$

où $W$ et $L$ sont les dimensions géométrique du transistor, $I_S$, $n$ et $V_{off}$ sont des constantes déterminées empiriquement pour un procédé donné, et $V_T$ est la tension thermique [22]. Typiquement, $V_{off}$ est petit et $1 - e^{-V_{dd}/V_T}$ est proche de 1, quelques soit $V_{dd}$. De cette approximation et en substituant A.2 dans A.8, nous obtenons :

$$I_{subn} = K_3 e^{K_4.V_{dd}}.e^{K_5.V_{bs}} \tag{A.9}$$

où $K_3$ à $K_5$ sont des constantes s'adaptant aux paramètres.

Lorsque $|V_{bs}|$ augmente, le courant de fuite de la jonction augmente et contre-réagit avec l'économie faite en diminuant $I_{subn}$

La valeur maximale pour $|V_{bs}|$ avant que le courant de fuite de la jonction ne surpasse la réduction de courant sous le seuil dépend du procédé et varie entre -0,6 et 2,5V, comme cela est montré dans [83] et [72].

Ce point de croisement est aussi dépendant de la température, qui lorsqu'elle est plus élevée, permet une plus grande réduction de $I_{subn}$ (et donc tolère un plus grand $|V_{bs}|$) avant que $I_j$ augmente [83].

Pour les technologies sub-microniques profondes, 45nm et en dessous, le courant de fuite de grille $I_{gate}$ devient équivalent à $I_{subn}$. Ce courant est dû à l'effet tunnel de la jonction grille-oxyde et aussi à l'injection de porteurs chauds. [74] en présente un modèle simplifié :

$$I_{gate} = K_3.W.(\frac{V_{dd}}{T_{ox}})^2.e^{\frac{-\alpha.T_{ox}}{V_{dd}}} \tag{A.10}$$

où $K_3$ et $\alpha$ sont des constantes déterminées empiriquement, $T_{ox}$ est l'épaisseur d'oxyde.

Bien que, suivant l'Equation A.10, l'augmentation de $T_{ox}$ semble être bénéfique, cela reste très limité du fait de la réduction de la taille du transistor dans les technologies sub-micronique profonde. Cependant, en utilisant certains matériaux, tel que *Hight-K* en place de l'oxyde permet de réduire considérablement le courant de fuite de grille

$I_j$ peut être approximé comme étant constant, alors le courant statique total $I_{leak}$ devient :

$$I_{leak} = K_3 e^{K_4.V_{dd}}.e^{K_5.V_{bs}} + I_j \tag{A.11}$$

En substituant A.8 et A.9 dans A.7, nous obtenons :

$$P_{DC} = V_{dd}.K_3 e^{K_4.V_{dd}}.e^{K_5.V_{bs}} + |V_{bs}|I_j \tag{A.12}$$

et la puissance totale consommée devient :

$$P = C_{eff}.V_{dd}^2.f + V_{dd}.K_3 e^{K_4.V_{dd}}.e^{K_5.V_{bs}} + |V_{bs}|I_j \tag{A.13}$$

Ainsi, l'énergie consommée par cycle est définit comme la puissance instantanée multipliée par la durée d'un cycle. En utilisant A.13 l'énergie totale consommée par cycle $E_{cyc}$ pour un circuit est donnée par :

$$E_{cyc} = C_{eff}.V_{dd}^2 + L_g f^{-1}(V_{dd}.K_3 e^{K_4.V_{dd}}.e^{K_5.V_{bs}} + |V_{bs}|I_j) \tag{A.14}$$

### A.1.2 Réduction de la consommation d'énergie

Il est souvent fait état de "réduction de la consommation", en réalité, il s'agit bien de réduire l'énergie et pas seulement la puissance instantanée (cf. Eq. A.14). Comme le montre la figure A.2, deux puissances différentes peuvent correspondre à deux énergies identiques. Contrairement à l'énergie consommée qui a pour conséquence (entre autre) la décharge des batteries, la puissance instantanée, elle, pose des problèmes de dissipation thermique et de rupture de composants.

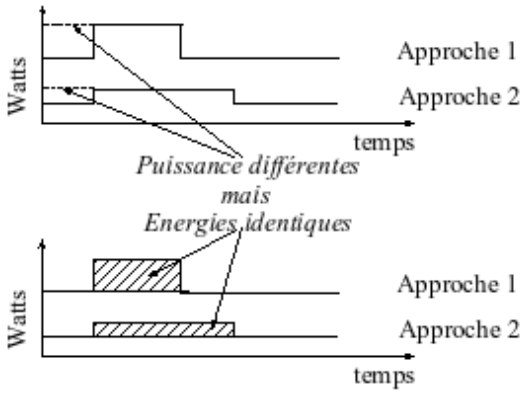



Figure A.2 – Relation entre Puissance et Energie

La consommation des SoC, comme décrit dans la sous-section A.1.1, est composée de la consommation statique et de la consommation dynamique.

Les courants de fuites augmentent dramatiquement avec la réduction de la technologie. En particulier, avec la réduction de $V_{th}$ (afin d'atteindre de hautes performances), la consommation statique devient une composante significative de la puissance totale consommée que ce soit

dans le mode de fonctionnement actif ou en veille. Partant de ce constat, pour supprimer la puissance consommée dans les circuits à faible tension, il est nécessaire de réduire les courants de fuites dans les deux modes, actif et de veille. Cette réduction doit être accomplie en utilisant des techniques allant du niveau processus jusqu'au niveau système, en passant par le niveau circuit. Au niveau procès, la réduction des fuites peut être réalisée en contrôlant les dimensions (longueur, épaisseur d'oxyde, profondeur de jonction etc...) ainsi que le profil de dopage des transistors. Au niveau circuit, la tension de seuil et le courant de fuite des transistors peuvent être effectivement contrôlés au travers des tensions des différents points d'accès (drain, source, gate, body). Différentes techniques au niveau processus et circuit sont présentées dans [101]. La plupart de ces techniques proposent de contrôler le courant de fuite sous le seuil, certaines adresses le contrôle des autres composantes du courant de fuite.

#### A.1.2.1 Réduction des courants de fuites

Comme dit précédemment, parmi les courants de fuites présent dans un transistor, les courants de seuil ($I_{subn}$ ) et de grille ($I_{gate}$) sont les deux prédominants. Différentes techniques permettent de réduire le courant de seuil, comme les approches "multi-VT", "Dual-VT" ou d'"Adaptative body biasing".

La technique de *Dual-Vt* consiste à utiliser deux types de transistors, l'un avec un petit $V_{th}$ faible (LVT : *Low VT*) sur le chemin critique et l'autre avec un $V_{th}$ plus important (HVT : *Hight VT*) hors du chemin critique. Ceci permet de réduire les courants de fuite (avec HVT) , tout en gardant de bonnes performances sur les chemins critiques (LVT).

Une variante de ce dernier est le multi-Vt ou MTCMOS (Multi Threshold CMOS)([56]) construit autour de deux transistors avec comme précédemment un $V_{th}$ élevé et un $V_{th}$ faible, si ce n'est qu'ici, ils sont utilisés par paire de façon à obtenir un $V_{th}$ bas lors d'un fonctionnement normal et un $V_{th}$ important en mode repos. La figure A.3 représente un inverseur de type MTCMOS où l'on voit l'organisation de la cellule.

L' *Adaptative Body Biasing* ou VTCMOS (Variable Threshold CMOS) permet de contrôler la tension de seuil en appliquant une tension au substrat du transistor (cf. $V_{bs}$ eq. A.12). En mode actif, le substrat est connecté à la masse alors qu'en mode repos une tension négative

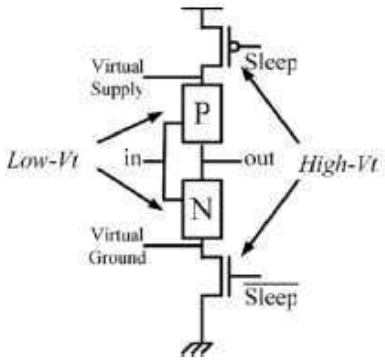



Figure A.3 – Représentation d'un inverseur MTCMOS (Multi-Vt CMOS

est appliquée, ce qui permet d'augmenter la tension de seuil et ainsi de bloquer le courant de fuite. Il est à noter que cette technique permet, de part l'application d'une tension négative sur le substrat, de contrôler la tension de seuil effective.

### A.1.2.2 Réduction de l'activité de commutation

Un facteur, non négligeable, dans la consommation est l'activité de commutation. Celle-ci affecte principalement la consommation dynamique mais aussi, dans de moindres mesures, la puissance statique. Différentes techniques, plus ou moins ad-hoc, peuvent être employées.

**A.1.2.2.1** ***Clock gating*** Dans un circuit numérique tout est cadencé au rythme de l'horloge, ce qui permet de synchroniser et séquencer les divers éléments entre eux. L'inconvénient est, que lorsque le bloc ou sous système considéré n'est pas utilisé dans la fonctionnalité courante, il consomme malgré tout de la puissance dynamique à chaque battement de l'horloge. L'idée du "clock gating" est donc de désactiver l'horloge des parties ne rentrant pas dans le chemin de donnée du calcul en cours. De nombreuses publications sur ce sujet ont été réalisées dont on peut citer [125], [40], [80], [87].

**A.1.2.2.2** ***Power gating*** L'idée du "Power Gating" aussi appelé "Power Shut-Off" (niveau système) est similaire à celle du clock gating dans le sens ou il s'agit de mettre en sommeil les blocs ou sous systèmes qui n'interviennent pas dans la fonctionnalité courante. Ainsi lors-

qu'une partie (plus ou moins importante) du circuit n'est pas utilisée, il est préférable de couper son alimentation, ce qui permet de réduire non seulement la consommation dynamique mais aussi statique. L'un des gros inconvénient de cette technique est que l'état de la partie éteinte sera perdue. Un système de sauvegarde et de restauration sera, suivant les besoins de l'application, nécessaire. De plus, un certains temps est nécessaire au réveil, ce qui peut être un facteur d'utilisation non efficace de cette technique.

**A.1.2.2.3 Autres techniques** D'autres techniques permettent aussi de réduire l'activité de commutation lors du fonctionnement du système. Par exemple dans les communications, il est possible d'encoder les informations en changeant leur format comme le "Bus-invert Coding" dans [115] et [111], les codes de Gray [117] ou encore le "T0 code" de [12]. Il est aussi possible d'effectuer une compression de codes comme dans [106], [130], qui se base sur le fait que seul une petite partie du jeu d'instruction des processeurs est utilisé dans un programme donné et ainsi permet de coder sur un minimum de bits les instructions les plus utilisées. D'autre part un choix judicieux sur la largeur des bus peut participer à une économie d'énergie non négligeable. De même, au niveau de la compilation, des optimisations peuvent être réalisées, telles que par exemple, en changeant la position de deux instructions consécutives (tout en gardant la fonctionnalité correcte) peut réduire les transitions sur le bus entre la mémoire et un processeur. Un autre point qui est source de consommation d'énergie, sont les erreurs de mémoire caches (Cache miss). En effet, lorsque le processeur tente d'accéder à une information (donnée ou instruction) en mémoire cache et que celle-ci ne s'y trouve pas, elle doit être recopiée (en générale par page) de la mémoire centrale jusqu'au cache, ce qui génère un trafic supplémentaire et donc une consommation inutile.

**A.1.2.2.4 Approches multi-tensions** Parmi les différentes techniques permettant de réduire la consommation dynamique, les approches multi-tensions ( [70]) ont étées très largement étudiées et ont fait l'objet de nombreuses publications. On trouves, dans la littérature, différentes solutions qui peuvent être distinguées de la façon suivante :

– SVS (Static Voltage Scaling) :

Différents blocs ou sous systèmes sont alimentés par des des tensions fixes différentes. Il y a alors plusieurs domaines d'alimentations.

– MVS (Multi Voltage Scaling) :
Similaire au SVS, si ce n'est que les tensions de chaque blocs ou sous-systèmes peuvent prendre plusieurs valeurs discrètes, en générale 2 ou 3.

– DVFS (Dynamic Voltage and Frequency Scaling) :
Extension du mode précèdent, avec un panel de tension plus important. De plus, la fréquence ce voit elle aussi changée, permettant un ajustement dynamique à la charge de travail. En général le contrôle est fait par une partie logiciel.

– AVS (Adaptative Voltage Scaling) :
Semblable au mode DVFS, le contrôle des tensions et fréquence est, ici, réalisé au travers d'une boucle d'asservissement. L'asservissement peut être réalisé entièrement en matériel ou intégrer une partie au niveau du logiciel.

Dans les approches multi-tensions, il faut pouvoir alimenter chaque partie ou bloc avec une tension différente et variable (suivant les cas). Pour le SVS il est juste nécessaire d'avoir plusieurs alimentations internes ou externe, ce qui ne pose pas réellement de problèmes si le nombre de tensions différentes est relativement faible. Pour ce qui est du MVS, il faut en plus un système (matériel) qui permettent de commuter, en fonctionnement, entre les différentes tensions. Les techniques DVFS et AVS, quand à elles, ont besoin que la tension d'alimentation puisse varier plus finement, voir entièrement en continu. Ces derniers utilisent, la plupart du temps un convertisseur DC-DC ainsi qu'une boucle asservie de type PID. Le contrôle de cette dernière se fait par logiciel, c'est le cas du DVFS, ou comme dans l'AVS, au travers de l'extraction en fonctionnement d'un ou plusieurs paramètres permettent d'avoir un retour d'information sur le comportement du système, tel que le taux de charge du système. Bien évidement, lorsque le contrôle est réalisé par logiciel, il est nécessaire d'intégrer une partie responsable de cette tâche

soit dans l'application ou bien au niveau de l'OS. Une autre méthode, avec DVFS, est une implémentassion matériel dédiée, réalisant la ou les politiques de gestion de la consommation.

Dans les paragraphes suivant sont donnés quelques exemples de techniques d'utilisations et de mise en oeuvre de systèmes multi-tensions de type AVS.

– Avec queue (file d'attente)

  Dans [124] est présenté une méthode DVFS en ligne, avec de multiple domaines d'horloge (MCD : Multi Clock Domaine) pilotés dynamiquement par la charge de travail. Présentant une amélioration basée sur [108] et [107], ses auteurs apportes une approche analytique, dans laquelle le processeur MCD est modélisé comme un réseau de file d'attente et le DVFS comme une boucle pilotée par l'occupation des files d'attentes. Ils propose une technique théorique de contrôle DVFS en ligne permettant de garantir la stabilité tout en réduisant la consommation d'énergie. Des mêmes auteurs, [61], propose d'étendre l'approche faite dans [124], nommée ici Local-PID, au niveau d'un CMP. Cette extension, appelée dist-PID (pour distributed PID) est une version distribuée de leur méthode analytique qui permet de piloter, non plus différentes parties d'un même processeur, mais plusieurs processeurs. C'est à dire que dans le premier cas, suivant s'il s'agit d'une opération flottante ou entière par exemple, elle ne sera pas exécutée sur la même unité et donc une file d'attente ainsi qu'un bloc DVFS différent seront utilisés. Dans le second cas, ce n'est pas directement les instructions, mais les tâches (threads) qui seront utilisées pour piloter la boucle de DVFS.

