Estimation de la consommation dans la conception système des applications embarquées temps réels

Johann Laurent

N° d'ordre : 18

THESE

pour obtenir le grade de :

DOCTEUR DE l'UNIVERSITE DE BRETAGNE SUD

Spécialité **: Sciences de l'Ingénieur**
Mention **: Electronique et Informatique Industrielle**

Présentée et soutenue publiquement par

JOHANN LAURENT

Le 9 décembre 2002

Estimation de la consommation dans la conception système des applications embarquées temps réels

---

COMPOSITION DU JURY

M. M. RENAUDIN	TIMA, INPG Grenoble	Président
M. C. PIGUET	Ecole Polytechnique de Lausanne	Rapporteur
M. O. SENTIEYS	ENSSAT, Université de Rennes 1	Rapporteur
M. P. KAJFASZ	THALES Communications	Examinateur
M. E. MARTIN	Université de Bretagne Sud	Directeur de thèse
Me. N. JULIEN-TALON	Université de Bretagne Sud	Examinatrice

Laboratoire d'Electronique des Systèmes Temps Réel (LESTER)
Université de Bretagne Sud

REMERCIEMENTS

Je remercie tout particulièrement Monsieur Eric Martin, Professeur des Universités de l'Université de Bretagne Sud et Directeur du laboratoire LESTER, pour m'avoir accueilli dans son équipe de recherche et accepté d'être mon directeur de thèse. Je le remercie également pour les conseils qu'il m'a apportés tout au long de ces trois années.

Je tiens à remercier Madame Nathalie Julien-Talon et Monsieur Eric Senn, Maîtres de Conférences à l'Uiversité de Bretagne Sud pour leur disponibilité et leur encadrement. Je remercie également Mr Thierry Gourdeaux, Ingénieur de Recherche au laboratoire LESTER, pour ses précieux conseils lors du développement de l'outil automatique d'estimation SoftExplorer.

Je remercie Messieurs, Christian Piquet Professeur à l'Ecole Polytechnique de Lausanne et Olivier Sentieys Professeur des Universités de l'Université de Rennes 1, pour avoir accepté d'être rapporteurs de ce travail et membres du jury.

Je tiens également à remercier Messieurs, Philippe Kajfasz de la société Thalès Communications et Marc Renaudin Professeur à L'INPG de Grenoble, pour leur participation à ce jury.

Enfin, je remercie tous les membres du laboratoire LESTER que j'ai côtoyé au cours de ces trois dernières années (voire plus) pour leur amitié et leur bonne humeur. Je remercie également ceux qui auront le courage de lire cette thèse.

Table des matières

Table des figures

CHAPITRE I

CHAPITRE II

CHAPITRE III

CHAPITRE IV

CHAPITRE V

Table des tableaux

CHAPITRE II

CHAPITRE III

CHAPITRE IV

CHAPITRE V

Chapitre I

Introduction

Les techniques et les capacités d'intégration de circuits électroniques ne cessent d'évoluer au fil des années. Cette évolution facilite l'intégration d'applications toujours plus complexes. Jusqu'à très récemment, les concepteurs se préoccupaient seulement de la surface et des performances de leurs applications. Aujourd'hui, il est indispensable de prendre en compte leurs consommations de puissance et d'énergie. Pour mesurer l'impact de ses optimisations un concepteur doit avoir à sa disposition une méthode d'estimation rapide, peu complexe et fiable. Les travaux de cette thèse se focaliseront donc sur le développement d'une méthodologie d'estimation respectant ces contraintes.

Dans ce chapitre introductif, nous présenterons dans un premier temps le contexte de cette étude. Nous présenterons quelles seront les cibles architecturales ainsi que le domaine d'applications pris en compte dans cette étude. Enfin, le plan du manuscrit sera présenté.

I.1 Le contexte

I.1.1 Contexte matériel

Aujourd'hui, les concepteurs de systèmes électroniques ont un vaste espace de solutions architecturales. En effet, ils peuvent utiliser des processeurs généraux, des ASIC (Application Specific Integrated Circuit), des processeurs spécialisés (DSP…) sans compter les solutions FPGA (Field Programmable Gate Array). La figure ci-dessous représente une partie des solutions envisageables pour réaliser un codeur MPEG-2 ainsi que leur consommation associée **[DeMic00]**.

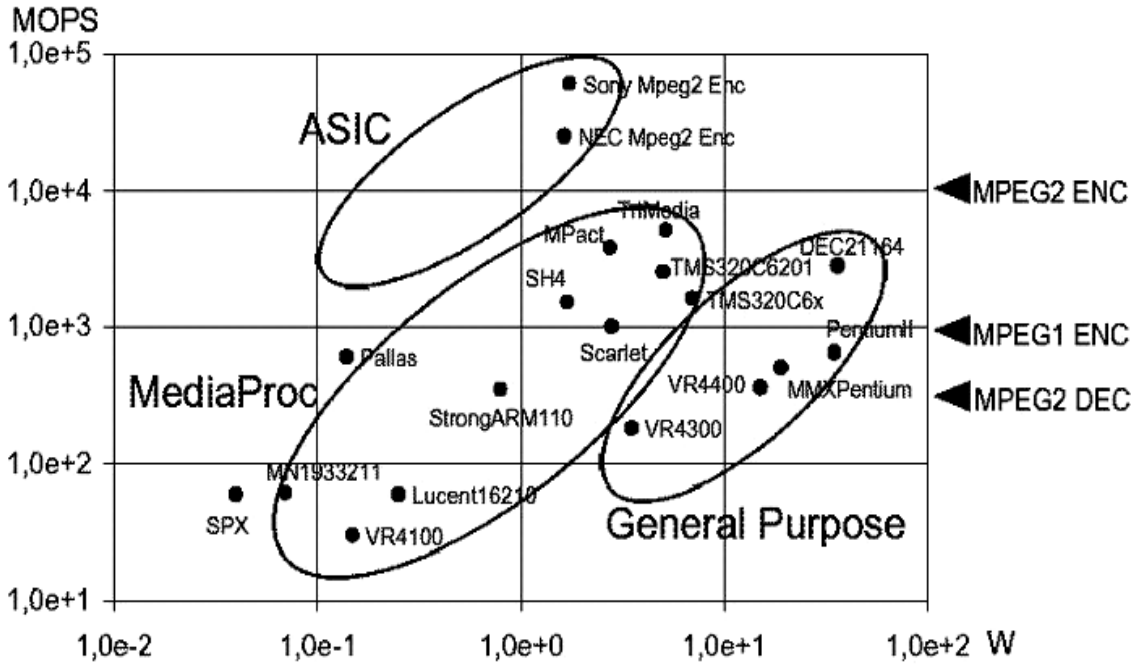

Figure I.1 Espace des solutions pour une application MPEG-2 [DeMic00]

Cette figure montre bien que différents choix technologiques s'offrent aux concepteurs mais que ceux-ci ont un impact sur la consommation de puissance.

Nous verrons que l'objectif de cette thèse est de proposer une méthode générale d'estimation de la consommation qui a pour originalité d'être peu complexe (élévation du niveau d'abstraction), d'être rapide tout en conservant une bonne précision. Au vue du nombre de possibilités architecturales, nous nous focaliserons dans cette thèse sur les processeurs et plus particulièrement sur les processeurs de traitements du signal (DSP). Cette seule famille est, elle, même constituée d'un certain nombre de cibles potentielles.

En effet, beaucoup de constructeurs proposent dans leur catalogue des DSP ; c'est le cas notamment de Motorola, d'Analog Device, d'Intel, de Lucent et de Texas Instruments.

Ces processeurs ont été développés pour améliorer l'exécution des applications de traitement du signal et de l'image. En effet, le nombre d'applications de ce type n'a cessé de croître depuis ces dix dernières années avec l'apparition des téléphones portables, des PDAs, des lecteurs de music MP3 ainsi que des lecteurs de DVD. L'avantage de ce type de composants par rapport aux ASICs est qu'il conserve la flexibilité des processeurs puisqu'ils exécutent des programmes logiciels. De plus, les DSPs permettent d'augmenter les performances par rapport aux processeurs généraux puisqu'ils possèdent des architectures spécialisées améliorant le calcul intensif.