– Avec FIFO

  Dans [97] et [96] une présentation est faite d'un GPU faible consommation qui utilise une technique DVFS avec trois domaines de tensions pour les trois processeurs du circuit (RISC, Vertex Shader, Rendering Engine), organisés en série, avec un fonctionnement de type pipeline. Chacun de ces trois domaines est piloté par son propre PMU (Power Management Unit) qui permet d'ajuster la tension et la fréquence local au domaine. Afin de

pouvoir adapter les performances des processeurs, aux besoins de calcul devant êtres réalisés, il est fait usage des FIFO (First In First Out) servant de canal de communication des données entre le processeur RISC et VS (Vertex Shader) d'une part, et VS et RE (Rendering Engine) d'autre part. Les FIFO, permettent suivant leur taux de remplissage de savoir si les fréquences ainsi que les tensions, doivent êtres augmentées ou au contraire diminuées. En figure A.4 est représenté le schéma synoptique de l'architecture, sur lequel on peut voir les boucles permettant d'ajuster la consommation d'énergie suivant les besoins en performances. On peut voir en figure A.5 les chronogrammes de différents signaux du VS, dont la tension d'alimentation, la fréquence, le nombre de données entrant dans la FIFO et le nombre de pixels. On constate que lorsque le niveau de la FIFO diminue (moins de données admise), le calcul doit être accéléré en augmentant la fréquence et la tension. Il est clair que cette technique s'adapte très bien aux applications de streaming. Ce circuit, réalisé en technologie 130nm, présente un surcoût matériel de 0,45mm$^2$ par PMU.

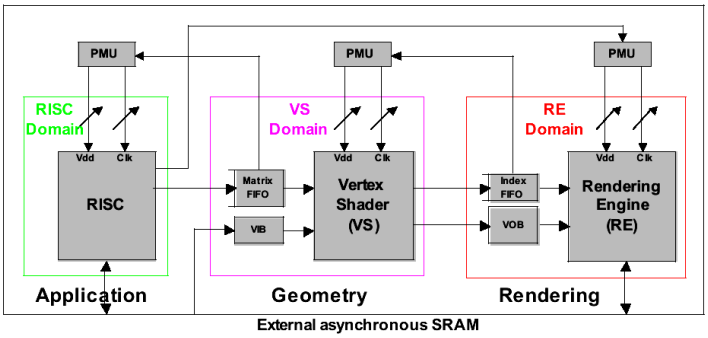



Figure A.4 – Triple domaines DVFS

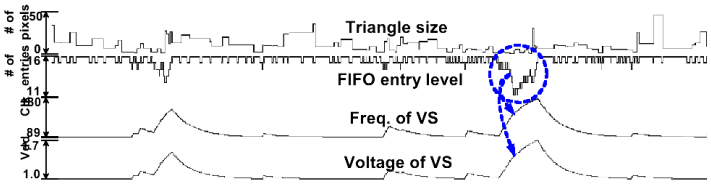



Figure A.5 – Chronogramme des signaux DVFS

– RAZOR logic

[39] présente une technique nommée RAZOR, qui utilisée conjointement avec une boucle DVS, permet de réduire la consommation d'énergie. L'idée de base est relativement simple et part du principe que la tension peut être réduite au maximum tant que les erreurs de temps générées ne sont pas trop importantes ou bien qu'il soit possible de les corriger. Le principe de fonctionnement est d'avoir deux chemins de donnée, décalés l'un de l'autre par un délai relativement court (typiquement une demie période). Le premier est le chemin "réel", utile, qui réalise le calcul. Le second est une partie destinée à détecter et à "réparer" les erreurs survenues. La présence ou non d'erreurs permet au travers d'une boucle asservie, d'augmenter ou de diminuer la tension de fonctionnement de l'étage. Principalement dédié a être utilisé dans des structures de pipeline, RAZOR se présente sous la forme de bascules (flip-flop) augmentées d'un circuit de détection et de correction d'erreur. Bien que son principe soit relativement simple, sa mise oeuvre l'est beaucoup moins. En effet, de nombreuses précautions doivent êtres prises afin d'éliminer certains problèmes comme les méta-stabilités. Son intégration dans un pipeline n'est, en revanche, nécessaire que dans les chemins critiques, ce qui permet un surcoût en surface relativement faible. [32], des mêmes auteurs, présente une intégration en 180nm, d'un processeur 64 bits dans lequel 207 bascules sur les 2388 ont été remplacées par des Razor FF, ce qui correspond à environ 9% et permet d'obtenir une réduction de la consommation de l'ordre de 44%. La figure A.6 représente la RFF, dans laquelle on peut voir le chemin principal (main flip-flop) ainsi que le chemin de détection (shadow latch). Un avantage intéressant de cette technique, est la possibilité de faire fonctionner l'étage avec un taux d'erreur non nul.

– CANARY logic

Dans [104] est présenté une variante plus simple de Razor, dans laquelle est considéré les temps typiques plutôt que les plus mauvais temps (WCET). D'après les auteurs, leur technique nommée Canary (Canary Flip-Flop), permet d'obtenir de meilleurs performances en terme de consommation que Razor, tout en étant plus simple à mettre en oeuvre.

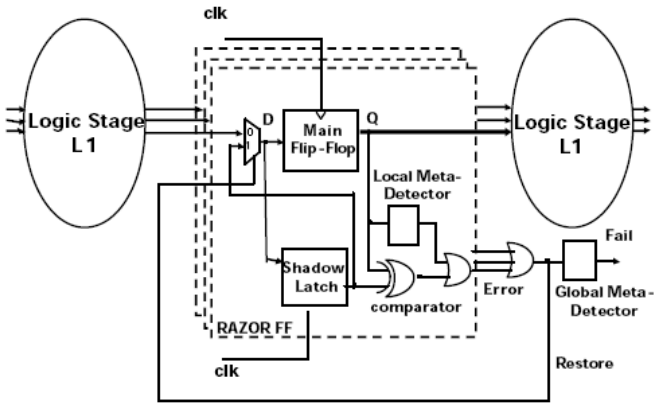



Figure A.6 – Bascule Razor (Razor Flip-Flop : RFF)

### A.1.2.3 Autre approches et approches mixtes

– UDVS (Ultra Dynamic Voltage Scaling)

[55] étudie une technique dite UDVS (Ultra Dynamic Voltage Scaling), qui lorsque la demande en performance est très faible, permet d'utiliser les transistors en mode "sub-threshold", c'est à dire avec une tension d'alimentation (Vdd) en dessous de la tension de seuil VT. Les auteurs soulignent les impacts importants du processus de fabrication surtout lorsqu'il s'agit de technologie en dessous de 100nm. La technique repose principalement en l'adaptation du facteur $\beta$, qui est le ratio des largeurs des transistors NMOS et PMOS ( $\beta$ = Wp/Wn), afin de déplacer le point de fonctionnement de la porte CMOS et de s'affranchir de certaines disparités (dûes aux variations de process) telles que sur VT et Ioff. Ceci permet de pouvoir fonctionner (en mode dégradé) en dessous de la tension de seuil. Les simulations (HSPICE), faites par les auteurs, d'un RCA (Ripple Carry Adder) 8 bits, montreraient un gain de consommation d'un rapport 42 entre le mode normal (Vdd > VT) et le mode "dégradé" (UDVS) (Vdd < VT) avec seulement 9% de surface supplémentaire (130nm). La figure A.7 représente le schéma d'un inverseur avec la logique supplémentaire nécessaire au fonctionnement du DST (Device Shadowing Technique) qui est le nom donné à celle-ci. Le mode UDVS est obtenue lorsque en=1 et enb=0.

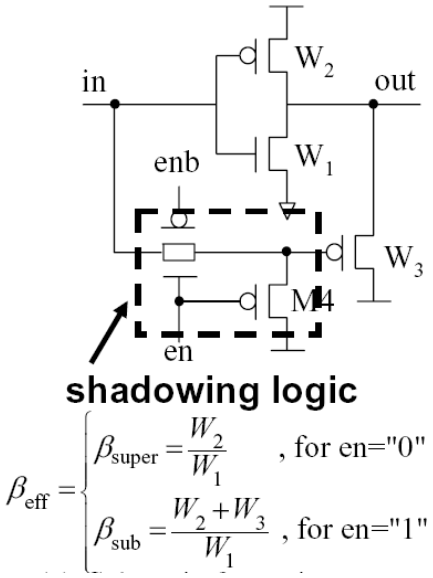



Figure A.7 – Schéma d'un inverseur avec DST

– LDV (Local Dithering Voltage) + UDVS

  [18] présente un circuit d'essai démontrant la faisabilité du concept de couplage de LVD (Local Dithering Voltage) (équivalent au Vdd-hopping) et de l'UDVS au travers d'un additionneur (32 bits Kogge-Stone) . Ils utilisent 3 tensions 1.1V, 0.6V et 0.33V qui correspondent aux niveaux optimales pour le minimum d'énergie. Ils utilisent un oscillateur en anneau permettant de générer l'horloge en adéquation avec la tension d'alimentation. Dans leur réalisation, le choix de la valeur de tension UDVS se fait par rapport aux mesures des paramètres technologiques (90nm).

– Power Gating avec plusieurs sleep modes

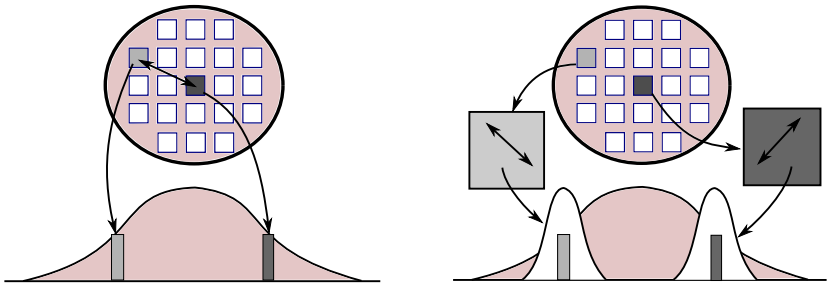
  [3] présente une technique de power gating avec une particularité supplémentaire, qui est de proposer plusieurs modes de mise en veille. Quatre modes sont disponibles : Active (éveiller), Sleep (endormir ), Dream (rêver), Snore (ronfler). Le mode Active est le mode de fonctionnement normal. Sleep, Dream et Snore sont les modes faibles consommations avec des temps de réveils, respectivement, de plus en plus long mais offrant des consommations plus faibles. Ainsi le mode Snore consomme le moin d'énergie mais nécessite

le temps de réveil le plus important. Un particularité concernant le mode Sleep, est qu'il permet de maintenir l'état courant du système concerné.

## A.2 La Variabilité du processus de fabrication et ses Impacts

Les variations de paramètre des transistors CMOS (PVT [14] :Process Voltage Temperature) sont, en grande partie, induit par le procédé de fabrication lui-même, dont la precision est de plus en plus difficile à maîtriser. Ceci est particulièrement vraie pour les noeuds technologiques sub-micronique avancés tel que 32 nm et en deça.

Il est en général, considéré, trois niveaux de variations. Le première interviennant entre les *Wafer* (*wafer-to-wafer*, W2W). Un Wafer est un disque de silicium pouvant aller jusqu'à 450mm de diametre, sur lequel est intégré plusieurs milliers de puce ou die. Le second niveau, se produit à l'intérieur d'un même *wafer*, entre les différentes puces et est nommé variation *die-to-die* (D2D). Enfin, les variations *within-die* (WID), impactent directement la puce elle-même, en créant des disparités sur les paramètres à l'intérieur du die.



(a) Variation inter-die (b) Variation intra-die

Figure A.8 – Illustrations de la variabilité inter et intra-die.

D'autre part, on constate que les effets des variations peuvent être classifiés selon qu'ils soient systematiques ou aléatoires. Les variations systématiques ( [42], [116]), comme son nom l'indique, se produisent à chaque fois de la même façon et sont dûes, par exemple, aux abbérations des lentilles utilisées pour la lithographie. Les variations aléatoires [13] quand à elles, sont principalment dûes aux fluctuations de densité des dopants ou par des effets de bords.

Les paramètres clés, influencés par ces variations, sont principalement la tension de seuil des transistors ($V_{th}$), et la longueur effective du canal ($L_{eff}$). $V_{th}$ est spécialement importante, car ses variations ont un impact substantiel sur deux propriétés majeur d'un processeur, à savoir,

la fréquence maximale qu'il pourra atteindre et, via les courants de fuites, la puissance statique qu'il dissipera. De plus, $V_{th}$ est aussi très lié à la température qui accroît sa variabilité [119].

### A.2.1 Variations PVT

La figure A.9 montre la distribution de la fréquence et du courant de fuite au repos ($I_{sb}$) de microprocesseurs dans un *wafer*. Comme nous pouvons le constater, l'étalement de la distribution, en fréquence et en courant de fuite, est relativement important.Ce phénomène est dû aux variations des paramètres de transistor. Avec un étalement maximal en fréquence de 30%, et un courant de fuite dont le maximum est 20 fois supérieur au minimum. Il est à noter, que les puces ayant les plus grandes fréquences ont aussi un plus large étalement de courants de fuites. De plus, nous constatons que pour les faibles courants de fuites, l'étalement en fréquence est plus important.

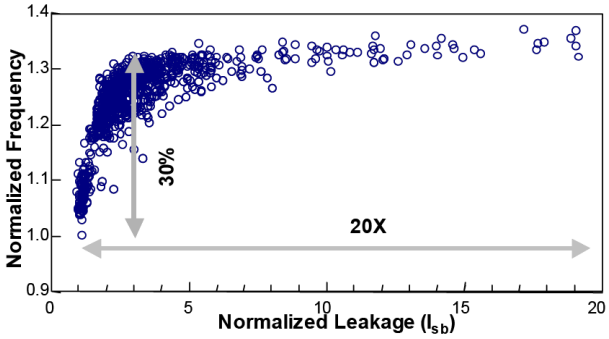



Figure A.9 – Variations de fréquences et de courants de fuites ([14])

L'étalement du courant au repos est dû aux variations de longueurs du canal ainsi qu'à celles de la tension de seuil. La figure A.10 illustre la distribution D2D de $V_{th}$ et le courant $I_{sb}$ résultant. Pour une technologie CMOS de 180nm, comme c'est le cas ici, $V_{th}$ présente une distribution normale avec un $3.\sigma$ (3x écart-type) qui vaut environ 30mV, ce qui cause des variations significatives de performances et de fuites dans le circuit. De plus, le chemin critique peut être différent d'un chip à l'autre. La figure A.10 montre aussi le detail de la distribution de courants de fuites.

Les différences d'activité de commutation ainsi que des divers types de logiques sur le die,

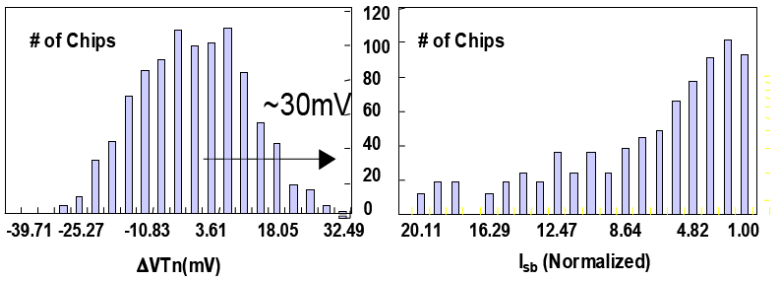



Figure A.10 – Variations D2D de $V_{th}$ et de $I_{sb}$

ont pour résultat des différences de consommations à travers le circuit et créeent des points chauds (*Hot-Spot*) causant variations supplémentaire des paramètres du transistor. La figure A.11représente une image thermique d'un microprocesseur avec un point chaud à 120 °C, alors que sa mémoire cache est à 70 °C. Les fluctuations de température posent de nombreux problèmes en terme de performance et de mise en boitier. De plus, la tension d'alimentation n'a cessée de diminuer avec chaque nouvelle génération technologique, ce qui rend difficile d'y ajuster $V_{th}$ ainsi que d'atteindre les perfomnances espérées du transistor.

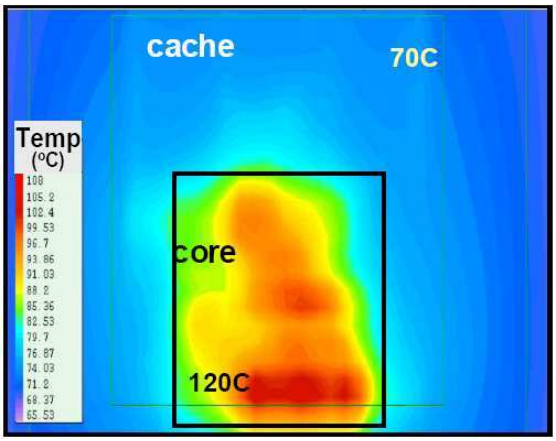



Figure A.11 – Variation de température WID

L'une des conséquences des variations PVT se manifeste directement sur la variation de fréquence du circuit. La figure A.12 représente la distribution en fréquences entre les dies, pour une technologie de 180nm.

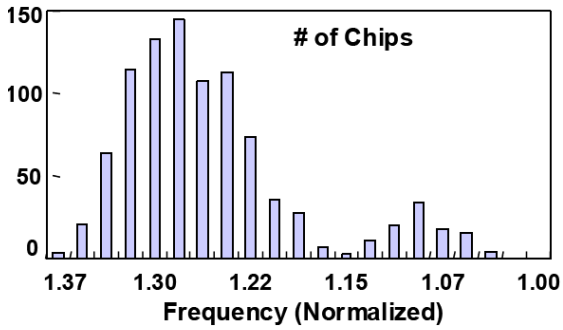



Figure A.12 – Variations D2D de fréquences

### A.2.2 Modèle de variation de process

Les variations de process peuvent être D2D et WID. Ce dernier peut lui même être décomposé en deux, avec un effet systématique et un effet aléatoire. Par définition, l'effet systématique présente une corrélation spatiale et donc les transistors voisins partagent des valeurs de paramètres similaires. Au contraire, les effets aléatoires ne présentent pas de corélations spatiales, et par conséquent les paramèters de deux transistors voisins different de façon aléatoire. Plus généralement, les variations de n'inporte quel paramètre P peuvent être représentées par : $\Delta P = \Delta P_{D2D} + \Delta P_{WID} = \Delta P_{D2D} + \Delta P_{rand} + \Delta P_{sys}$.

Dans [103] est présenté VARIUS, un modèle de variation de process. Il y est présenté un modèle principalement déstiné aux variations WID. Les effets systématiques et aléatoires sont traités séparement et sont modélisés comme étant des distributions normales avec des effets additifs tel que dans [113]. Il est à noter que les varaitions D2D et WID ne sont en réalité pas statistiquement indépendants. Dans les paragraphes qui suivent, le modèle VARIUS est expliqué plus en détails.

#### A.2.2.1 Variations systématiques

La modélisation des effets systématique utilise une distribution normale multi-variables [98] avec une structure sphérique de corélation spatiale [30]. Le chip est divisé en n petites sections rectangulaires de tailles égales, chacunes d'entres elles ayant une seule valeur de composante systématique pour $V_{th}$ (et pour $L_{eff}$) avec une distribution normale de valeur moyenne $\mu = 0$

et d'écart type $\sigma_{sys}$. Ce qui est une approche générale qui a deja été utilisée dans [113]. Il est aussi supposé que la corrélation spatiale est homogène (indépendant de la position) et isotrope (indépendant de la direction). Ce qui signifie que la corrélation de la composante systématique de deux points $\overrightarrow{x}$ et $\overrightarrow{y}$ sur le chip ne depend que de la distance entre ces derniers [126]. Ainsi, la corrélation de la composante systématique d'un paramètre P est : $corr(P_{\overrightarrow{x}}, P_{\overrightarrow{y}}) = \rho(r), r = |\overrightarrow{x} - \overrightarrow{y}|$. Par définition $\rho(0) = 1$, c'est à dire totalement corrélé, et $\rho(\infty) = 0$ totalement non-corrélé. Pour spécifier le comportement de $\rho(r)$ aux limites, il est utilisé le modèle sphérique de [30] car il présente un bon accord avec les mesures de [42], bien que la fonction de corrélation utilisée dans ce dernier ne soit pas isotropique. Mais cependant, la forme que ce soit horizontalement ou verticalement, est la même et dans les deux cas elle se superpose parfaitement au modèle sphérique. La fonction spérique utilisée est définit par :

$$\rho(r) = \begin{cases} 1 - \frac{3r}{2\phi} + \frac{r^3}{2\phi^3}, & (r \leq \phi) \\ 0, & sinon \end{cases} \tag{A.15}$$

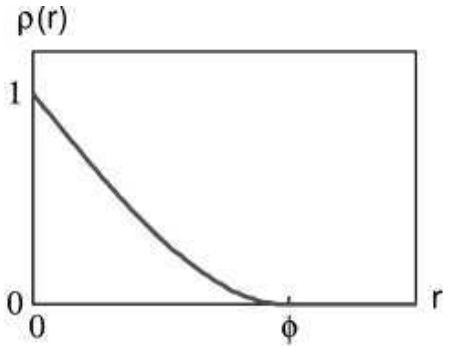



Figure A.13 – Corrélation systématique de paramètres de deux points est fonction de la distance $r$ qui les sépare

La figure A.13 représente le tracé de $\rho(r)$, où l'on peut voir la corrélation décroître approximativement linéairement, puis plus lentement pour atteindre 0 à la distance $\phi$, appelée *range*, signifiant que plus aucune corrélation n'existe entre les variations WID de deux transistors. $\phi$ étant exprimé comme une fraction de la longueur du chip, une valeur importante de celui-ci implique que de larges sections de chip sont corrélées avec d'autre(l'opposé est vrai).

La figure A.14 illustre les cartes de deux exemples de variations systématiques (générée avec geoR [100]) pour $V_{th}$ avec $\phi = 0.1$ et $\phi = 0.5$. Dans le cas où $\phi = 0.5$, on peut constater de larges plages signifiant une corrélation sur de grandes sections. Il est à noter, que lorsqu'il n'y a pas de corrélation ($\phi = 0$), la carte apparait comme un bruit blanc.

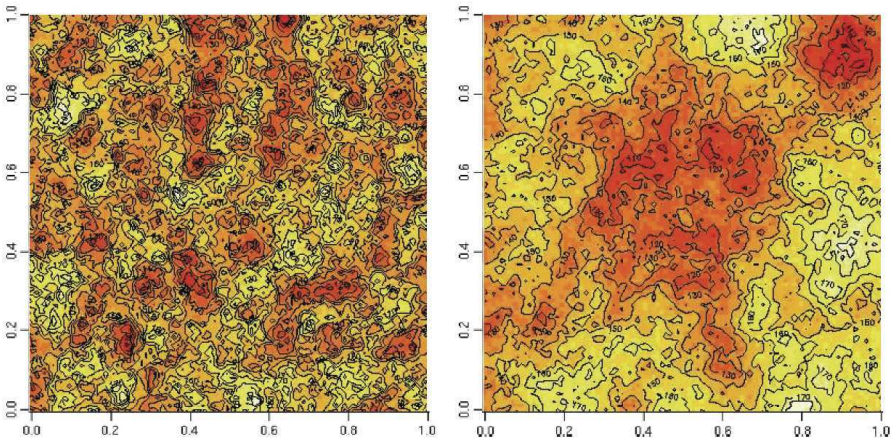


Figure A.14 – Carte des variations systématique de $V_{th}$ sur un chip avec $\phi = 0.1$(gauche) et $\phi = 0.5$ (droite)

Les parmètres de process concernés sont ici $L_{eff}$ et $V_{th}$, pour lesquels le rapport ITRS [57] a projetté que le total $\frac{\sigma}{\mu}$ de $L_{eff}$ devrait, grossièrement, être la moitié de celui de $V_{th}$. Ainsi, il est fait l'approximation que $\frac{\sigma_{sys}}{\mu}$ de $L_{eff}$ est égale à un demi de $\frac{\sigma_{sys}}{\mu}$ de $V_{th}$. De plus, les variations systématique de $L_{eff}$ causent des variations systématique de $V_{th}$, le reste des variations étant, pour la plupart, dûes aux effets aléatoires du dopage. En conséquence, l'équation suivante :

$$L_{eff} = L^0_{eff} = \Big(1 + \frac{V_{th} - V^0_{th}}{2V^0_{th}}\Big), \tag{A.16}$$

est utilisé pour générer une valeur pour la composante systématique de $L_{eff}$ dans une section de chip, connaissant celle de $V_{th}$ pour la même section. $L^0_{eff}$ et $V^0_{th}$ étant réspectivement les valeurs nominale de $L_{eff}$ et $V_{th}$.

#### A.2.2.2 Variations aléatoires

Les variations aléatoires, contrairement aux variations systématiques, se produisent à un grain relativement fin, au niveau transistor, il est donc plus difficile de l'obtenir par simulation d'une grille où chaque section a sa propre valeur de paramètre. Il est fait supposition que les composantes aléatoires de $L_{eff}$ et $V_{th}$ sont tout deux des distributions normale de valeur moyenne $\mu = 0$, d'écart type $\sigma_{rand}$ et, que pour un transistor donné, il n'existe pas de corrélations entre les valeurs pour $V_{th}$ et celles de $L_{eff}$.

#### A.2.2.3 Valeurs de $\sigma$ et de $\phi$

Puisque les composantes systématiques et aléatoire sont toutes deux normalement distribuées et indépendantes, le total de variations WID est aussi une distribution normale ayant pour écart type :

$$\sigma_{total} = \sqrt{\sigma_{rand}^2 + \sigma_{sys}^2} \tag{A.17}$$

Etant donné les projections trop optimistes [62] de [57] qui donnait un $\frac{\sigma_{total}}{\mu} = 0.06$ pour 2005 à $V_{th}$, il est utilisé une valeur de $\frac{\sigma_{total}}{\mu} = 0.09$. Celle-ci s'accorde aux donnés empiriques de [64], à savoir que les composantes systématiques et aléatoires sont approximativement les mêmes en technologie 32nm. Ainsi, pusique $\sigma_{sys} \simeq \sigma_{rand}$ alors avec A.17 on a $\sigma_{sys} = \mu\frac{0.09}{\sqrt{2}} = 6.3\%$ de $\mu$ (cette valeur s'accorde avec les données empiriques de [73]). Pour $L_{eff}$, puisque $\frac{\sigma_{total}}{\mu}\Big|_{L_{eff}} = 0.5\frac{\sigma_{total}}{\mu}\Big|_{V_{th}}$ alors $\sigma_{sys} = 3.2\%$ de $\mu$.

Pour éstimer $\phi$, les mesures expérimentales de [42], qui montrent que le paramètre de la longueur de grille à une étendue proche de la moitié de la longueur du chip, sont utilisées et donc $\phi = 0.5$

### A.2.3 Impacte sur la consommation statique et la fréquence

#### A.2.3.1 Equations de transistor

Les équations du courant de drain d'un transistor, utilisant le modèle traditionnel Shockley, sont les suivantes :

$$I_d = \begin{cases} 0, & si\ V_{gs} \leq V_{th} \\ \beta(V_{gs} - V_{th} - \frac{V_{ds}}{2})V_{ds}, & si\ V_{ds} < V_{gs} - V_{th} \\ \beta\frac{(V_{gs}-V_{th})^2}{2}, & si\ V_{ds} \geq V_{gs} - V_{th} \end{cases} \tag{A.18}$$

Avec $\beta = \frac{\mu C_{ox} W}{L_{eff}}$, où $\mu$ est la mobilité et $C_{ox}$ la capacité d'oxyde. Dans les technologies submicroniques profondes, ces relations sont suplentées par la loi *Alpha-Power* de [102]. Ce dernier est une extension du modèle de Shockley et a été introduit pour rendre compte de l'effet de saturation de la vitesse des porteurs dans les MOSFET à canal court.

$$I_d = \begin{cases} 0, & si\ V_{gs} \leq V_{th} \\ \frac{W}{L_{eff}}\frac{P_c}{P_v}(V_{gs} - V_{th})^{\frac{\alpha}{2}}V_{ds}, & si\ V_{ds} < V_{d0} \\ \frac{W}{L_{eff}}P_c(V_{gs} - V_{th})^{\alpha}, & si\ V_{ds} \geq V_{d0} \end{cases} \tag{A.19}$$

avec $\alpha$, l'index de saturation de vitesse, $P_v$ et $P_c$ étant des paramètres (constants ici), et :

$$V_{d0} = P_v(V_{gs} - V_{th})^{\frac{\alpha}{2}} \tag{A.20}$$

Ainsi, le temps requit pour commuter une sortie logique découle de A.19 et s'exprime :

$$T_g \propto \frac{L_{eff}V}{\mu(V - V_{th})^{\alpha}} \tag{A.21}$$

où $\alpha$ est typiquement de 1.3 et $\mu$ est fonction de la température tel que $\mu(T) \propto T^{-1.5}$. On constate que lorsque $V_{th}$ décroît, $V - V_{th}$ augmente et la porte devient plus rapide. De même, lorsque $T$ augmente, $V_{th}$ diminue et $V - V_{th}(T)$ augmente. Cependant, $\mu(T)$ diminue [63]. Le second facteur domine et avec une valeur élevée de $T$, la porte devient plus lente. Il est à

remarquer que le modèle Shockley est un cas particulier du modèle Alpha-Power avec $\alpha = 2$.