Ces processeurs sont apparus dans les années 80 et n'ont dès lors pas cessé d'évoluer. Leur performance a été multipliée par 150 en 15 ans ce qui leur a permis d'être présent dans beaucoup de domaines d'applications (automobile, téléphone portable, chaîne Hi-fi etc…).

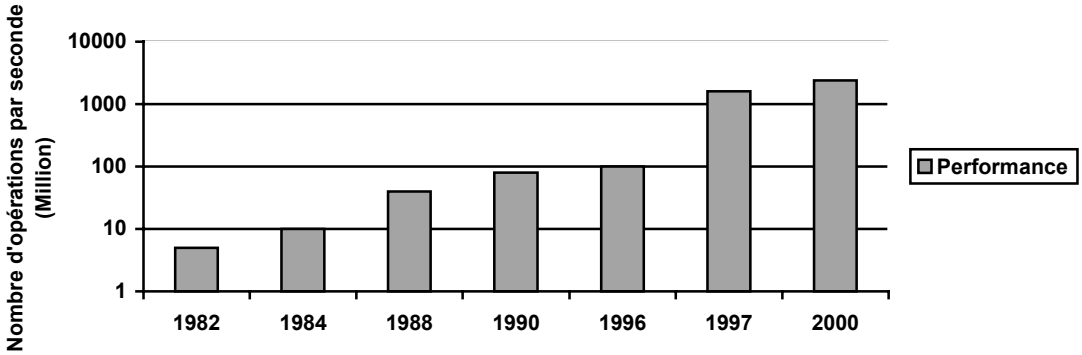

Figure I. 2 Evolution des performances des DSP

Aujourd'hui, les architectures de ces processeurs intègrent des mécanismes architecturaux similaires à ceux utilisés par les processeurs généraux. En effet, les dernières générations de DSP intègrent des pipelines profonds (jusqu'à une dizaine d'étages), des jeux d'instructions VLIW ou SIMD (Single Instruction Multiple Data), des mémoires caches (d'instructions et/ou de données) ainsi que des périphériques pour faciliter le transfert de données (DMA). De plus, les fréquences de fonctionnement suivent la même pente croissante que celles des processeurs généraux (quelques centaines de méga Hertz). La croissance de la complexité des architectures ainsi que celle des fréquences d'horloge entraînent une augmentation de la consommation de

telles architectures. En effet, en technologie CMOS la puissance dynamique est fonction de la capacité équivalente, de la fréquence et de la tension d'alimentation du circuit. Il devient alors important d'optimiser au mieux la consommation des applications notamment d'explorer les meilleurs mécanismes architecturaux permettant de diminuer cette consommation. Il faut donc pouvoir estimer cette consommation avant de pouvoir l'optimiser.

L'illustration de notre étude portera sur des processeurs de traitement du signal puisqu'ils nous permettront d'explorer un large panel des mécanismes architecturaux utilisés dans la conception de systèmes. Nous avons choisi les DSP de la société Texas Instruments puisqu'ils sont les leaders incontestés sur le marché. En effet, 75% des programmeurs de DSP utilisent leurs processeurs et 8 des 10 plus grands équipementiers de réseaux sans fil réalisent leurs applications avec des DSP Texas. De plus, ce constructeur possède une large gamme de DSP:

- Des DSP orientés contrôle (famille C2x)
- Des DSP orientés basse consommation (famille C54 et C55)
- Des DSP orientés performance (famille C6x)

Notre méthodologie devant s'appliquer à n'importe quel type de DSP, le choix de cette société nous permettra de valider notre méthode sur tous les types de DSP.

I.1.2 Contexte d'application

Pour ces travaux, nous nous placerons dans le cadre d'applications de traitement du signal et de l'image (TdSI). Le spectre de ces applications est large puisqu'elles permettent de réaliser :

- des filtrages (FIR, IIR, LMS)
- des codes correcteurs d'erreurs (Turbo code)
- des compressions d'images (JPEG)
- des compressions de flux vidéos (MPEG)

Ces applications nécessitent toutes un nombre important de calculs et certaines d'entre elles y rajoutent des besoins en contrôle et un grand nombre de transferts de données (utilisation importante des mémoires). De plus, elles ont généralement de fortes contraintes de temps réel ; elles sont donc tout à fait indiquées comme cadre d'application pour notre étude.

I.2 Objectif : Estimation de la consommation dans la conception système des applications embarquées temps réels

Le paragraphe précédent a mis en évidence que l'utilisation d'applications de plus en plus complexes nécessite l'emploi d'architectures adaptées aux traitements intensifs. Le principal effet de l'augmentation de la complexité des applications, donc des architectures, est l'augmentation de la consommation du système. Il devient aujourd'hui indispensable de prendre en compte l'aspect consommation. Or, la prise en compte de ce paramètre suppose le développement d'un modèle de consommation de la cible architecturale. Ce modèle doit répondre à trois critères principaux :

- Il doit être peu complexe malgré la complexité croissante des architectures.
- Sa mise en œuvre doit être rapide pour ne pas augmenter le temps de développement du système.
- Il doit être précis pour permettre aux concepteurs de prendre les bonnes décisions.

La rapidité de mise en œuvre du modèle est inversement proportionnelle à sa complexité. La seule façon de limiter la complexité du modèle de consommation est de relever son niveau d'abstraction i.e. de s'éloigner du niveau transistor.

Dans ce travail, nous développerons deux propositions :

- La première est de créer un modèle fonctionnel de l'architecture et non pas un modèle fonctionnel de l'ISA (Instruction Set Architecture) afin de réduire la complexité du modèle de consommation.
- La deuxième est de développer une méthodologie d'estimation en partant de la spécification du langage (C ou assembleur) et basée sur le modèle de consommation de la cible.

La deuxième proposition de ce travail entraînera donc le développement d'un outil automatique permettant d'estimer la consommation de puissance et d'énergie d'une application quelle que soit sa complexité. La complexité des algorithmes associés aux applications retenues dans le cadre de cette étude peut atteindre plusieurs dizaines de milliers d'opérations, voire même plusieurs centaines de milliers dans le cas, par exemple, d'algorithmes de traitement d'images.

I.3 Présentation du plan

Ce manuscrit est composé de six chapitres distincts comme indiqué sur la Figure I.3.

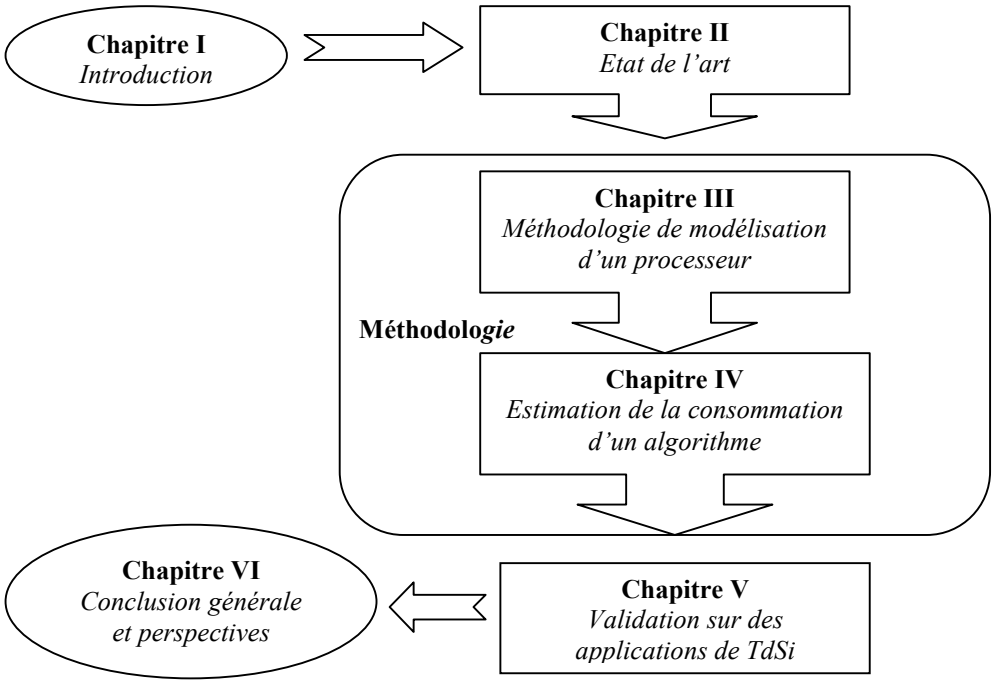

Figure I. 3 Plan du mémoire

Le chapitre II présentera un état de l'art non exhaustif sur les méthodes d'estimation de la consommation existantes pour les processeurs. Il présentera également les outils pouvant être utilisés pour réaliser ces estimations. Pour chacune des méthodes et des outils présentés, nous en donnerons les points forts et les points faibles.