Le courant de fuite sous le seuil est le principal courant de fuite dans un transistor. Le modèle de courant de fuite sous le seuil d'un transistor, basé sur [132] est le suivant :

$$I_{leak} \propto (\frac{kT}{q})^2 e^{\frac{q(V_{off}-V_{th})}{\eta kT}} \tag{A.22}$$

où $k$, $q$ sont des constantes et $\eta$ et $V_{off}$ sont des paramètres déterminés empiriquement. De cette équation, on peut constater que les transistors avec un $V_{th}$ faible, ont un courant $I_{leak}$ élevé. De plus, à mesure que $T$ augmente, le transistor voit son courant de fuite rapidement augmenter car, d'une part $I_{leak}$ dépend de $T$ et d'autre part $V_{th}$ diminue quand $T$ augmente.

#### A.2.3.2 Impacte sur la fréquence du chip

A travers l'Eq. A.21, les variations de process de $V_{th}$ et de $L_{eff}$ introduisent des variations de délais dans les portes et donc dans les chemins critiques, ce qui, malheureusement, affectent la vitesse maximale des processeurs, laquelle étant basée sur ces derniers. L'équation A.21 décrit approximativement le délai d'un inverseur. En substituant A.16 à A.21 et en factorisant les constantes, on obtient :

$$T_g \propto \frac{V\left(1+\frac{V_{th}}{V_{th}^0}\right)}{(V-V_{th})^{\alpha}} \tag{A.23}$$

De façon empirique, A.23 est presque linéaire à l'égard de $V_{th}$ pour la zone du paramètre qui nous intéresse. Ainsi, puisque $V_{th}$ est normale et qu'une fonction linéaire d'une variable normale est elle même normale, on peut dire que $T_g$ est approximativement normale.

Supposant que tous les chemins critiques d'un processeur sont constitués de $n_{cn}$ portes et qu'un processeur moderne comporte de l'ordre de 1000 chemins cristiques, [16] calcul la probabilité de distribution du plus long chemin critique ( max $T_{cp}$) dans le chip. Alors, la fréquence du processeur peut être éstimée comme l'inverse du plus grand chemin critique ( 1 / max $T_{cp}$).

La figure A.15 montre la probabilité de distribution de fréquence du chip pour différentes valeurs de $\sigma_{total}/\mu$ de $V_{th}$. La fréquence est donnée relativement à un processeur sans variation de $V_{th}$ (F/ $F_0$). La figure montre que à mesure que $\sigma_{total}/\mu$ augmente, la fréquence moyenne

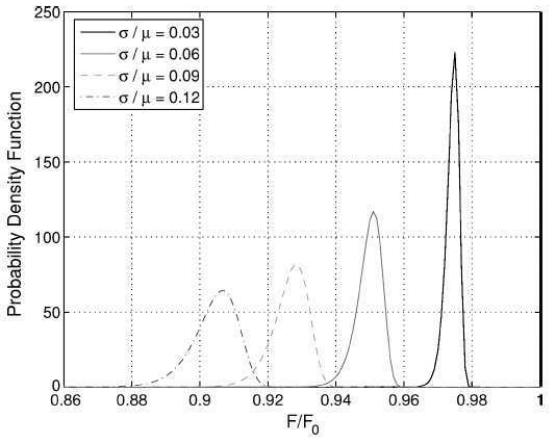



Figure A.15 – Distribution de probabilité de la fréquence du chip en fonction du $\frac{\sigma_{total}}{\mu}$ de $V_{th}$. ($V_{th}^0 = 0.150V$ à 100°C, 12FO4 dans le chemin critique et 10000 chemins critiques dans le chip([103])

sur le chip diminue et sa distribution s'étale de plus en plus. En d'autres termes, partant d'un lot de chips, à mesure que le $\sigma_{total}/\mu$ de $V_{th}$ augmente, la fréquence moyenne du lot diminue et chaque chip voit sa fréquence s'écarter de la moyenne.

Néamoins, ce type de perte de fréquence peut être réduit si le processeur possède un système permettant de tolérer les erreurs de timming dûes à ces variations.

#### A.2.3.3 Impacte sur la puissance statique

Pour estimer la consommation statique, il est utilisé A.22 appliqué à tous les transitors de chip, tout en utilisant la distribution de $V_{th}$. Soit $P_{leak}$, $I_{leak}$ respectivement la consommation et le courant statiques soumis à des variations et $P_{leak}^0$, $I_{leak}^0$, les mêmes paramètres lorsqu'il n'y a pas de variations. Le ratio de post-variation et de pré-variation des fuites est :

$$\frac{P_{leak}}{P_{leak}^0} = \frac{I_{leak}}{I_{leak}^0} = e^{(\frac{q\sigma}{\eta kT})^2/2} \tag{A.24}$$

qui montre que l'augmentation de la consommation et du courant statique dépendent de $\sigma$. La figure A.16 montre la consommation statique relative en fonction de $\sigma$ pour T=25°C, moyenne pour $V_{th}$ : $\mu = 150mV$ et $\phi = 0.5$. Les valeurs typiques de $\sigma$ pour $V_{th}$ sont 0.09-0.12x$\mu$. Nous

pouvons voir sur cette figure que les fuites augmentent très rapidement avec $\sigma$

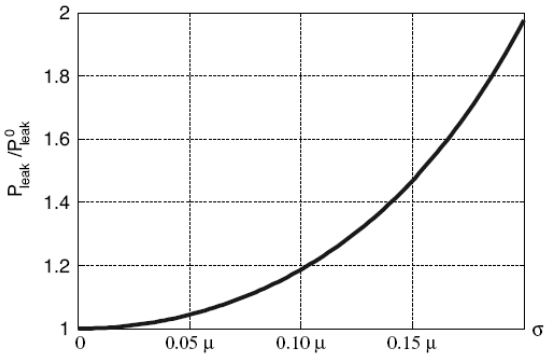



Figure A.16 – Puissance statique relative dans le chip en fonction de $\sigma$ de $V_{th}$ ([120])

En conclusion, les variations de $V_{th}$ sont très dommageables pour la fréquence et la consommation statique du chip. Plus $\sigma$ de $V_{th}$ augmente, plus les courants de s augmentent rapidement et plus la fréquence moyenne du chip diminue, tout en variant plus.

Annexe B

# Les Graphes

## Sommaire

## B.1 Graphes Acycliques Dirigés (DAG)

### B.1.1 Définitions

Un Graphe Acyclique Dirigé ou DAG (*Directed Acyclic Graph*) est défini comme le n-uplet $G = (V, E, w, c)$, où $V = \{v_i,\ i = 1..v\}$ et $E = \{e_{i,j},\ i \neq j,\ i \in [1, v],\ j \in [1, v]\}$ sont, respectivement, l'ensemble des sommets et l'ensemble des arcs dirigés reliant les sommets. Le nombre de sommets dans $V$ est noté $|V|$. Le nombre d'arcs dans $E$ est $|E|$. A chaque sommet et à chaque arc, est associé un poid ou coût, respectivement $w(v_i)$ et $c(e_{i,j})$ (qui peut être simplifié par $w_i$ et $c_{i,j}$).

**Définition 7** (Chemin)**.** *Dans un DAG $G = (V, E, w, c)$, un chemin $p$ d'un sommet $v_1$ à un sommet $v_k$ est une séquence de sommets $v_1, v_2, ..., v_k$ tel que les sommets soient connectés par des arcs $e_{i,i+1} \in E,\ i = 1, 2, ...k - 1$. Ce chemin est noté par $p = p(v_1, v_K) = v_1, v_2, ...v_k$. Un sommet $v_i$ appartenant au chemin $p$ est noté $v_i \in p$. Un arc $e_{i,j}$ appartenant au chemin $p$ est noté $e_{i,j} \in p$.*

**Définition 8** (Longueur d'un chemin)**.** *La longueur d'un chemin $p$, est la somme des poids de tous les sommets et de tous les arcs appartenant à celui-ci.*

$$len(p) = \sum_{v_i \in p, v_i \in V} w(v_i) + \sum_{e_{i,j} \in p, e_{i,j} \in E} c(e_{i,j})$$

**Définition 9** (Ordre topologique)**.** *L'ordre topologique d'un DAG $G = (V, E, w, c)$, est une séquence linéaire de tous les sommets du graphe, tel que si il existe un arc $e_{ij} \in E$, alors $v_i \in V$ doit apparaître avant le sommet $v_j \in V$. Un DAG peut avoir plusieurs ordres topologiques. L'algorithme de recherche en profondeur (Alg. B.2)([27]) peut être modifié pour ranger en ordre topologique les sommets dans une liste. La complexité temporelle est alors $O(V + E)$.*

**Définition 10** (Prédécesseurs)**.** *L'ensemble des sommets $v_x \in V$, tel que $\exists e_{x,i} \in E$, est appélé ensemble des prédécesseurs immédiats de $v_i$. Celui-ci est noté $\Gamma^-(v_i)$. Un sommet $v_i$ avec $\Gamma^-(v_i) = \varnothing$ est appelé sommet parent ou source. Le cardinal $|\Gamma^-(v_i)|$ est appelé degré d'entrée et est noté $inDeg(v_i)$.*

**Définition 11** (Successeurs)**.** *L'ensemble des sommets $v_x \in V$, tel que $\exists e_{i,x} \in E$, est appelé ensemble des successeurs immédiats de $v_i$. Celui-ci est noté $\Gamma^+(v_i)$. Un sommet $v_i$ avec $\Gamma^+(v_i) = \varnothing$ est appelé sommet enfant ou puit. Le cardinal $|\Gamma^+(v_i)|$ est appelé degré de sortie et est noté $outDeg(v_i)$.*

**Définition 12** (Chemin critique (CP))**.** *Le chemin critique (Critical Path) $cp$ d'un DAG $G$ est le plus grand chemin de $G$.*

$$len(cp) = max_{p \in G}\{len(p)\}$$

**Définition 13** (*Top Level*)**.** *Le Top Level ou tlevel d'un sommet $v_i$ est le poid du chemin le plus long du sommet source à $v_i$ (excluant $v_i$).*

$$tl(v_i) = \begin{cases} 0 & \text{, si } v_i \in source(G) \\ \max_{v_k \in \Gamma^-(v_i)}\{tl(v_k) + w(v_k) + c(e_{k,i})\} & \text{, sinon} \end{cases}$$

**Définition 14** (*Bottom Level*)**.** *Le Bottom Level ou blevel d'un sommet $v_i$ est le poid du chemin le plus long de $v_i$ au sommet puit (incluant $v_i$).*

$$bl(v_i) = \begin{cases} w_(v_i) & \text{, si } v_i \in puit(G) \\ \max_{v_k \in \Gamma^+(v_i)}\{bl(v_k) + c(e_{i,k}) + w(v_i)\} & \text{, sinon} \end{cases}$$

Les Algorithmes B.3 et B.4 implémentent les procédures permettant de déterminer, respectivement *tlevel* et *blevel*. Tous deux ont une complexité de calcul de $O(V + E)$.

**Relation entre *level* et chemin critique :** Le *Bottom Level* est la longueur du chemin le plus long à partir d'un sommet donné. Donc, le plus grand *Bottom Level* est la longueur du chemin le plus long du DAG. Ce qui correspond au chemin critique. De plus, la somme des *Top Level*s et des *Bottom Level*s d'un sommet donné, permet de déterminer la longueur du chemin le plus long "passant" par ce sommet.

### B.1.2 Algorithmes associés

L'Algorithme B.1 implémente le tri topologique définit en section précédente. Il utilise l'Algorithme B.2 de parcours en profondeur (DFS).

#### B.1.2.1 Tri topologique

**Algorithme B.1** Tri topologique

**Entrée:** DAG $G = (V, E, w, c)$
**Sortie:** Liste des sommets en ordre topologique

```
procedure TOPOLOGIC_SORT(G)
    TList ← {∅}
    for all sommet v_i ∈ V do
        is_dicovered(v_i) ← False
    end for
    for all sommet v_i ∈ V do
        if is_dicovered(v_i) = False then
            DFS_VIST(v_i, TList)
        end if
    end for
end procedure
```

#### B.1.2.2 Parcours en profondeur

**Algorithme B.2** Parcours en profondeur (DFS : *Deep First Search*)

**Entrée:** DAG $G = (V, E, w, c)$
**Sortie:** Liste $TList$ modifiée

```
procedure DFS_VISIT(v_i, TList)
    is_dicovered(v_i) ← True
    for all sommet v_j ∈ successeurs(v_i) do
        if is_dicovered(v_i) = False then
            DFS_VIST(v_i, TList)
        end if
    end for
end procedure
```

Les Algorithmes B.3 et B.4 calculent respectivement, les *Top Level*s et les *Bottom Level*s en visitant les sommets dans l'ordre topologique et l'ordre topologique inverse. Pour ces deux algorithmes, le nombre total de répétitions de boucle *for* est le nombre de sommets plus le nombre de prédécesseurs/successeurs, c.-à-d. $O(V + E)$. Le tri en ordre topologique avec l'algorithme DFS ayant une complexité $O(V + E)$, la complexité totale pour calculer les *Top Level*s ou les *Bottom Level*s est $O(V + E)$.

#### B.1.2.3 *Top level*

**Algorithme B.3** Calcul des *top_level*

**Entrée:** DAG $G = (V, E, w, c)$
**Sortie:** Liste des *top_level* pour chaque sommet de $G$

```
procedure TLEVEL(G)
    TList ← TOPOLOGIC_SORT(G)
    for all sommet n de TList do
        max ← 0
        for all predecesseurs p de n do
            if tlevel(p) + w_p + c_{p,n} < max then
                max ← tlevel(p) + w_p + c_{p,n}
            end if
        end for
        tlevel(n) ← max
    end for
end procedure
```

#### B.1.2.4 *Bottom level*

**Algorithme B.4** Calcul des *bottom_level*

**Entrée:** DAG $G = (V, E, w, c)$
**Sortie:** Liste des *bottom_level* pour chaque sommet de $G$

```
procedure BLEVEL(G)
    RTList ← REVERSED_TOPOLOGIC_SORT(G)
    for all sommet n de RTList do
        max ← 0
        for all successeurs s de n do
            if blevel(s) + c_{n,s} > max then
                max ← blevel(p) + c_{n,s}
            end if
        end for
        blevel(n) ← w_n + max
    end for
end procedure
```

## B.2 Partitionnement de graphe multi-niveaux

Le problème de partionnement de graphe en k-partitions (*k-way*) est définit de la façon suivante : Etant donné un graphe $G = (V, E)$, avec $|V| = n$, partitionné $V$ en $k$ sous-ensembles, $V_1, V_2, ..., V_k$ tel que $V_i \cap V_j = \varnothing \; \forall i \neq j$, $|V_i| = \frac{n}{k}$, $\cup_i V_i = V$ et que le nombre d'arcs de $E$, dont les sommets incidents appartiennent à des sous-ensembles différents, soit minimisé.

Un partitionnement *k-way* de $V$ est communément représenté par un vecteur de partitionnement $P$, de longeur $n$, tel que pour chaque vertex $v \in V$, $P[v]$ est un entier entre 1 et $k$, indiquant la partition auquelle le sommet $v$ appartient. Etant donné un partitionnement $P$, le nombre d'arcs dont les sommets incidents appartiennent à des partitions différentes est appelé *edge-cut* ("front de coupe") du partitionnement.

La structure de base d'un algorithme de partitionnement de graphe en k-partition est relativement simple. Elle se décompose en trois phases : Contraction(Coarsening), Partitionnement et Expansion (Uncoarsening) ([67, 68, 66, 65]), détaillées dans la suite.

### B.2.1 Phase de Contraction (Coarsening Phase)

Durant la phase de contraction, une séquence de plus petits graphes $G_i$ = ($V_i$ , $E_i$ ), est construite à partir du graphe original $G_0$ = ($V_0$, $E_0$ ) tel que $|V_i| > |V_{i+1}|$. Le graphe $G_{i+1}$ est construit à partir de $G_i$ en trouvant un assortiment maximal $M_i \subseteq E_i$ de $G_i$ et fusionnant ensemble les sommets qui sont incidents à chaque arc de l'assortiment. Dans ce processus pas plus de deux sommets sont fusionnés ensemble, car un assortiment du graphe est un jeux d'arcs, dont aucun des deux n'est incident au même sommet. Les sommets qui ne sont incidents à aucun arc de l'assortiment, sont simplement copiés vers $G_{i+1}$.

Lorsque des sommets $v, u \in V_i$ sont fusionnés pour former le sommet $w \in V_{i+1}$, le poid du sommet $w$ est fixé égal à la somme des poids des sommets $v$ et $u$, alors les arcs incidents sur $w$ sont fixés comme étant l'union des arcs incidents sur $v$ et $u$, moins l'arc $(v, u)$.

Si il existe un arc, à la fois, incident à $v$ et à $u$, alors le poid de cet arc est fixé égal à la somme des poids de ces arcs. De plus, durant les niveaux successifs de contraction, à la fois le poid des sommets et des arcs augmentent.

Un appariement maximal peut être caculé de différentes façons [66]. La méthode utilisée pour calculer cet appariement, impacte grandement la qualité de la bisection, ainsi que le temps nécessaire à la phase de contraction.

Deux de ces schémas d'appariement peuvent être utilisés : *Random Matching* (RM) ou *Heavy-Edge Matching* (HEM).

Le schéma RM, calcul l'appariement maximal en utilisant un algorithme aléatoire [67, 68], c-à-d. visite les sommets du graphe dans un ordre aléatoire et sélection aléatoire d'un sommet adjacent parmi ceux non-marqués pour la contraction.

Le schéma HEM calcul un appariement $M_i$, tel que le poid des arcs de $M_i$ soit grand.

L'appariement est calculé en utilisant un algorithme aléatoire de la même façon que RM. Cependant, au lieu de selectionner aléatoirement un sommet pour la contraction, HEM, selectionne le sommet non-marqué qui lui est lié par l'arc de poid maximal.

Il est à remarquer que le schéma HEM réduit la somme des poids des arcs dans le graphe contracté de façon plus importante que RM. Le schema HEM produit de meilleurs résultats que RM avec, de plus, une durée d'affinage moins importante ([66]).

### B.2.2 Phase de Partitionnement du graphe contracté (Initial Partitioning Phase) :

La seconde phase, consiste à calculer une bisection du graphe contracté $G_k = (V_k, E_k)$, qui soit équilibrée. Une évaluation de différents algorithmes afin de partitionner le graphe contracté peut être trouvée dans [66].

### B.2.3 Phase d'Affinage ou d'expansion (Uncoarsening Phase) :

La partition du graphe contracté $G_k$ est projetée progressivmenet sur le graphe original en affinant successivement les partitions intermédiaires grâce à l'utilisation d'un heuristique locale. En effet, même si une partition $G_i$ est à un minima local, la partition $G_{i-1}$ obtenue par projection peut ne pas être à un minima local.

Annexe C

# Algorithmes génétiques

**Sommaire**

## C.1 Introduction

Les algorithmes génétiques ou *GA* ([46]) font parties d'une catégorie de méthode de recherche et d'optimisation imitant les principes de l'évolution des espèces et de la sélection naturelle. Les algorithmes génétiques travaillent sur une population d'individus, chacune représentant une solution au problème d'optimisation. Les GA vont évoluer cette population afin d'en améliorer les individus à chaque nouvelle génération. En général, un individu (solution) est représenté par une chaîne binaire appelé *chromosome* ou *génome*, ceci correspond au *codage*. A l'initialisation, une population d'individu est générée, soit aléatoirement, soit à l'aide d'un heuristique, constituant ainsi la *population initiale*. Chaque chromosome de la population est alors évalué au travers de la fonction objectif du problème à résoudre et un facteur de qualité ou *fitness* lui est associé. La fonction fitness normalise les valeurs de la fonction objectif en les transformants en une mesure relative comprise entre 0 et 1. Par exemple, la fonction fitness $F(x) = g(f(x))$ avec la fonction $g$ transformant les valeurs de la fonction objectif $f(x)$ en nombre non-negatifs. Vient alors la phase de reproduction. Celle-ci se base sur la fonction fitness pour sélectionner, parmi la population, les "meilleurs" chromosomes qui se reproduiront. La reproduction est réalisée au moyen de l'opérateur de croisement ou *crossover* qui va échanger une ou plusieurs portions de deux chromosomes parents afin de créer un nouvel individu, potentiellement de meilleur qualité. Occasionnellement, une *mutation* est appliquée. Cet opérateur va venir altérer, le plus souvent aléatoirement, une partie du patrimoine génétique de l'individu fraîchement créé. Ainsi, une nouvelle population est constituée et le processus est réitéré en réévaluant la population puis en appliquant les trois opérateurs (sélection, croisement, mutation) jusqu'à ce que le critère de terminaison soit atteint. La figure C.1 illustre les différentes phase typiques du processus d'exécution des algorithmes génétiques

Etant une approche basée sur une population, les algorithmes génétiques sont bien adaptés à la résolution de problèmes d'optimisation multiobjectifs. Un algorithme génétique générique mono objectif, peut être modifié pour trouver un ensemble de solutions non-dominées en une seule exécution. La capacité des algorithmes génétiques à explorer simultanément différentes régions de l'espace des solutions, permet de trouver divers jeux de solutions à des problèmes

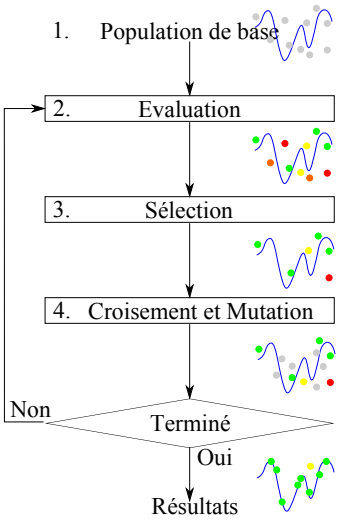



Figure C.1 – Synoptique du flot de fonctionnement des algorithmes génétiques. L'algorithme commence par constituer une population initiale, puis évalue chaque individu de la population et sélectionne les meilleurs individus afin de les reproduire en appliquant les opérateurs de croisement et de mutation. Ce processus se répète jusqu'à ce que le critère de terminaison soit atteint.

difficiles avec un espace de solution non-convex, discontinu et multimodale.

Le premier GA multiobjectif, appelé *vector evaluated GA* a été proposé par Schaffer [105]. Par la suite, de nombreux autres algorithmes multiobjectifs évolutionistes furent développés tels que : *Multi Objective Génétique Algorithm* [41], *Niched Pareto Genetic Algorithm* [53], *Weighted-based Genetic Algorithm*[51], *Random Weighted Genetic Algorithm*[93], *Non-dominated Sorting Genetic Algorithm*[112], *Strength Pareto Evolutionary Algorithm (SPEA)* [138], *improved SPEA* [137], *Pareto-Archived Evolution Strategy*[76], *Pareto Envelope-based Selection Algorithm*[28], *Region-based Selection in Evolutionary Multiobjective Optimization*[29], *Fast Non-dominated Sorting Genetic Algorithm* [35], *Multi-objective Evolutionary Algorithm* [2], *Micro-GA* [25], *Rank-Density Based Genetic Algorithm*[85], et *Dynamic Multi-objective Evolutionary Algorithm*[129]. Il est a noter qu'il y a, dans la littérature, de nombreuses variations de GA multiobjectifs. Les GA cités, sont bien connus et ont été utilisés dans de nombreuses applications et leurs performances ont été testées dans de nombreuses études comparatives. De nombreuses veilles scientifiques, concernant l'optimisation multiobjectifs évolutionniste, ont été publiées [136],[138],[26]–[77]. En général, les GA multiobjectifs diffères principalement sur

leur procédure d'assignation de fitness, elitisme ou de l'approche de diversification.

## C.2 Encodage des chromosomes

Il y a deux éléments distincts dans les GA, les individus et les populations, comme le suggère la section précedente. Un individu est une unique solution alors que les populations sont un ensemble d'individus impliqués dans le processus de recherche. Le cadre formel des GA considère deux formes de solutions tel que décrit en Figure C.2 :

1. Le chromosome, qui est l'information génétique "brute" avec laquelle les GA travaillent.
2. Le phénotype, qui est l'expression du chromosome (génotype) dans les termes du modèle.

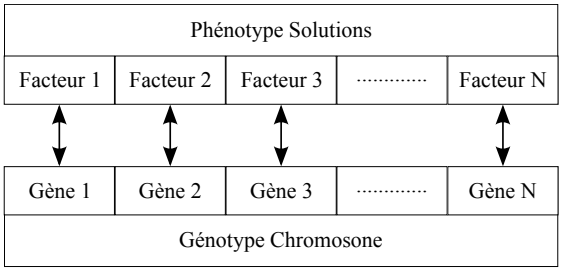



Figure C.2 – Représentation du *Phénotype* et du *Génotype*

Un chromosome est subdivisé en gènes. Un gène est la représentation unique qu'utilise le GA pour un facteur de contrôle. A chaque facteur, dans l'ensemble des solutions, correspond un gène dans le chromosome. La figure C.3 montre la représentation d'un génotype. Un chromosome doit en quelque sorte contenir les informations concernant la solution qu'il représente.

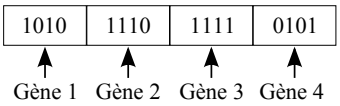



Figure C.3 – Représentation d'un génotype

L'encodage des chromosomes est le processus de représentation des gènes individuels. Ce processus peut être réalisé en utilisant des bits, des nombres, des arbres, des tableaux, des listes

ou bien tout autre objet. L'encodage dépend principalement du problème à résoudre. Le plus courant des encodages est une chaîne binaire où chaque bit peut représenter une caractéristique de la solution. L'encodage binaire rend possible un grand nombre de chromosomes avec un très petit nombre de allèles.

## C.3 Fonction fitness

Différentes approches de fonction fitness existent pour répondre aux problèmes d'optimisation multiobjectifs ([77]). Les techniques les plus utilisées sont la somme pondérée (*weighted sum*), l'altération de la fonction objectif (*altering objective function*) et le classement pareto (*pareto-ranking*).

### C.3.1 Approche par Somme pondérée

L'approche par somme pondérée est l'approche classique pour résoudre les problèmes d'optimisation multiobjectifs. Celle-ci assigne un poids $w_i$ à chaque fonction objectif normalisée $z'_i(x)$ tel que le problème soit transformé en un problème mono objectif avec une fonction objetif scalaire de la forme :

$$\min z = w_1 z'_1(x) + w_2 z'_2(x) + \dots w_k z'_k(x) \tag{C.1}$$

où $z'_i(x)$ est la fonction objectif normalisée de $z_i(x)$ et $\sum w_i(x) = 1$. Cette approche est parfois appelée "approche à priori" puisque l'utilisateur est censé fournir les coefficients de pondération (poids). Résoudre un problème avec la fonction objective de C.1 pour un vecteur de pondération donné $\mathbf{w} = w_1, w_2, \dots, w_k$ ne produira qu'une seule solution. Si plusieurs solutions sont désirées, le problème doit être résolu plusieurs fois avec différentes combinaisons de pondérations. Le principal inconvénient de cette méthode est de sélectionner un vecteur de pondération pour chaque exécution. Hajela et Lin [51] propose une méthode d'automatisation dans laquelle chaque solution $x_i$ de la population embarque le vecteur de pondération $w_i$. Ainsi, plusieurs solutions peuvent être simultanément explorées durant une seule exécution. De plus, les vecteurs de pondérations peuvent être ajustés afin de promouvoir la diversité au sein de la population.

Dans d'autres travaux [93], [94], un vecteur de pondération est généré aléatoirement durant la phase de sélection à chaque génération pour chaque solution $x_i$. Cette approche se veut stipuler de multiples directions de recherche sans paramètres additionnels. Les principaux avantages de l'approche par somme pondérée est qu'elle est relativement simple à implémenter et relativement efficace en terme de temps d'optimisation. L'inconvénient majeur, est que toutes les solutions pareto-optimale ne peuvent être trouvées lorsque le front paréto réel n'est pas convexe.