La méthodologie de modélisation d'un processeur ainsi que la méthode d'estimation de la consommation d'un algorithme seront présentées dans les troisième et quatrième chapitres.

Le cinquième chapitre présentera les résultats obtenus sur des applications de traitement du signal et de l'image. Ces résultats nous permettront de valider notre modèle de puissance ainsi que la méthodologie de modélisation.

Enfin, le dernier chapitre permettra de conclure cette étude et de dégager des perspectives de recherche.

Chapitre II

Etat de l'art

La première partie de ce chapitre expliquera pourquoi la consommation est devenue un paramètre important dans la conception des systèmes. Nous présenterons également les différentes sources de consommation afin de montrer que leurs causes et leurs conséquences sont diverses.

La deuxième partie de ce chapitre expliquera les différentes techniques d'estimations de la consommation existantes. Ces techniques ne s'appliquent pas toutes au même niveau d'abstraction ni au même type de cible technologique. Nous expliquerons seulement les méthodes se rapprochant le plus de nos objectifs puisque toutes ces méthodes ne sont pas applicables dans le contexte que nous nous sommes fixé.

Enfin la dernière partie de ce chapitre montrera quelles sont les possibilités et les limites de ces méthodes ; la méthodologie développée pendant cette thèse sera alors introduite.

II.1. La consommation

Le terme consommation regroupe, en fait, les dissipations de puissance et d'énergie. Dans cette thèse, nous les distinguerons à chaque fois puisque ces deux paramètres ont leur importance lors de l'étude de consommation d'un système.

La consommation de puissance est un paramètre important si l'on s'intéresse à la dissipation thermique dans un système. En effet, pour pouvoir dimensionner les circuits de refroidissement, il faut connaître avec précision la puissance dissipée par ce système (surtout pour les applications embarquées où le système de refroidissement requiert une surface, une masse et un coût non négligeables).

L'énergie est un paramètre à prendre en compte si l'on veut étudier la durée de vie des batteries. Ce paramètre est bien sûr lié à la puissance consommée par l'application mais, également au temps d'exécution. Deux applications peuvent avoir la même puissance dissipée mais, avoir des consommations d'énergie différentes comme l'illustre l'exemple pédagogique du Tableau II.1.

APPLICATION	PUISSANCE	TEMPS D'EXECUTION	ENERGIE
ALGO-1	2.5W	10µs	*25µJ*
ALGO-2	2.5W	25µs	*62.5µJ*

Tableau II. 1 Consommation de puissance et d'énergie

Dans ce tableau, nous pouvons voir que l'algorithme 1 et l'algorithme 2 consomment exactement la même puissance ; nous aurions donc tendance à dire que ces algorithmes sont identiques du point de vue de la consommation. Or, les temps d'exécution de ces algorithmes sont sensiblement différents puisque le premier est exécuté en 10µs et que le second l'est en 25µs. Cette différence de temps d'exécution aura une répercussion sur l'énergie consommée par l'algorithme puisque le deuxième algorithme consomme 2,5 fois plus d'énergie que le premier à puissance égale.

Ce petit exemple pédagogique nous montre bien que pour les applications embarquées, la consommation de puissance est un paramètre nécessaire mais pas suffisant puisque la durée de vie des batteries dépend de l'énergie consommée.

II.1.1 Qu'est ce qui consomme ?

La rapide évolution des technologies CMOS vers le submicronique profond (<0,18µm) génère une augmentation de plus en plus importante du nombre de transistors sur une même puce. Cet accroissement du nombre de transistors sur une puce entraîne une augmentation de la consommation de puissance. La puissance consommée par un circuit est composée de 2 termes, un terme représentant la puissance statique et un autre la puissance dynamique.

La puissance statique est due aux courant de fuite des transistors (courant généré entre le substrat et la grille du transistor). Jusqu'à présent, la consommation statique des transistors était négligée puisque les technologies CMOS étaient supérieures à 0,13µm. Or, depuis cette année les fondeurs atteignent des finesses de gravure en production de l'ordre de 0,13µm (ex : Le processeur Pentium 4 d'Intel). Aujourd'hui, il devient donc indispensable de prendre en compte cette consommation statique ; elle devrait même devenir prépondérante dans les années à venir du fait de la diminution de la technologie.

La puissance dynamique est, elle, due aux commutations (passage de l'état bloqué à passant et inversement) du transistor. Cette consommation peut être calculée en utilisant la formule suivante :

$$P(W) = \alpha * C_{eq} * F * Vdd^2 \quad (1)$$

Les technologies submicroniques permettent également une diminution des tensions d'alimentation mais, également une augmentation des fréquences de fonctionnement.

Le Tableau II.2 montre l'évolution des technologies CMOS de 1996 à nos jours ainsi que les perspectives d'évolution dans les 10 ans à venir **[Sia01]**.

	1996	1999	2002	2005	2008	2011
Technologie (µm)	0.35	0.18	0.13	0.1	0.07	0.05
Tension d'alimentation (V)	2.5	1.8	1.5	1.2	0.9	0.6
Transistor par cm² (M)	10	20	54	133	328	811
Nb de transistors par circuit (M)	80	160	432	1064	2624	6488
Nb de couche de métal	4-5	5	6	7	8-9	9
Fréquence (MHz)	300	500	700	900	1200	1500

Tableau II. 2 Evolution des caractéristiques d'intégration des ASICs en technologie CMOS

La puissance dynamique est directement liée au taux de commutation du circuit α, à la capacité équivalente du circuit $\mathbf{C_{eq}}$, à sa fréquence de fonctionnement **F** ainsi qu'à sa tension d'alimentation $\mathbf{V_{dd}}$ au carré. L'augmentation de la finesse de gravure devrait donc entraîner une diminution de la capacité équivalente du circuit ainsi qu'une diminution de sa tension d'alimentation. En pratique, la capacité équivalente des circuits n'a pas tendance à diminuer mais plutôt à augmenter car le nombre de transistors augmente avec une pente plus forte que ne décroît la capacité des transistors. Les fréquences de fonctionnement, elles aussi augmentent avec la diminution de la taille des transistors (ex : Le processeur Pentium 4 d'Intel fonctionne à une fréquence de 2GHz) ; ceci entraîne donc une augmentation de la consommation des circuits. Les tensions d'alimentation diminuent avec l'augmentation de la finesse de gravure. Ce paramètre intervenant au carré dans la formule (1) de la puissance dynamique, l'effet de la diminution des tensions d'alimentation a donc un impact important. Toutefois comme le montre le Tableau II.3, cette diminution n'est pas assez importante pour contre balancer l'augmentation de la consommation due aux autres paramètres.

Le tableau II.3 montre l'évolution de la consommation de puissance de processeurs généraux des familles d'Intel et d'AMD. Ce tableau est issu d'une étude réalisé dans **[Bur01]**.

Processeur	Technologie (µm)	Nb transistors (10^6)	Vdd (V)	Fréquence (MHZ)	Puissance crête (W)
INTEL					
386SX	1	0.275	5	33	2
486DX	0.8	1.2	5	50	5
P5	0.8	3.1	5	66	16
P6	0.35	5.5	3.3	166	29.4
AMD					
K5	0.35	4.3	3.5	120	14
K6	0.35	8.8	3.2	233	28.3
Athlon	0.25	22	1.6	700	50

Tableau II. 3 Evolution de la consommation des familles de processeurs INTEL et AMD

Le tableau ci-dessus montre bien que malgré la diminution de la technologie et de la tension d'alimentation la puissance des processeurs ne cesse de croître.