### C.3.2 Approche par Altération de la fonction objectif

[105] est le premier GA utilisé pour approximer l'ensemble des points pareto-optimale avec un ensemble de solution non-dominées. Dans [105] et de façon similaire dans [78], la population $P_t$ est aléatoirement divisée en $K$ sous-populations de taille égale : $P_1, P_2, \ldots, P_K$ ; Puis, à chaque solution de la sous population $P_i$ est associée une valeur fitness basée sur la fonction objectif $z_i$. Les solutions sont sélectionnées dans ces sous populations en utilisant une sélection proportionnelle. Le principal avantage de cette méthode est sa facilité de mise en oeuvre et sa rapidité d'exécution. Cependant, les sous population ont tendances à converger vers une "bonne" solution pour un seul objectif, mais de mauvaise qualité pour les autres.

### C.3.3 Approche par Classement Pareto

L'approche par classement pareto utilise explicitement le concept de dominance pareto en évaluant le fitness ou en assignant une probabilité de sélection aux solutions. La population est classée selon la règle de dominance puis à chaque solution est associée un fitness basé sur sont rang dans la population. La première technique de classement pareto fut proposée par Goldberg [46] dont l'Algorithme C.1 est donné ci-après.

D'autre travaux, tels que [112], [35] étendent et améliorent la technique proposée par [46]. Fleming [41] utilise une approche différente pour l'assignation du rang tel que :

$$r_2(x,t) = 1 + nq(x,t) \tag{C.2}$$

où $nq(x,t)$ est le nombre de solutions dominant la solution $x$ à la génération $t$. Cette méthode de

**Algorithme C.1** Classement pareto selon Goldberg [46]

**procedure**
$Step1$ : Set $i = 1$ et $TP = P$
$Step2$ : Identify non-dominated solutions in $TP$ and assigned them set to $F_i$
$Step3$ : Set $TP = TPF_i$
**if** $TP = \varnothing$ **then**
goto $Step4$
**else**
$i = i + 1$
goto $Step2$
**end if**
$Step4$ :
**for all** solution $x$ in population $P$ at generation $t$ **do**
**if** $x \in F_i$ **then**
rank $r_1(x,t) = i$
**end if**
**end for**
**end procedure**

classement pénalise les solutions localisées dans les régions de l'espace de la fonction objectif qui sont dominées par les sections du front pareto ayant une densité de population importante. [138] utilise une procédure de classement pour assigner les meilleurs fitness aux solutions non-dominées des régions de l'espace objectif sous représentés. Une liste $E$ de taille fixe stock les solutions non-dominées qui ont été traitées plus longtemps durant la recherche. Pour chaque solution $y \in E$ une intensité définit telle que :

$$s(y,t) = \frac{np(x,t)}{N_p + 1} \tag{C.3}$$

où $p(x,t)$ est le nombre de solutions de $P$ que $y$ domine. Le rang $r(y,t)$ d'une solution $y \in E$ est assigné tel que $r_3(y,t) = s(y,t)$ et le rang d'une solution $x \in P$ est calculé de la façon :

$$r_3(x,t) = 1 + \sum_{y \in E,\ y > x} s(y,t) \tag{C.4}$$

La stratégie de [85] pénalise aussi la redondance due aux sous représentations dans la population. Celle-ci est la suivante :

$$r_4(x,t) = 1 + \sum_{y \in P,\ y > x} r(y,t) \tag{C.5}$$

## C.4 Diversité : assignation de fitness, partage de fitness et *niching*

Maintenir la diversité d'une population est une considération importante dans les GA multiobjectifs afin d'obtenir des solutions uniformément distribuées sur le front pareto. En l'absence de mesure préventive, la population tend à former de petits agglomérats. Ce phénomène est appelé dérive génétique (*genetic drift*) et de nombreuses méthodes ont étées développées afin de l'éviter.

### C.4.1 Partage de fitness

Le partage de fitness encourage la recherche dans les sections non explorées du front pareto en réduisant artificiellement le fitness des solutions se trouvant dans une zone fortement peuplée. Dans ce but, ces dernières sont identifiées et une méthode pour pénaliser les solutions localisées dans ces zones est utilisée. L'idée de partage de fitness a été proposée par Goldberg et Richardson [47] dans la recherche d'optimum locaux pour les fonctions multimodales. Fonseca et Fleming [41] utilisent cette idée pour pénaliser des solutions agglomérées ayant le même rang de la façon suivante :

*Step 1* : Calculer la distance Euclidienne entre chaque paire de solutions $x$ et $y$ dans l'espace objectif normalisé entre 0 et 1 de la façon suivante :

$$dz(x,y) = \sqrt{\sum_{k=1}^{K} \left( \frac{z_k(x) - z_k(y)}{z_k^{max} - z_k^{min}} \right)^2} \tag{C.6}$$

où $z_k^{max}$ et $z_k^{min}$ sont les valeurs maximale et minimale de la fonction objectif $z_k(.)$, observées durant la recherche.

*Step 2* : Sur la base de ces distances, calculer le nombre de *niche* pour chaque solution $x \in P$ tel que :

$$nc(x,t) = \sum_{y \in P,\ r(y,t)=r(x,t)} \max \left\{ \frac{\sigma_{share} - dz(x,y)}{\sigma_{share}}, 0 \right\} \tag{C.7}$$

où $\sigma_{share}$ est la taille de la "niche".

*Step 3* : Après le calcul du nombre de *niche*, le fitness de chaque solution est ajusté de la

façon suivante :

$$f'(x,t) = \frac{f(x,t)}{nc(x,t)} \tag{C.8}$$

Dans la procédure ci-dessus, $\sigma_{share}$ definit un voisinage des solutions dans l'espace objectif. Les solutions d'un même voisinage contribuent au nombre de *niche* des autres. Ainsi, un solution ayant un voisinage surpeuplé aura un plus grand nombre de *niche*, réduisant la probabilité de sélection de cette solution comme parent. Il en résulte que le niching limite la prolifération de solutions dans un voisinage particulier de l'espace objectif. Une autre alternative, est d'utiliser la distance dans l'espace des variables de décision entre deux solutions $x$ et $y$ pour calculer le nombre de *niche*s tel que :

$$dx(x,y) = \sqrt{\frac{1}{M}\sum_{i=1}^{M}(x_i - y_i)^2} \tag{C.9}$$

Cette équation est une mesure de la différence structurelle entre deux solutions. Deux solutions peuvent être très proches dans l'espace objectif alors qu'elles sont très différentes d'un point de vue structurelle. De plus, le partage de fitness basé sur l'espace de la fonction objectif peut réduire la diversité dans l'espace des variables de décisions. Cependant, Deb et Goldberg [34] montrent que le partage de fitness dans l'espace objectif donne de meilleurs résultats que celui basé sur l'espace des variables décisions. Un des inconvénients du partage de fitness basé sur le décompte des *niche*s, est que l'utilisateur doit sélectionner un nouveau paramètre $\sigma_{share}$. Un autre inconvénient du niching est l'effort de calcul nécessaire pour compter le nombre de *niche*s. Cependant, les bénéfices du partage de fitness dépassent ce surcoût de calcul.

[41] fut le premier GA multiobjectifs utilisant explicitement le classement pareto couplé avec une technique de niching, afin d'encourager la recherche vers le front pareto réel tout en maintenant la diversité de la population. La procédure est la suivante :

Dans [137], une mesure de densité est utilisée pour discerner les solutions ayant le même rang, où la densité d'une solution est définit comme l'inverse de la distance de son $k^{imme}$ plus proche voisin dans l'espace objectif. La densité d'une solution est similaire à son nombre de *niche*. Cependant, sélectionner une valeur pour $k$ est plus aisé que de sélectionner une valeur pour $\sigma_{share}$

**Algorithme C.2** Couplage Classement pareto avec une technique de niching [41]

**procedure**

$Step1$ : Start with random initial population $P_0$ ; Set $t = 0$

$Step2$ : If the stopping criterion is satisfied, return $P_t$

$Step3$ : Evaluate fitness of the population as follow :

$Step3.1$ : Assign a rank $r(x,t)$ to each solution $x \in P_t$ using the ranking scheme given in Eq. C.2

$Step3.2$ : Assign a fitness values to each solution based on the solution's rank as follow [33] :

$f(x,t) = N - \sum_{k=1}^{r(x,y)-1} n_k - 0.5 \times (n_r(x,t) - 1)$

where $n_k$ is the number of solutions with rank k

$Step3.3$ : Calculate the niche count $nc(x,t)$ of each solution $x \in P_t$ using Eq C.7

$Step3.4$ : Calculate the shared fitness value of each solution $x \in P_t$ as follow :

$f'(x,t) = f(x,t)/nc(x,t)$

$Step3.5$ : Normalize the fitness values by using the shared fitness values

$f''(x,t) = \frac{f'(x,t);n_r(x,t)}{\sum_{y \in P_t,\ r(y,t)=r(x,t)} f'(x,t)} f(x,t)$

$Step4$ : Use stochatic selection method based on $f''$ to select parents for the mating pool. Apply crossover and the mutating pool until offspring population $Q_t$ of size $N$ is filled. Set $P_{t+1} = Q_t$

$Step5$ : Set $t = t + 1$, goto $Step2$

**end procedure**

**Algorithme C.3** *Crowding Distance* selon [35]

**procedure**

$Step1$ : Rank the population and identify non-dominated fronts $F_1, F_2, \ldots, F_R$.

$Step2$ :

**for all** front $j = 1, \ldots, R$ **do**

**for all** objective function $k$ **do**

sort the solutions in $F_j$ in the ascending order.

Let $l = |F_j|$ and $x_{[1,k]}$ represent the $i^{th}$ solution in sorted list with respect to the objective function $k$.

Assign $cd_k(x_{[1,k]}) = \infty$ and $cd_k(x_{[l,k]}) = \infty$

**for all** $i = 2, \ldots, l-1$ **do**

Assign $cd_k(x_{[i,k]}) = \frac{z_k(x_{[i+1,k]}) - z_k(x^k_{[i-1,k]})}{z_k^{max} - z_k^{min}}$

**end for**

**end for**

**end for**

$Step3$ : To find the total crowding distance $cd(x)$ of a solution $x$, sum solution's crowding distances with respect to each objective.

**end procedure**

### C.4.2 *Crowding Distance*

L'approche de *crowding distance* veut obtenir une dispertion uniforme des solutions le long du meilleur front pareto connu, sans utiliser un paramètre de partage de fitness (fitness sharing). Par exemple, [35] utilise cette méthode sous la forme suivante (Figure C.4) :

Le principal avantage de l'approche de crowding est que la mesure de la densité de population autour d'une solution est calculée sans interventions de paramètres définit par l'utilisateur,

tel que $\sigma_{share}$ ou le $k^{ieme}$ plus proche voisin.

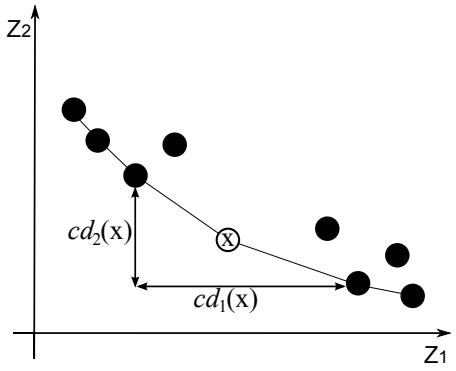



Figure C.4 – Exemple de *Crowding Distance*

### C.4.3 Densité basée sur une cellule (*Cell-Based*)

Dans cette approche ([85], [129], [75]), l'espace objectif est divisé en $K$ cellules (Figure C.5). Le nombre de solutions dans chaque cellule est définit comme la densité de la cellule et la densité de la solution est égale à la densité de la cellule dans lequel la solution est localisée. Cette densité est utilisée pour atteindre une certaine diversité de la même façon que dans le *fitness sharing*. Par exemple, entre deux solutions non-dominées, celle avec une densité plus faible est préférable ([28]). La procedure est la suivante :

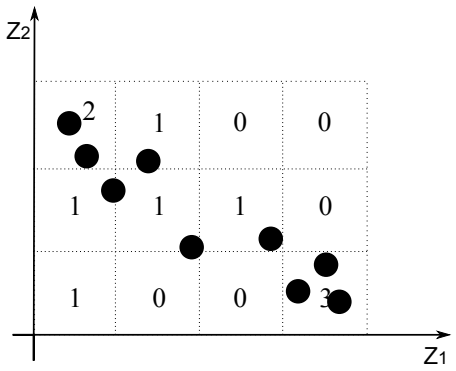



Figure C.5 – Exemple de densité *Cells-Based*

**Algorithme C.4** Algorithme *Cell-Based*

**procedure**

$N_E$ = the maximum size of non-dominated archive $E$

$N_P$ = the population size

$n$ = the number of grids along each objective function axix.

$Step1$ : Start with random initial population $P_0$ ans set external archive $E_0 = \varnothing$, $t = 0$

$Step2$ : Divide the normalized objective space into $n^k$ hyper-cubes where $n$ is the number of grids along a single objectiveaxis and $K$ is the number of objectives.

$Step3$ : Update non-dominated archive $E_t$ by incorporating new solution from $P_t$ one by one as follow :

$Case1$ : If new solution is dominated by at least a solution in $E_t$, discerd the new solution.

$Case2$ : If new solution dominated some solutions in $E_t$, removed those dominated solutions from $E_t$ and add the new solution to $E_t$. Update the membership of the hypeer-cubes.

$Case3$ : If a new solution is not dominated by and does not dominate any solution in $E_t$, add this solution to $E_t$. If $|E_t| = N_E + 1$, randomly choose a solution from the most crowded hyper-cubes to be removed. Update the member ship of hyper-cubes.

$Step4$ : If the criterion is satisfied, stop end return $E_t$

$Step5$ : Set $P_t = \varnothing$, and delect solution from $E_t$ for crossover and mutation based on the density information of the hypercubes. Apply crossover and mutation to genérate $N_p$ offspring and copy them to $P_{t+1}$

$Step6$ : Set $t = t + 1$ and goto $Step3$

**end procedure**

Le principal avantage de cette technique est qu'une carte des densitées globale de l'espace de la fonction objectif, est obtenue à l'issue du calcul de densité.

### C.4.4 Elitisme

Dans un contexte de GA mono-objectif, *élitisme* signifie que la meilleure solution trouvée drurant l'exploration, survie à la prochaine génération. Dans cet optique, toutes les solutions non-dominées découvertes par le GA multiobjectifs sont considérées comme des solutions d'élites. Cependant, implémenter l'élitisme dans une optimisation multiobjectifs n'est pas aussi facile que pour une optimisation mono objectif, principalement dû au grand nombre possible de solution d'élite. Les premiers Ga multiobjectifs n'utilisaient pas d'élitisme. Cependant, la plupart des GA multiobjectifs récents l'utilisent. Comme décrit dans [138], [33] et [121], les GA avec une stratégie d'élitisme tendent à surpasser les GA sans élitsme. Les GA multiobjectifs utilisent deux stratégies pour implementer l'élitisme : (i)en maintenant les solutions d'élites dans la population et (ii)stocker les solutions d'élites dans un liste secondaire et en les reintroduisant à la population.

# Bibliographie


[1] Jpeg 2000 image coding system (jpeg 2000 part i final committee draft version 1.0). March 2000. 135, 136, 137

[2] M. A. Abido. A new multiobjective evolutionary algorithm for environmental economic power dispatch, 2001. 205

[3] K. Agarwal, K. Nowka, H. Deogun, and D. Sylvester. Power gating with multiple sleep modes. In *ISQED '06 : Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 633–637, Washington, DC, USA, 2006. IEEE Computer Society. 181

[4] A. Andrei, P. Eles, Z. Peng, M. Schmitz, and B. Hashimi. Energy optimization of multiprocessor systems on chip by voltage selection. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(3) :262 –275, march 2007. 168

[5] G. Ascia, V. Catania, and M. Palesi. Multi-objective mapping for mesh-based noc architectures. In *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, pages 182 – 187, sept. 2004. 39

[6] D. R. Avresky, C. M. Cunningham, and H. Ravichandran. Fault-tolerant adaptive routing for two-dimensional meshes. *Int. Journal of Computer Systems Science and Engineering*, 14(6), november 1999. 132

[7] H. Aydin and D. Zhu. Reliability-aware energy management for periodic real-time tasks. *Computers, IEEE Transactions on*, 58(10) :1382 –1397, oct. 2009. 42

[8] E. Beigne, F. Clermidy, S. Miermont, and P. Vivet. Dynamic voltage and frequency scaling architecture for units integration within a gals noc. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 129 –138, april 2008. 143

[9] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *-NUM-MATH*, 4 :238–252, dec 1962. 39

[10] L. Benini, D. Bertozzi, A. Guerri, and M. Milano. Allocation, scheduling and voltage scaling on energy aware mpsocs. In *In Procs. of CPAIOR-2006*, pages 44–58, 2006. 39

[11] L. Benini, D. Bertozzi, A. Guerri, M. Milano, and F. Poletti. Measuring efficiency and executability of allocation and scheduling in multi-processor systems-on-chip. *Intelligenza Artificiale*, 2(3) :13–20, 2005. 39

[12] L. Benini, G. de Micheli, E. Macii, D. Sciuto, and C. Silvano. Asymptotic zero-transition activity encoding for address busses in low-power microprocessor-based systems. In *GLS '97 : Proceedings of the 7th Great Lakes Symposium on VLSI*, page 77, Washington, DC, USA, 1997. IEEE Computer Society. 175

[13] S. Borkar, T. Karnik, and V. De. Design and reliability challenges in nanometer technologies. *Design Automation Conference, 2004. Proceedings. 41st*, pages 75–75, 2004. 183

[14] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *DAC '03 : Proceedings of the 40th conference on Design automation*, pages 338–342, New York, NY, USA, 2003. ACM. 11, 183, 184

[15] K. Bowman, B. Austin, J. Eble, X. Tang, and J. Meindl. A physical alpha-power law mosfet model. In *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, pages 218 – 222, 1999. 169

[16] K. A. Bowman, S. G. Duvall, and J. D. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *Solid-State Circuits, IEEE Journal of*, 37(2) :183–190, 2002. 191

[17] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture : a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996. 144

[18] B. Calhoun and A. Chandrakasan. Ultra-dynamic voltage scaling using sub-threshold operation and local voltage dithering in 90nm cmos. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, 1 :300–599, Feb. 2005. 181

[19] F. Chaix, D. Avresky, N. Zergainoh, and M. Nicolaidis. Fault-tolerant deadlock-free adaptive routing for any set of link and node failures in multi-cores systems. In *Network Computing and Applications (NCA), 2010 9th IEEE International Symposium on*, pages 52 –59, july 2010. 132

[20] F. Chaix, D. Avresky, N.-E. Zergainoh, and M. Nicolaidis. A fault-tolerant deadlock-free adaptive routing for on chip interconnects. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1 –4, march 2011. 132, 148

[21] M.-J. Chen, H.-T. Huang, C.-S. Hou, and K.-N. Yang. Back-gate bias enhanced band-to-band tunneling leakage in scaled mosfet's. *Electron Device Letters, IEEE*, 19(4) :134 –136, apr 1998. 170

[22] Y. Cheng, M.-C. Jeng, Z. Liu, J. Huang, M. Chan, K. Chen, P. K. Ko, and C. Hu. A physical and scalable i-v model in bsim3v3 for analog/digital circuit simulation. *Electron Devices, IEEE Transactions on*, 44(2) :277 –287, feb 1997. 168, 170

[23] C.-L. Chou and R. Marculescu. Run-time task allocation considering user behavior in embedded multiprocessor networks-on-chip. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(1) :78 –91, jan. 2010. 85

[24] C.-L. Chou, U. Ogras, and R. Marculescu. Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10) :1866 –1879, oct. 2008. 85

[25] C. Coello and G. Toscano Pulido. A micro-genetic algorithm for multiobjective optimization. In E. Zitzler, L. Thiele, K. Deb, C. Coello Coello, and D. Corne, editors, *Evolutionary Multi-Criterion Optimization*, volume 1993 of *Lecture Notes in Computer Science*, pages 126–140. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-44719-9_9. 205

[26] C. A. C. Coello. A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques. *Knowledge and Information Systems*, 1(3) :129–156, 1999. 205

[27] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. 196

[28] D. Corne, J. D. Knowles, and M. J. Oates. The pareto envelope-based selection algorithm for multi-objective optimisation. In *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, PPSN VI, pages 839–848, London, UK, 2000. Springer-Verlag. 205, 213

[29] D. W. Corne, N. R. Jerram, J. D. Knowles, M. J. Oates, and J. Martin. PESA-II : Region-based Selection in Evolutionary Multiobjective Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2001*, pages 283–290, 2001. 205

[30] N. A. Cressie. *Statistics for spatial data*. John Wiley and Sons, Inc., 1993. 186, 187

[31] C. M. Cunningham and D. R. Avresky. Fault-tolerant adaptive routing for two-dimensional meshes. In *HPCA '95 : Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, page 122, Washington, DC, USA, 1995. IEEE Computer Society. 132

[32] S. Das, S. Pant, D. Roberts, S. Lee, D. Blaauw, T. Austin, T. Mudge, and K. Flautner. A self-tuning dvs processor using delay-error detection and correction. *VLSI Circuits, 2005. Digest of Technical Papers. 2005 Symposium on*, pages 258–261, June 2005. 179

[33] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 1 edition, June 2001. 212, 214

[34] K. Deb and D. E. Goldberg. An Investigation of Niche and Species Formation in Genetic Function Optimization. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 42–50, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc. 211

[35] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm : Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2) :182 –197, apr 2002. 15, 87, 205, 208, 212

[36] R. P. Dick and N. K. Jha. Mogac : a multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, ICCAD '97, pages 522–529, Washington, DC, USA, 1997. IEEE Computer Society. 40

[37] R. P. Dick, D. L. Rhodes, and W. Wolf. Tgff : task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, CODES/CASHE '98, pages 97–101, Washington, DC, USA, 1998. IEEE Computer Society. 98

[38] M. Drosg. Frequency and probability distributions. In *Dealing with Uncertainties*, pages 51–69. Springer Berlin Heidelberg, 2007. 10.1007/978-3-540-29608-9_5. 131

[39] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner. Razor : circuit-level correction of timing errors for low-power operation. *Micro, IEEE*, 24(6) :10–20, Nov.-Dec. 2004. 179

[40] M. F.Emnett. Power reduction through rtl clock gating. 2000. 174

[41] C. Fonseca and P. Fleming. Multiobjective genetic algorithms. In *Genetic Algorithms for Control Systems Engineering, IEE Colloquium on*, pages 6/1 –6/5, may 1993. 15, 205, 208, 210, 211, 212

[42] P. Friedberg, Y. Cao, J. Cain, R. Wang, J. Rabaey, and C. Spanos. Modeling within-die spatial correlation effects for process-design co-optimization. In *ISQED '05 : Proceedings of the 6th International Symposium on Quality of Electronic Design*, pages 516–521, Washington, DC, USA, 2005. IEEE Computer Society. 183, 187, 189

[43] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, January 1979. 36

[44] S. Gheorghita, S. Stuijk, T. Basten, and H. Corporaal. Automatic scenario detection for improved wcet estimation. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 101 – 104, june 2005. 39

[45] S. V. Gheorghita, T. Basten, and H. Corporaal. Intra-task scenario-aware voltage scheduling. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '05, pages 177–184, New York, NY, USA, 2005. ACM. 39

[46] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. 15, 79, 204, 208, 209

[47] D. E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 41–49, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc. 210

[48] R. Gonzalez, B. Gordon, and M. Horowitz. Supply and threshold voltage scaling for low power cmos. *Solid-State Circuits, IEEE Journal of*, 32(8) :1210 –1216, aug 1997. 169, 170

[49] B. Gorjiara, N. Bagherzadeh, and P. Chou. An efficient voltage scaling algorithm for complex socs with few number of voltage modes. In *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 381 – 386, aug. 2004. 37

[50] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives : a seamless flow of graphs transformations. In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003. 50

[51] P. Hajela and C. Y. Lin. Genetic search strategies in multicriterion optimal design. *Structural and Multidisciplinary Optimization*, 4 :99–107, 1992. 10.1007/BF01759923. 205, 207

[52] P. Hoang and J. Rabaey. Scheduling of dsp programs onto multiprocessors for maximum throughput. *Signal Processing, IEEE Transactions on*, 41(6) :2225 –2235, jun 1993. 72, 99

[53] J. Horn, N. Nafpliotis, and D. Goldberg. A niched pareto genetic algorithm for multiobjective optimization. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 82 –87 vol.1, jun 1994. 205

[54] J. Hu and R. Marculescu. Energy- and performance-aware mapping for regular noc architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(4) :551–562, April 2005. 85

[55] M.-E. Hwang, T. Cakici, and K. Roy. Interactive presentation : Process tolerant $\beta$-ratio modulation for ultra-dynamic voltage scaling. In *DATE '07 : Proceedings of the conference on Design, automation and test in Europe*, pages 1550–1555, San Jose, CA, USA, 2007. EDA Consortium. 180

[56] M. H.Yasuura, T.Ishihara. Energy management techniques for soc design. pages 177–223, 2006. 173

[57] ITRS. The international technology roadmap for semiconductors. 1999. 188, 189

[58] ITRS. The international technology roadmap for semiconductors. 2009. 21, 22

[59] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Synthesis of fault-tolerant schedules with transparency/performance trade-offs for distributed embedded systems. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1 –6, march 2006. 41

[60] H. Jingcao and R. Marculescu. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 234 – 239 Vol.1, feb. 2004. 38

[61] P. Juang, Q. Wu, L.-S. Peh, M. Martonosi, and D. W. Clark. Coordinated, distributed, formal energy management of chip multiprocessors. In *ISLPED '05 : Proceedings of the 2005 international symposium on Low power electronics and design*, pages 127–130, New York, NY, USA, 2005. ACM. 177

[62] A. Kahng. How much variability can designers tolerate. *Design and Test of Computers, IEEE*, 20(6) :96–97, Nov.-Dec. 2003. 189

[63] K. Kanda, K. Nose, H. Kawaguchi, and T. Sakurai. Design impact of positive temperature dependence of drain current in sub 1 v cmos vlsis. *Custom Integrated Circuits, 1999. Proceedings of the IEEE 1999*, pages 563–566, 1999. 190

[64] T. Karnik, S. Borkar, and V. De. Probabilistic and variation-tolerant design : Key to continued moore ?s law. *Proc. Workshop Timing Issues in Specification Synthesis Digital Systems*, Feb. 2004. 189

[65] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society. 200

[66] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20 :359–392, December 1998. 200, 201

[67] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society. 72, 100, 200, 201

[68] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48 :96–129, January 1998. 200, 201

[69] H. Kawaguchi, G. Zhang, S. Lee, and T. Sakurai. An lsi for vdd-hopping and mpeg4 system based on the chip. In *ISCAS (4)*, pages 918–921, 2001. 76

[70] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi. Low power methodology manual for system-on-chip design. 2008. 175

[71] A. Keshavarzi, S. Ma, S. Narendra, B. Bloechel, K. Mistry, T. Ghani, S. Borkar, and V. De. Effectiveness of reverse body bias for leakage control in scaled dual vt cmos ics. In *Low Power Electronics and Design, International Symposium on, 2001.*, pages 207 –212, 2001. 170

[72] A. Keshavarzi, S. Narendra, S. Borkar, C. Hawkins, K. Royi, and V. De. Technology scaling behavior of optimum reverse body bias for standby leakage power reduction in cmos ic's. In *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, pages 252 – 254, 1999. 171

[73] A. Keshavarzi, G. Schrom, S. Tang, S. Ma, K. Bowman, S. Tyagi, K. Zhang, T. Linton, N. Hakim, S. Duvall, J. Brews, and V. De. Measurements and modeling of intrinsic fluctuations in mosfet threshold voltage. *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, pages 26–29, Aug. 2005. 189

[74] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan. Leakage current : Moore's law meets static power. *Computer*, 36(12) :68 – 75, dec. 2003. 171

[75] J. Knowles and D. Corne. The pareto archived evolution strategy : a new baseline algorithm for pareto multiobjective optimisation. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 1, pages 3 vol. (xxxvii+2348), 1999. 213

[76] J. D. Knowles and D. W. Corne. Approximating the nondominated front using the pareto archived evolution strategy. *Evol. Comput.*, 8 :149–172, June 2000. 205

[77] A. Konak, D. W. Coit, and A. E. Smith. Multi-objective optimization using genetic algorithms : A tutorial. *Reliability Engineering & System Safety*, 91(9) :992–1007, Sept. 2006. 205, 207

[78] F. Kursawe. A variant of evolution strategies for vector optimization. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer Science*, pages 193–197. Springer Berlin / Heidelberg, 1991. 10.1007/BFb0029752. 208

[79] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. *Design Automation Conference, 2000. Proceedings 2000. 37th*, pages 806–809, 2000. 76

[80] H. Li, S. Bhunia, Y. Chen, T. N. Vijaykumar, and K. Roy. Deterministic clock gating for microprocessor power reduction. In *HPCA '03 : Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 113, Washington, DC, USA, 2003. IEEE Computer Society. 174

[81] Q. Li, Y. Ruan, ShidaYang, and T. Jiang. An optimal scheduling algorithm for fork-join task graphs. pages 587 – 589, aug. 2003. 122

[82] Y. Lin, M. Kudlur, S. Mahlke, and T. Mudge. Hierarchical coarse-grained stream compilation for software defined radio. In *CASES '07 : Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 115–124, New York, NY, USA, 2007. ACM. 122, 144

[83] X. Liu and S. Mourad. Performance of submicron cmos devices and gates with substrate biasing. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, pages 9 –12 vol.4, 2000. 171

[84] Z.-H. Liu, C. Hu, J.-H. Huang, T.-Y. Chan, M.-C. Jeng, P. Ko, and Y. Cheng. Threshold voltage model for deep-submicrometer mosfets. *Electron Devices, IEEE Transactions on*, 40(1) :86 –95, jan 1993. 168

[85] H. Lu and G. Yen. Rank-density-based multiobjective genetic algorithm and benchmark test function study. *Evolutionary Computation, IEEE Transactions on*, 7(4) :325 – 343, aug. 2003. 205, 209, 213

[86] J. Luo and N. K. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, ICCAD '00, pages 357–364, Piscataway, NJ, USA, 2000. IEEE Press. 37

[87] Y. Luo, J. Yu, J. Yang, and L. Bhuyan. Low power network processor design using clock gating. In *DAC '05 : Proceedings of the 42nd annual conference on Design automation*, pages 712–715, New York, NY, USA, 2005. ACM. 174

[88] Z. Ma, P. Marchal, D. P. Scarpazza, P. Yang, C. Wong, J. I. Gmez, S. Himpe, C. Ykman-Couvreur, and F. Catthoor. *Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogenous Platforms*. Springer Publishing Company, Incorporated, 1st edition, 2007. 94

[89] S. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads.