II.1.2 Pourquoi la consommation est-elle un paramètre important lors de la conception ?

Comme nous avons pu le voir dans le paragraphe précédent, la consommation de puissance ne cesse de croître du fait de l'augmentation continue du nombre de transistors sur les puces. Cette augmentation n'est pas seulement due à la diminution de la technologie mais aussi aux attentes des utilisateurs. En effet, les applications électroniques grand public ne cessent d'augmenter. Les puces électroniques font aujourd'hui parties de notre quotidien, tous les biens de consommation courants utilisent désormais ces puces. Elles sont partout, de notre machine à laver à notre voiture en passant par nos téléphones portables et nos ordinateurs.

Nous pourrions croire que les aspects consommation ne sont importants que pour les applications embarquées (satellites, missiles etc..) ou pour les applications portables (GSM, PDA etc…) ; il n'en est rien. Même si pour beaucoup d'applications, il est possible de recharger les batteries ou d'avoir une alimentation de type prise de courant, la consommation reste un point critique puisqu'elle engendre une dissipation thermique qui n'est pas toujours facile à évacuer. Le coût des systèmes de refroidissement est non négligeable et il est important d'avoir le meilleur produit possible tout en ayant un coût de revient ou de vente acceptable. Le coût de ces systèmes n'est pas le seul paramètre à prendre en compte. En effet, leur taille ainsi que leur poids engendrent des problèmes de dimensionnement pour les applications portables.

De plus, la dissipation thermique a pour effet un vieillissement prématuré des composants. En effet, une simple élévation de la température de 10°C entraîne une diminution par deux de la fiabilité du circuit **[Rab96]**. Or, il n'est pas souhaitable d'avoir un appareil n'ayant pas une durée de vie suffisante si l'on veut pénétrer le marché qui est devenu très concurrentiel.

Les utilisateurs veulent que de plus en plus de fonctionnalités soient intégrées dans les produits. Aujourd'hui, par exemple, il n'est pas envisageable d'avoir un téléphone portable se contentant seulement d'être un téléphone. Il faut qu'il puisse être utilisé comme assistant personnel (PDA), messagerie électronique, console de jeux etc… Avec l'apparition de la 3ème génération (UMTS), il sera même possible de

visionner un film, surfer sur Internet et bien d'autres choses encore. Le fait de rajouter toutes ses fonctionnalités entraîne une croissance de la complexité des traitements à effectuer. Les applications sont de plus en plus « gourmandes » en ressources que ce soit en ressources de calculs ou de mémoires. Pour réaliser ces nouvelles applications, les constructeurs doivent intégrer de plus en plus de transistors sur les puces.

Un autre argument, en faveur de la prise en compte du facteur consommation, est le fait que la capacité des batteries évoluent beaucoup plus lentement (x2.5 watt-heure/kg en 30 ans) que le besoin de puissance des applications (x4 tous les 3 ans) **[Sen02]**.

Au lieu d'avoir, comme il y a quelques années, un composant sur une puce nous avons maintenant un système entier sur une puce (SoC) ; il est donc important de pouvoir optimiser la consommation (de puissance et d'énergie) de tels systèmes. Pour cela, il devient indispensable de pouvoir estimer ces consommations de plus en plus tôt dans le flot de conception. Il faut également que ces estimations soient rapides et précises ; c'est pourquoi depuis quelques années le nombre de travaux sur l'estimation de consommation n'a cessé d'augmenter. Mais, la grande difficulté actuelle est d'estimer la consommation de façon simple et rapide. En effet, les systèmes devenant plus complexes, il est aussi plus dur d'estimer leurs consommations.

II.1.3 Positionnement des méthodes d'estimation

Pour réaliser une application, différents choix technologiques s'offrent au concepteur. En effet, il peut utiliser une solution de type sur mesure (ASIC full custom), de type pré caractérisé (FPGA), de type processeur (général ou spécialisé) ou de type composant virtuel (VC ou IP) etc... En fonction de son choix, le modèle de consommation de la cible ne sera pas élaboré au même niveau d'abstraction puisque les informations disponibles seront plus ou moins complètes. Plus le niveau d'abstraction de la cible sera bas (c'est à dire proche du niveau transistor) et plus la précision du modèle sera importante mais, plus le temps de modélisation augmentera (cf. Figure II.1)

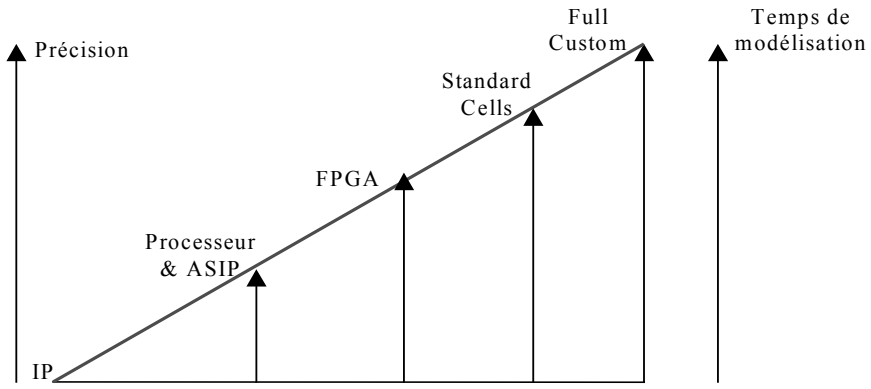

Figure II. 1 Rapport temps de modélisation/précision en fonction de la cible

Comme nous l'avons présenté dans le chapitre d'introduction, notre étude se fera dans le cadre de la conception de systèmes à base de processeurs et/ou de processeurs spécialisés. En effet, le concepteur peut utiliser 3 choix technologiques : ASIC, FPGA ou composants du commerce. Pour chacun de ces choix, de nombreuses méthodes d'estimation existent mais, il nous paraît difficile de développer une méthodologie de modélisation capable de s'appliquer à tous ces choix.

Par exemple, si nous prenons une solution sur mesure(ASIC) pour réaliser une application, nous aurons alors la possibilité de connaître avec précision la technologie employée ainsi que tous les détails architecturaux. Il sera donc possible, dans ce cas, d'utiliser une méthode d'estimation bas niveau qui utilise la capacité équivalente du circuit ainsi que le taux de commutation pour estimer la consommation (Cf. équation 1). Ce type de méthode est utilisé par des outils constructeurs comme Design Power de Synopsys. Le principal inconvénient de cette méthode est le temps nécessaire à l'estimation. En effet, sur un processeur VLIW d'ordre 4 (Lx), Bona et al **[Bona02]** ont montré qu'il fallait 25 minutes (Sun Ultra Sparc 450MHz et 1Go de mémoire) pour estimer la consommation d'un programme simple avec Design Power.

En revanche, si nous décidons d'utiliser une solution de type composant commercial, les caractéristiques technologiques (capacité équivalente, taux de commutation…) nous seront inconnues. Seules les caractéristiques architecturales (nombre d'étages de pipeline, nombre d'unités de traitement…) seront, dans ce cas, utilisables. Nous devrons alors augmenter le niveau d'abstraction du modèle pour pouvoir caractériser le comportement du circuit en consommation. Le modèle de

consommation sera dans ce cas soit établi sur les spécifications de l'architecture soit sur des mesures physiques.

Pour un même choix technologique, il s'avère que cinq niveaux d'abstraction sont possibles pour la modélisation de son comportement en consommation :

- Au niveau système
- Au niveau architectural
- Au niveau fonctionnel
- Au niveau micro architecture
- Au niveau portes.

Dans le paragraphe suivant, nous expliquerons les différentes méthodes d'estimation de la consommation qui existent pour des cibles de type processeurs et processeurs spécialisés. Nous détaillerons ces méthodes en fonction du niveau d'abstraction du modèle de consommation ; nous expliquerons comment est déterminé le modèle et nous donnerons les possibilités et les limites de chacune de ces méthodes.