In *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, pages 721 – 725, nov. 2002. 168

[90] S. Miermont, P. Vivet, and M. Renaudin. A power supply selector for energy- and area-efficient local dynamic voltage scaling. In *PATMOS*, pages 556–565, 2007. 76

[91] B. Min, A. Andrei, P. Eles, and P. Zebo. On-line thermal aware dynamic voltage scaling for energy optimization with frequency/temperature dependency consideration. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 490 –495, july 2009. 168

[92] F. Muhammad, F. Muller, and M. Auguin. Contentions-conscious dynamic but deterministic scheduling of computational and communication tasks. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1487–1492, New York, NY, USA, 2006. ACM. 39

[93] T. Murata and H. Ishibuchi. Moga : multi-objective genetic algorithms. In *Evolutionary Computation, 1995., IEEE International Conference on*, volume 1, page 289, nov-1 dec 1995. 205, 208

[94] T. Murata, H. Ishibuchi, and H. Tanaka. Multi-objective genetic algorithm and its applications to flowshop scheduling. *Comput. Ind. Eng.*, 30 :957–968, September 1996. 208

[95] E. Musoll. Mesh-based many-core performance under process variations : a core yield perspective. *SIGARCH Comput. Archit. News*, 37 :27–34, January 2010. 36

[96] B.-G. Nam, J. Lee, K. Kim, S. J. Lee, and H.-J. Yoo. A 52.4mw 3d graphics processor with 141mvertices/s vertex shader and 3 power domains of dynamic voltage and frequency scaling. *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 278–603, Feb. 2007. 177

[97] B.-G. Nam, J. Lee, K. Kim, S. J. Lee, and H.-J. Yoo. A low-power handheld gpu using logarithmic arithmetic and triple dvfs power domains. In *GH '07 : Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 73–80, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. 177

[98] A. Papoulis. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill Companies, February 1991. 186

[99] N. Ranaldo and E. Zimeo. An economy-driven mapping heuristic for hierarchical master-slave applications in grid systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 13 pp., april 2006. 144

[100] P. J. Ribeiro and P. Diggle. geor : A package for geostatistical analysis. *R-NEWS*, 1(2), 2001. 188

[101] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. *Proceedings of the IEEE*, 91(2) :305 – 327, feb 2003. 173

[102] T. Sakurai and R. Newton. Alpha-power law mosfet model and its applications to cmos inverter delay and other formulas. *Journal of Solid-State Circuits*, April 1990. 169, 190

[103] S. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Varius : A model of process variation and resulting timing errors for microarchitects. *Semiconductor Manufacturing, IEEE Transactions on*, 21(1) :3–13, Feb. 2008. 11, 186, 192

[104] T. Sato and Y. Kunitake. A simple flip-flop circuit for typical-case designs for dfm. In *ISQED '07 : Proceedings of the 8th International Symposium on Quality Electronic Design*, pages 539–544, Washington, DC, USA, 2007. IEEE Computer Society. 179

[105] J. D. Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 93–100, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc. 205, 208

[106] S. Segars, K. Clarke, and L. Goudge. Embedded control problems, thumb, and the arm7tdmi. *IEEE Micro*, 15(5) :22–30, 1995. 175

[107] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *MICRO 35 : Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 356–367, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. 177

[108] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *HPCA '02 : Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 29, Washington, DC, USA, 2002. IEEE Computer Society. 177

[109] G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. In *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 3 –16, 2000. 144

[110] D. Shin and J. Kim. Power-aware scheduling of conditional task graphs in real-time multiprocessor systems. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 408–413, New York, NY, USA, 2003. ACM. 40

[111] Y. Shin, S.-I. Chae, and K. Choi. Partial bus-invert coding for power optimization of system level bus. In *ISLPED '98 : Proceedings of the 1998 international symposium on Low power electronics and design*, pages 127–129, New York, NY, USA, 1998. ACM. 175

[112] N. Srinivas and K. Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evol. Comput.*, 2 :221–248, September 1994. 205, 208

[113] A. Srivastava, D. Sylvester, and D. Blaauw. Statistical analysis and optimization for vlsi : Timing and power. 2005. 186, 187

[114] M. Stan. Optimal voltages and sizing for low power [cmos vlsi]. In *VLSI Design, 1999. Proceedings. Twelfth International Conference On*, pages 428 –433, jan 1999. 170

[115] M. R. Stan and W. P. Burleson. Bus-invert coding for low-power i/o. *IEEE Trans. Very Large Scale Integr. Syst.*, 3(1) :49–58, 1995. 175

[116] B. E. Stine, D. S. Boning, and J. E. Chung. Analysis and decomposition of spatial variation in integrated circuit processes and devices. *IEEE Transactions on Semiconductor Manufacturing*, 10 :24–41, 1997. 183

[117] C.-L. Su, C.-Y. Tsui, and A. Despain. Low power architecture design and compilation techniques for high-performance processors. *Compcon Spring '94, Digest of Papers.*, pages 489–498, Feb-4 Mar 1994. 175

[118] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 401–410, New York, NY, USA, 2006. ACM. 62

[119] Y. Taur and T. H. Ning. *Fundamentals of modern VLSI devices*. Cambridge University Press, New York, NY, USA, 1998. 184

[120] R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Mitigating parameter variation with dynamic fine-grain body biasing. *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 27–42, Dec. 2007. 11, 193

[121] D. A. Van Veldhuizen and G. B. Lamont. Multiobjective evolutionary algorithms : Analyzing the state-of-the-art. *Evol. Comput.*, 8 :125–147, June 2000. 214

[122] G. Varatkar and R. Marculescu. Communication-aware task scheduling and voltage selection for total systems energy minimization. In *Computer Aided Design, 2003. ICCAD-2003. International Conference on*, pages 510 – 517, nov. 2003. 39

[123] H. Veendrick. Short-circuit dissipation of static cmos circuitry and its impact on the design of buffer circuits. *Solid-State Circuits, IEEE Journal of*, 19(4) :468 – 473, aug 1984. 170

[124] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. *SIGPLAN Not.*, 39(11) :248–259, 2004. 177

[125] Q. Wu, M. Pedram, and X. Wu. Clock-gating and its application to low power design of sequential circuits. *Custom Integrated Circuits Conference, 1997., Proceedings of the IEEE 1997*, pages 479–482, May 1997. 174

[126] J. Xiong, V. Zolotov, and L. He. Robust extraction of spatial correlation. In *ISPD '06 : Proceedings of the 2006 international symposium on Physical design*, pages 2–9, New York, NY, USA, 2006. ACM. 187

[127] P. Yang and F. Catthoor. Pareto-optimization-based run-time task scheduling for embedded systems. In *CODES+ISSS '03 : Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 120–125, New York, NY, USA, 2003. ACM. 93, 94, 96, 157

[128] T. Ye, L. Benini, and G. De Micheli. Analysis of power consumption on switch fabrics in network routers. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 524 – 529, 2002. 85

[129] G. Yen and H. Lu. Dynamic multiobjective evolutionary algorithm : adaptive cell-based rank and density estimation. *Evolutionary Computation, IEEE Transactions on*, 7(3) :253 – 274, june 2003. 205, 213

[130] Y. Yoshida, B.-Y. Song, H. Okuhata, T. Onoye, and I. Shirakawa. An object code compression approach to embedded processors. In *ISLPED '97 : Proceedings of the 1997 international symposium on Low power electronics and design*, pages 265–268, New York, NY, USA, 1997. ACM. 175

[131] H. Yu, B. Veeravalli, and Y. Ha. Leakage-aware dynamic scheduling for real-time adaptive applications on multiprocessor systems. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 493 –498, june 2010. 168

[132] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotleakage : A temperature-aware model of subthreshold and gate leakage for architects. Technical report, 2003. 191

[133] B. Zhao, H. Aydin, and D. Zhu. Reliability-aware dynamic voltage scaling for energy-constrained real-time embedded systems. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 633 –639, oct. 2008. 41

[134] B. Zhao, H. Aydin, and D. Zhu. Enhanced reliability-aware power management through shared recovery technique. In *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pages 63 –70, nov. 2009. 42

[135] H. C. Zhao, C. H. Xia, Z. Liu, and D. Towsley. A unified modeling framework for distributed resource allocation of general fork and join processing networks. *SIGMETRICS Perform. Eval. Rev.*, 38(1) :299–310, 2010. 122

[136] E. Zitzler, K. Deb, and L. Thiele. Comparison of multiobjective evolutionary algorithms : Empirical results. *Evol. Comput.*, 8 :173–195, June 2000. 205

[137] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2 : Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Gloriastrasse 35, CH-8092 Zurich, Switzerland, 2001. 205, 211

[138] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms : a comparative case study and the strength pareto approach. *Evolutionary Computation, IEEE Transactions on*, 3(4) :257 –271, nov 1999. 205, 209, 214

# Publications Personnelles


– Gilles Bizot, Dimiter Avresky, Fabien Chaix, Nacer-Eddine Zergainoh, and Michael Nicolaidis. Self-recovering parallel applications in multi-core systems. In Network Computing and Applications (NCA), 10th IEEE International Symposium on, pages 51–58, aug. 2011.

– Gilles Bizot, Nacer-Eddine Zergainoh, and Michael Nicolaidis. Variability and reliability-aware application tasks scheduling and power control (voltage and frequency scaling) in the future nanoscale multiprocessors system on chip. In On-Line Testing Symposium, IOLTS. 15th IEEE International, page 155, june 2009.

– Gilles Bizot, Dimiter Avresky, Fabien Chaix, Nacer-Eddine Zergainoh, and Michael Nicolaidis. Adaptive mapping of parallelized application (fork-join dag) on multicore system in the presence of multiple failures. In 17th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS), Shanghai, China, Jul. 2012.

– Gilles Bizot, Nacer-Eddine Zergainoh, and Michael Nicolaidis. Energy and variability aware scheduling for clusterized mpsoc architecture. In 2nd IEEE Design for Reliability and Variability (DRVW), Austin, TX, USA, Nov. 2009.

– Fabien Chaix, Gilles Bizot, Michael Nicolaidis, and Nacer-Eddine Zergainoh. Variability-aware task mapping strategies for many-cores processor chips. In On-Line Testing Symposium (IOLTS), IEEE 17th International, pages 55 –60, july 2011.

– Fabien Chaix, Gilles Bizot, Michael Nicolaidis, and Nacer-Eddine Zergainoh. Variability-aware task mapping strategies for many-cores processor chips. In 4th IEEE Design for Reliability and Variability Workshop (DRVW), Dana Point, CA, USA, May 2011.

## Titre

**Gestion de l'activité et de la consommation dans les architectures multi-coeurs massivemnt parallèles**


## Résumé

Les variabilités du processus de fabrication des technologies avancées sont de plus en plus difficiles à maîtriser. Elles impactent plus sévèrement la fréquence de fonctionnement et la consommation d'énergie, et induisent de plus en plus de défaillances dans le circuit. Ceci est particulièrement vrai pour les MPSoCs, où le nombre de coeurs de calculs est très important. Ces travaux étudient différentes approches permettant le placement et l'ordonnancement d'applications dans des systèmes MPSoCs massivement parallèles. Les aspects variabilité, consommation, performance et tolérance aux fautes sont pris en compte. Dans un premier temps, nous proposons une méthodologie hybride (mixte statique - dynamique) permettant le traitement d'applications complexes et de grandes tailles de façon automatisée. La prise en compte des variabilités du processus de fabrication, permet de réduire les pertes de performances et les "gaspillages" d'énergies potentielles. Dans un deuxième temps, nous proposons une stratégie de placement dynamique auto-adaptative, permettant le recouvrement de tâches lors de défaillances. Celle-ci, garantie la terminaison de l'application sans avoir recours à la prise de « check-points ». Cette technique, est complétée par des algorithmes adaptatifs distribués, qui permettent d'optimiser le compromis performance - consommation en prenant en compte la variabilité.



## Mots-Clés :

MPSoC, NoC, variabilité, placement, ordonnancement, tolérance aux fautes, faible consommation d'énergie.


## Title

**Activity and Power management in massively parallel multi-cores architectures**


## Abstract

With the advanced technologies, it is more and more difficult to control the manufacturing variability's. It impacts more severely the working frequency and the consumed energy, and induces more and more failures inside the device. This is particularly true for MPSoC with a large number of computing cores. This work studies several approaches allowing the mapping and scheduling of applications in the massively parallel MPSoC systems. Variability, consumption, performance, and fault tolerances aspects are taken into account. Firstly, we propose an automated hybrid methodology (mix static - dynamic) enabling to deal with large and complex applications. The variability awareness makes it possible to reduce the performance losses and the potential waste of energy. Secondly, we propose a dynamic self-adaptive mapping strategy, allowing the tasks recovering in the presence of failures. It guarantees the termination of the application, without the « check-points » requirement. This technique has been extended with adaptive distributed algorithms, optimizing the performance-consumption tradeoffs by taking into account the variability.



## Keywords :

MPSoC, NoC, variability, mapping, scheduling, fault tolerance, low power.



**Laboratoire TIMA - 46 avenue Félix Viallet 38000 Grenoble, France**



**ISBN : 978-2-11-129170-6**