II.2. Les méthodes d'estimations pour les processeurs

De nombreuses équipes de recherche se sont penchées sur les méthodes d'estimation de la consommation. Pour les processeurs, plusieurs méthodes d'estimation de la consommation ont été développées. Ces méthodes s'appliquent à 3 niveaux d'abstraction, soit au niveau instructions, soit au niveau fonctionnel, soit enfin au niveau système.

II.2.1 Méthodes au niveau instructions

Ce niveau d'abstraction est actuellement le plus utilisé pour les méthodes d'estimation de la consommation. Elles sont toutes basées sur une méthode appelée Analyse de la Puissance au niveau Instructions (ILPA). L'**ILPA** a été développé à l'Université de Princeton par Vivek Tiwari et al **[Tiw96b]**. Cette méthode est souvent considérée comme référence dans l'estimation de la consommation. Elle est applicable théoriquement à tous processeurs que ce soit les processeurs généraux (Pentium, Athlon,…) ou les processeurs spécifiques (DSP…). Nous verrons toutefois que cette

méthode limite généralement l'estimation au niveau du processeur et que souvent les interactions avec l'environnement extérieur ne sont pas prises en compte.

a. Méthode ILPA :[Tiw96] [Tiw94][Lee97]

L'ILPA est basée sur le fait que tout code exécuté sur une cible engendre des commutations de transistors et donc une consommation de puissance. Sachant qu'un code (programme) est constitué d'une suite d'instructions, il suffit de connaître la consommation de puissance ou d'énergie de chaque instruction pour pouvoir estimer la consommation du code.

L'utilisateur construit donc un modèle de consommation de sa cible en mesurant le courant de chacune des instructions du jeu. Le protocole de mesure de la consommation est relativement simple. Pour chaque instruction, un petit programme en assembleur (scénario) est créé ; dans ce scénario, cette seule instruction est répétée un grand nombre de fois et ce scénario doit être exécuté en boucle pour permettre une mesure du courant moyen consommé. Le principe de mesure est présenté sur la Figure II.2.

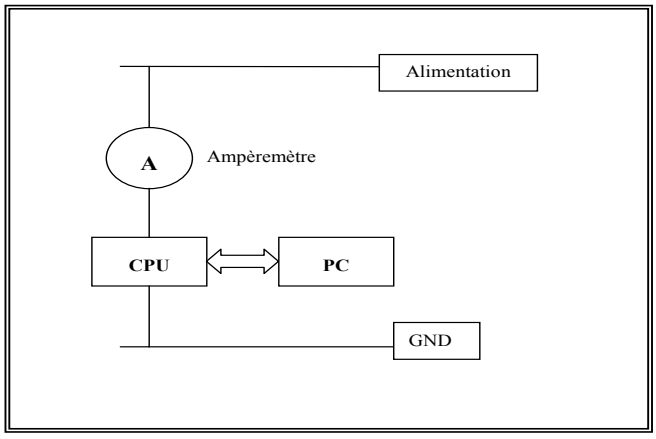

Figure II. 2 Principe de mesure de la consommation

Pour des architectures utilisant un pipeline peu profond, Tiwari et al ont déterminé que le nombre d'instructions dans la boucle devait être de 200 pour limiter l'effet dû au branchement effectué en fin de boucle **[Tiw94]**.

Ils ont également montré que pour estimer de façon plus précise la consommation d'un programme, il fallait prendre en compte la consommation due aux inter-instructions. En effet, une consommation supplémentaire est engendrée par l'interaction entre deux instructions successives. Par exemple, si un code exécute une

instruction d'addition suivie d'une instruction de multiplication alors, la consommation de ce code ne sera pas exactement la somme des consommations de ces deux instructions. En effet, il apparaît un surcoût dû au passage d'une instruction à l'autre i.e. qu'un certain nombre de bits commutent du fait du changement du code de l'instruction. Le modèle doit donc tenir compte de cette consommation inter-instruction ce qui oblige de mesurer les consommations de toutes les combinaisons deux à deux possibles.

Leur modèle prend également en compte l'ajout de consommation due aux ruptures de pipeline. En effet, des dépendances de ressources (conflits d'accès aux bancs, dépendance d'unités fonctionnelles…) génèrent une augmentation du temps d'exécution du programme donc une augmentation de l'énergie consommée. Ils ont donc intégré ce paramètre à leur modèle en mesurant le coût moyen pour chaque famille de ruptures.

Un dernier paramètre doit être pris en compte dans le modèle : c'est l'interaction du processeur avec son environnement extérieur (défauts de cache). En effet, le taux de défaut de cache entraîne une pénalité temporelle mais également un surcoût de consommation dû au remplacement de l'instruction dans le cache. Ce paramètre a été intégré en mesurant le coût temporel d'un cycle de défaut de cache et en le multipliant par son coût moyen en consommation de puissance.

Une fois tous ces paramètres déterminés, le modèle de consommation d'énergie du processeur est alors créé. L'estimation d'un code peut alors être effectuée en utilisant la formule 2 :

$$\textit{Energie programme} = (\Sigma Ei * n) + Eii + Erupture + Edc \quad (2)$$

Ei : Energie de l'instruction i.

n : Nombre d'exécution de l'instruction i.

Eii : Energie des inter-instructions.

Erupture : Energie due aux ruptures de pipeline.

Edc : Energie due aux défauts de cache.

Cette méthode d'estimation permet d'obtenir des estimations précises et rapides. En revanche, la création du modèle de consommation d'une cible est longue et fastidieuse du fait du nombre de mesures à réaliser.

Par exemple, considérons un processeur ayant un jeu d'instructions composé de 30 instructions. Pour réaliser le modèle de consommation de cette cible, il faudra

réaliser 900 mesures pour modéliser la cible (N^2 mesures où N représente le nombre d'instructions du jeu). Cette méthode peut être appliquée sur des cibles relativement simples (jeu d'instructions réduit) mais devient « irréalisable » pour les cibles actuelles qui sont de plus en plus complexes. En effet, le nombre de mesures à réaliser sur des architectures superscalaires ou VLIW (Very Long Instruction Word) devient prohibitif. Si nous reprenons l'exemple précédent (30 instructions dans le jeu) et que nous prenons une architecture VLIW d'ordre 2 alors, le nombre de mesures à réaliser sera de 810000 [$O(N^{2*k})$ où k représente l'ordre du VLIW].

D'autres méthodes, basée sur l'ILPA, ont été développées soit pour réduire le temps nécessaire à l'élaboration du modèle de consommation du processeur soit pour modéliser des cœurs de processeurs. Quelques-unes unes d'entre elles sont présentées ci-dessous.

b. Méthode ILPA modifiée 1 [Chak99]

La méthode développée par Chakrabarti et al de l'Université de Tempe reprend l'idée de réaliser un modèle de consommation au niveau instruction. La différence fondamentale, entre leur méthode et celle développée par Tiwari et al, est le niveau d'abstraction auquel elle est réalisée. En effet, Chakrabarti et al créent leur modèle en utilisant une description au niveau porte (ou au moins au niveau RTL). Ce niveau d'abstraction leur permet d'utiliser les outils commerciaux ou propriétaires pour déterminer le modèle de consommation de puissance de la cible.

La première étape de la méthode est la synthèse de la description comportementale de la cible pour obtenir une description RTL de celle-ci (outil Matisse de Motorola). Ensuite la description RTL est synthétisée pour obtenir une description au niveau porte (outil Synopsys) qui permettra l'utilisation d'un outil d'estimation de la consommation à ce niveau (Outil ASPEN de Motorola).

La deuxième étape est la modélisation du coût en consommation de chaque instruction du jeu. Pour cela, les instructions sont exécutées 1000 fois et une moyenne du coût de consommation est calculée. Les effets des inter-instructions, des taux de défauts de cache, des accès mémoires ainsi que les effets des ruptures de pipeline ne sont pour l'instant pas pris en compte.

Cette méthode a été appliquée sur un microcontrôleur 68HC11 de Motorola. Ils ont réalisé l'estimation de quelques programmes simples et ont comparé leurs résultats

avec ceux fournis par ASPEN. L'erreur maximale entre leurs estimations et celles fournies par l'outil est de **12%** pour une application de tri de 5 nombres. La comparaison de temps nécessaire pour la création du modèle en utilisant leur méthode et celle développée par Tiwari et al en utilisant des mesures physiques n'a pas été réalisée. Nous pouvons supposer que les différentes synthèses devant être effectuées pour obtenir le modèle requièrent un temps non négligeable. De plus, leurs résultats ne sont pas comparés à des mesures physiques mais à d'autres estimations réalisées par l'outil utilisé pour créer leur modèle. Il se pourrait que leur taux d'erreur soit encore plus important que ce qui est donné dans leur étude.

c. Méthode ILPA modifiée 2 [Klass98]

La méthode développée par Klass et al. de l'Université de Carnegie Mellon reprend les grandes lignes de celle de Tiwari et al mais, elle est appliquée sur un cœur de processeur (Motorola 56K). Les mesures de consommation du jeu d'instructions ne sont plus réalisées avec un ampèremètre mais avec l'outil d'analyse au niveau portes Mynoch **[Pur96]**. La nouveauté de cette approche réside dans le fait qu'elle diminue la complexité de la table d'énergie des instructions. En effet, pour créer le modèle au niveau instructions, nous avons vu qu'il fallait déterminer la consommation de chaque instruction du jeu. Or, ce problème a une complexité en $O(N^2)$ ce qui rend ce modèle beaucoup trop long à développer.

Leur méthode permet de réduire la complexité de ce modèle de $O(N^2)$ en $O(N)$. Pour cela, ils ont réalisé 4 modèles différents, un premier modèle appelé « Base Model », un deuxième « Pair Model », un troisième « Nop Model » et enfin un dernier « General Model ». Les deux premiers modèles développés dans ces travaux sont identiques au modèle de Tiwari et al. Cette méthode d'estimation a été testée sur 4 programmes, un FIR 4 points, un FIR 64 points, un FIR 4 points déroulé et enfin une FFT 256 points.

Base Model : L'énergie de chaque instruction du jeu est mesurée en utilisant un petit programme assembleur. Ce scénario est constitué d'une boucle à l'intérieur de laquelle l'instruction à mesurer est répétée un certain nombre de fois (similaire à l'approche Tiwari) ; à la fin de la boucle un branchement nul est généré pour créer une boucle infinie. L'estimation de la consommation d'un programme est réalisée en faisant la

somme des consommations des instructions. Les résultats obtenus avec ce modèle sur les 4 programmes de test montrent que la ***consommation d'énergie est sous-estimée de 17 à 25%*** selon le programme. Ils expliquent cette erreur par le fait que l'effet des inter-instructions n'est pas pris en compte. Pour résoudre ce problème, ils ont développé un autre modèle appelé « Pair Model ».

Pair Model : Ce modèle rajoute un coût en consommation, pour l'effet des inter-instructions, au coût de base d'une instruction. Pour cela, ils ont réalisé des mesures de consommation pour toutes les inter-instructions. Lors de l'estimation de la consommation d'un code, ils rajoutent ce coût au coût de base d'une instruction. L'erreur d'estimation pour les même programmes que précédemment varie alors de ***1 à 10%***. Le problème majeur de ce modèle est que sa complexité est de nouveau en $O(N^2)$ au lieu d'être en O(N) comme le précédent. Ils ont montré que pour 49 instructions modélisées, le nombre d'entrées de leur table d'énergie est de 1176 (ils considèrent que les effets inter-instructions sont symétriques), ce qui requièrent un temps important de modélisation.

Nop Model : Pour réduire cette complexité, ils ont développé un nouveau modèle appelé « Nop Model ». Ce modèle reprend le « Base Model » mais en rajoutant, à la consommation de l'instruction, la consommation d'un NOP (No Operation) si l'instruction i+1 est différente de l'instruction i. Cette utilisation du NOP permet de réduire la taille et donc la complexité de la table d'énergie de $O(N^2)$ à O(N). Pour ce modèle, la table d'énergie possède environ 50 entrées. L'erreur d'estimation sur les programmes de test, en utilisant ce modèle, ***varie de 1 à 8% selon le programme***. Enfin, un dernier modèle a été développé pour permettre la généralisation du Nop Model à tous les processeurs.

General Model : Ce modèle est similaire au Nop Model mais au lieu de créer une table d'énergie pour toutes les instructions du jeu, une table simple d'énergie est créée pour chaque unité fonctionnelle du processeur. Ils rajoutent pour ce modèle une approche fonctionnelle à l'approche au niveau instruction puisqu'ils ne modélisent plus tout le jeu d'instruction mais chacune des unités de traitement de la cible. Ce modèle est constitué de 4 tables correspondant aux 4 unités fonctionnelles : L'UAL (Unité Arithmétique et

Logique), l'UGA (Unité de Génération des adresses), l'UCC (Unité de Calculs Complexes) et l' «Other ». L'estimation de la consommation d'énergie du code est réalisée en faisant la somme de la consommation de chaque unité fonctionnelle. L'erreur d'estimation avec ce modèle sur les programmes tests est de ***10%***.

Le tableau II.4 reprend les 4 modèles développés par Klass et al et, pour chacun d'eux, donne le nombre d'entrées de la table d'énergie ainsi que la précision de l'estimation.

Modèle	Nombre d'entrées de la table	Erreur d'estimation
Base Model	49	***17 à 25%***
Pair Model	1176	***1 à 10%***
Nop Model	50	***1 à 8%***
General Model	4	***≤10%***

Tableau II. 4 Complexité et précision de la méthode en fonction du modèle

Cette méthode n'a pour l'instant été appliquée que sur une architecture de DSP (Digital Signal Processor) simple et elle n'a été testée que pour des applications de traitement du signal peu complexes. De plus, elle ne prend pas en compte les problèmes liés aux ruptures de pipeline et la consommation due aux mémoires. Même si cette méthode réduit la complexité de $O(N^2)$ en $O(N)$, son application sur des processeurs plus complexes risque fort d'augmenter le temps de développement du modèle. De plus l'erreur d'estimation pour des processeurs plus complexes risque d'être plus importante du fait des paramètres négligés.

d. Méthode ILPA modifiée 3 [Steinke01]

La méthode développée par Steinke et al. à l'Université de Dortmund reprend l'idée générale du modèle ILPA mais raffine ce modèle en tenant compte de la commutation des bits sur les bus d'adresses et de données. La consommation d'énergie d'un code est estimée en faisant la somme de 4 valeurs d'énergie selon l'équation suivante :

$$E_{totale} = E_{cpu\_instr} + E_{cpu\_data} + E_{mem\_instr} + E_{mem\_data} \quad (3)$$

$\mathbf{E_{cpu\_instr}}$: énergie due à l'exécution des instructions dans la CPU.

$\mathbf{E_{cpu\_data}}$: énergie due aux données nécessaires aux calculs.

$\mathbf{E_{mem\_instr}}$: énergie due aux accès à la mémoire de programme.

$\mathbf{E_{mem\_data}}$: énergie due aux accès à la mémoire de donnée.

Chacune des 4 énergies est, elle-même, constituée d'une somme d'énergies. Par exemple, E_{cpu_instr} est composée de la somme des coûts de base des instructions plus la somme du coût des accès aux registres internes plus la somme du coût des valeurs qui sont dans les registres plus un coût lié à la commutation des bits sur le bus d'adresse. Enfin est ajoutée une consommation due aux inter-instructions. Le détail des autres consommations est donné dans **[Steinke01]**.

La détermination de chacun des coûts en énergie est réalisée en utilisant la même méthode que celle utilisée par Tiwari et al. (c'est à dire par mesures physiques). Cette technique de modélisation a été appliquée sur un processeur ARM7. La seule estimation qui a été réalisée portait sur l'exécution d'une séquence de 12 instructions. Pour cette application l'erreur, entre leur estimation et la mesure physique, est de 1,7%.

Dans ce modèle, la consommation des mémoires caches ainsi que la consommation due aux ruptures de pipeline n'ont pas été prises en compte. De plus, l'erreur de cette méthode est faible comme nous avons pu le voir mais aucune application réaliste n'a été estimée. L'élaboration de ce modèle doit être longue et fastidieuse du fait du grand nombre de mesures à réaliser pour caractériser les 20 paramètres par régression linéaire ; un minimum de 640 mesures est nécessaire. Ce modèle est, il est vrai, précis mais aussi très complexe.

e. Méthode ILPA modifiée 4 [Gebotys98][Gebotys99]

La méthode développée par C. Gebotys et al. de l'Université de Waterloo est basée sur une approche de type ILPA et fonctionnelle à grain fin. En effet, le modèle de consommation d'énergie est toujours basé sur un coût par classe d'instructions mais, il intègre également des coûts en consommation dus aux commutations sur les différents bus ainsi que les coûts d'accès aux mémoires.

Dans ce modèle, 3 classes d'instructions ont été définies : la classe multiplication, la classe soustraction et enfin la classe chargement. A ces trois classes se rajoutent 4 paramètres IR, DABUS, PABUS, M représentant la consommation des différentes commutations (cf. Tableau II.5). Ces 4 paramètres sont déterminés par régression linéaire en utilisant les résultats de mesures obtenus pour différents scénarii.

Le modèle de chaque classe d'instructions est obtenu par l'exécution d'une boucle infinie contenant les instructions à modéliser (idem Tiwari) mais, également en utilisant un certain nombre de benchmarks. Ces benchmarks sont des blocs linéaires d'instructions (un BLI est une suite d'instructions n'ayant qu'un seul point d'entrée et un seul point de sortie) répétés un certain nombre de fois via une boucle. Ils utilisent comme entrées des données pseudo aléatoires ainsi que 2 types d'échantillons de voix.

Pour réaliser l'estimation d'un programme, ils ont déterminé 5 modèles d'estimation différents. Chacun de ces modèles utilise un certain nombre de paramètres propres. La totalité de ces paramètres d'estimation est présentée dans le tableau ci-dessous :

VARIABLES	DEFINITION
IR	Moyenne des commutations dans le registre instructions
DABUS	Moyenne des commutations sur le bus d'adresses données
PABUS	Moyenne des commutations sur le bus d'adresses programme
MUL	Nombre moyen de multiplications
SUB	Nombre moyen de soustractions
LOAD	Nombre moyen de chargements
M	Nombre moyen d'accès mémoire

Tableau II. 5 Paramètres du modèle

Les 5 modèles d'estimation sont les suivants :

- **Algo** : Utilise les paramètres SUB et MUL ; son erreur moyenne est de 4.14%.
- **$Instr_3$** : Utilise les paramètres MUL, PABUS, IR et M ; son erreur moyenne est de 2.49%.
- **$Instr_2$** : Utilise les paramètres IR, MUL et PABUS ; son erreur moyenne est de 2.44%.

- **$Instr_1$** : Utilise les paramètres LOAD, PABUS, SUB, IR et DABUS ; son erreur moyenne est de 1.91%.
- **Arch** : Utilise les paramètres a, Data Memory et t ; son erreur moyenne est de 2.91%. Ce dernier modèle n'est généralement pas utilisable par les concepteurs puisqu'il faut connaître les commutations des registres internes (représentées par a et t) ainsi que les commutations de la mémoire de données.

Les résultats donnés ici sont ceux déterminés pour un DSP TMS320C50.

Le modèle d'estimation présenté par Gebotys et al est précis puisque l'erreur d'estimation peut être de l'ordre de 2% pour le modèle $Instr_1$. Des limitations sont toutefois à mettre en exergue puisque les coûts liés aux interactions avec l'environnement extérieur ne sont pas pris en compte. De plus, ces modèles n'ont pas été appliqués sur des algorithmes conséquents puisque les benchmarks utilisés étaient des blocs linéaires d'instructions variant de 60 à 150 instructions.

Dans ce paragraphe, nous avons listé un certain nombre de méthodes d'estimation de la consommation basées sur une approche de type ILPA :

- Méthode ILPA complète.
- Méthode ILPA complète + prise en compte de l'environnement extérieur.
- Méthode ILPA + approche fonctionnelle à grain fin.

Cette liste, bien que non exhaustive nous a permis d'expliquer le fonctionnement des méthodes les plus représentatives. Nous avons pu voir que les modèles de consommation associés à ces méthodes peuvent être relativement longs à déterminer et que leur complexité s'avère généralement importante.

Pour réduire le temps d'obtention du modèle de consommation d'une cible et sa complexité, d'autres équipes ont travaillé uniquement sur des approches fonctionnelles à grain fin. Le paragraphe suivant va donc expliquer un certain nombre de ces méthodes.

II.2.2 Approches fonctionnelles

Les processeurs et les ASIPs devenant de plus en plus complexes, les méthodes au niveau instructions deviennent inapplicables du fait du temps nécessaire à l'obtention du modèle de consommation (puissance ou énergie). Il est donc devenu indispensable de trouver d'autres approches permettant de garder un niveau d'abstraction en adéquation avec le type de cible à modéliser tout en gardant une complexité raisonnable. Plusieurs travaux se sont attelés à cette tâche en accentuant l'approche fonctionnelle (plus de modélisation du jeu d'instructions). Nous verrons dans ce paragraphe quelques-unes unes des méthodes développées.

a. Approche fonctionnelle 1 [Brand00a][Brand00b]

La méthode développée par Brandolese et al. de l'école polytechnique de Milan est basée non plus sur la modélisation complète du jeu d'instruction mais, sur la modélisation de fonctionnalités utilisées lors de l'exécution d'instructions.

La première étape de leur méthode est l'étude des différentes fonctionnalités de la cible. En effet, l'exécution d'une instruction engendre l'utilisation d'un certain nombre de composants matériels.

Prenons un exemple simple pour expliquer ce principe : supposons que nous voulions réaliser une addition entre 2 données et ranger le résultat en mémoire. La première étape du traitement sera le chargement et le décodage de l'instruction à exécuter ; pour cela, les étages Fetch et Decode du pipeline seront utilisés. La deuxième étape sera le chargement des données requises par l'addition pour effectuer le calcul ; pour cela des transferts mémoires ainsi que des écritures dans des registres seront effectués (l'écriture des données dans des registres dépend de la cible). La troisième étape sera l'exécution du calcul à proprement parler ; pour cela une unité de traitement (UAL) sera utilisée. Enfin, il ne reste plus qu'à sauvegarder le résultat en mémoire ; un transfert mémoire sera donc réalisé.

Cet exemple montre bien que plusieurs fonctionnalités sont utilisées lors de l'exécution d'une instruction. Il est donc intéressant de pouvoir modéliser le coût de l'exécution d'une instruction par la somme des coûts engendrés par l'utilisation des fonctionnalités.

Le modèle de consommation, qu'ils ont développé, prend en compte 5 fonctionnalités différentes :

- **Fetch&Decode** : fonctionnalité représentant les opérations de chargement et de décodage des instructions.
- **Arithmetic&Logic** : fonctionnalité représentant la réalisation d'opérations logiques et/ou arithmétiques.
- **Write Register** : fonctionnalité représentant les opérations d'écritures en registre.
- **Load&Store** : fonctionnalité représentant la réalisation d'opérations load, store et pile.
- **Branch** : fonctionnalité représentant les opérations de sauts et d'appels aux procédures.

Le coût en consommation de chaque fonctionnalité est déterminé en utilisant un jeu d'apprentissage (ou scénario). Ce jeu d'apprentissage est constitué d'un certain nombre d'instructions du processeur. Pour limiter l'erreur due au choix du jeu d'apprentissage, les mesures sont réalisées sur une centaine de jeux différents choisis de façon aléatoire.

L'estimation de la consommation d'énergie d'un code se fait par sommation des différentes consommations : nous avons donc un modèle additif. La précision de ce modèle est de 15 à 20% mais, le temps nécessaire pour réaliser l'estimation est inférieur à la seconde. Pour améliorer la précision d'estimation, ils ont développé deux nouveaux modèles (Intermediate et Accurate), en utilisant un langage intermédiaire appelé VIS (Virtual Instruction Set). Le deuxième modèle rajoute une étape dans laquelle le code VIS est traduit vers l'assembleur utilisé par la cible.

Le langage VIS traduit le code source (par exemple le code C) en code intermédiaire et des classes d'instructions sont générées. Chaque classe d'instructions est définie par son code opérande et par son mode d'adressage mais les valeurs des opérandes sont ignorées. Pour chaque classe, un jeu d'instructions est créé.

Les résultats, sur la même application que précédemment, montrent que l'erreur pour le modèle Intermediate varie de 3 à 6% et 1% pour le modèle Accurate. Le temps d'estimation est de l'ordre de 30 secondes pour le modèle Intermediate et de l'ordre de la minute pour le modèle Accurate.

La méthode d'estimation proposée par Brandolese et al. montre que la précision varie en fonction du modèle utilisé. Plus le modèle est précis et plus le temps d'estimation mais également le temps de modélisation de la cible augmente.

De plus leur modèle n'intègre pas les consommations liées aux interactions avec l'environnement extérieur. Ce modèle n'ayant pas été appliqué sur des cibles architecturales récentes, il est difficile d'en estimer les performances sur ces architectures. Nous pouvons toutefois pressentir que les taux d'erreurs risquent d'augmenter du fait des paramètres non pris en compte par les modèles.

b. Approche fonctionnelle 2 [Sami00a][Sami00b]

La méthode développée par Sami et al de l'école polytechnique de Milan est basée comme précédemment sur une approche fonctionnelle de la cible. Cette méthode a été développée pour permettre, lors de la conception d'un système sur puce (SoC), d'estimer la consommation des différents types de processeurs utilisables dans le système. Elle s'intéresse plus particulièrement aux architectures de type VLIW (Very Long Instruction Word) qui sont plus complexes que les architectures classiques. En effet, ces architectures chargent plusieurs instructions en même temps et sont capables, dans certain cas, de les exécuter simultanément.

Cette méthode utilise un postulat qui est le suivant : la consommation d'énergie associée à une instruction VLIW est égale à la somme des énergies des instructions composant l'instruction VLIW.

Le modèle de consommation d'énergie a été développé pour une architecture VLIW de type LOAD/STORE : toutes les données servant à un calcul, doivent être chargées en registre avant que ce calcul puisse être exécuté. Ce cœur de processeur VLIW est également constitué d'un pipeline à 5 étages :

- L'étage chargement des instructions (IF).
- L'étage décodage des instructions (ID).
- L'étage de lecture des registres (RR).
- L'étage d'exécution (EX).
- L'étage d'écriture des résultats (WB).

L'estimation de consommation d'énergie d'un programme est donc réalisée en utilisant la formule 4 :

$$E(w_n) = \Sigma \{ U_s(w_n|w_{n-1}) + m_s(w_n) * p_s(w_n) * S_s(w_n) \} \quad (4)$$

- U_s $(w_n|w_{n-1})$: représente l'énergie consommée par l'étage s du pipeline durant une exécution idéale de l'instruction w_n après une instruction w_{n-1}.
- m_s : représente le nombre de cycles de latence durant l'exécution de l'instruction w_n.
- p_s : représente la probabilité de latence durant l'exécution de l'instruction w_n.
- S_s : représente l'énergie consommée par l'étage s du pipeline pour chacun de ces cycles de rupture ou de latence.

Les différents temps nécessaires au calcul de l'énergie sont obtenus en utilisant un simulateur cycle par cycle d'architecture VLIW (Trimaran **[Crest01]**) alors que les mesures de consommations sont, elles, obtenues par estimation en utilisant l'outil Design Power de Synopys.

Cette méthode a été appliquée sur une architecture VLIW simplifiée décrite en VHDL (Very High Description Language), synthétisée à l'aide l'outil Synopsys Design Compiler et implantée sur une technologie 0,35µm 3,3V. Cette architecture n'est capable de réaliser que trois types d'opérations : les additions, les multiplications et les NOPs (No Operation). Ce processeur possède également un cache d'instructions à correspondance directe. Pour tous les tests réalisés, aucun défaut de cache n'est généré i.e. que l'écriture des instructions dans le cache n'est pas modélisée. Les résultats montrent que le modèle de consommation est précis dans le cas où aucune rupture de pipeline n'est générée (au maximum 1,34% d'erreur). En revanche, si des ruptures de pipeline sont générées par l'exécution du programme la précision du modèle est alors plus faible puisqu'elle est de l'ordre de 12,54%. Cette erreur est donnée pour un programme exécutant seulement une addition suivie d'une multiplication.

Ce modèle de consommation d'énergie est très intéressant puisqu'il essaie de modéliser le comportement d'une architecture VLIW ainsi que l'influence de l'environnement extérieur au niveau fonctionnel plutôt qu'au niveau instructions. Les premiers résultats sont encourageants même si ceux-ci ont été obtenus sur une cible VLIW simple.

Le temps de modélisation n'a pas été comparé au temps nécessaire à l'obtention d'un modèle de niveau instruction mais nous pouvons toutefois supposer que ce temps

est plus court puisqu'un modèle instructions demanderais N^{2k} mesures (N :nombre d'instructions du jeu, k :nombre d'instructions pouvant être en parallèle).

Ce modèle montre tout de même quelques limitations puisque l'erreur d'estimation dépasse les 12% même sur un VLIW simple. Cette erreur est due en grande partie à la modélisation du coût des ruptures de pipeline. De plus, la réelle utilisation d'un cache d'instruction n'a pas été testée puisque aucun test n'a généré des défauts de cache. La dernière limitation vient de la comparaison entre l'estimation et la mesure puisqu'ils utilisent un autre outil d'estimation pour valider leurs estimations au lieu d'utiliser des mesures physiques. Il est donc probable que l'erreur d'estimation soit encore plus grande que celle déterminée à l'aide de Design Power même si cet outil peut être considéré comme référence.

c. Approche fonctionnelle 3 [Ben01][Bona02]

La méthode développée par Benini et al de l'Université de Bologne a pour ambition d'estimer la consommation d'un système basé sur une architecture de type cœur de processeur spécifique VLIW. Le modèle doit donc prendre en compte la consommation du cœur du processeur mais également celle des mémoires (programme et données) ainsi que celle des files de registres. Pour cela une méthode d'estimation à deux niveaux a été développée.

Le premier niveau est basé sur des macro modèles caractérisés soit par une analyse au niveau porte pour les modules synthétisés soit au niveau transistor pour les modules sur mesures (mémoires caches, files de registres). Cette approche est basée sur les travaux de E. Macii et al. **[Macii98].**

Le deuxième est basé sur un simulateur du jeu d'instructions (Instruction Set Simulator) de la cible. L'ISS interprète le programme exécutable par simulation et analyse l'effet de chaque instruction sur les composants du système. Ce modèle permet d'obtenir une estimation moyenne de la consommation de puissance du programme mais aussi, d'obtenir la puissance instantanée consommée. L'erreur moyenne de ce modèle est d'environ 5% si l'ISS est en adéquation avec le modèle RTL.

Pour limiter la complexité du modèle au niveau instruction, ils reprennent la méthode développée par Sami et al **[Sami00b]**. L'énergie moyenne par instruction est calculée à l'aide de la formule 5 :

$$E(w_n) = B + (\alpha_n * c_{syl}) + (m * p * S) + (l * q * M) \quad (5)$$

- B : coût moyen en énergie.
- α_n : nombre d'instructions différent d'un NOP dans un mot d'instruction VLIW.
- C_{syl} : consommation moyenne d'énergie associée à l'exécution d'une instruction.

Le troisième terme représente la consommation d'énergie due à un défaut de cache de données.

- m : nombre moyen de cycles de ruptures par mot d'instruction VLIW dû aux défauts de cache de données.
- p : probabilité par mot d'instruction VLIW qu'un défaut de cache de données survienne.
- S : énergie consommée par le cœur lors d'une rupture de pipeline.

Le quatrième terme représente la consommation d'énergie due à un défaut de cache d'instructions.

- l : nombre moyen d'instruction NOP par mot d'instruction VLIW introduit lors d'un défaut de cache instructions.
- q : probabilité, par mot d'instruction VLIW, qu'un défaut de cache instruction soit généré après l'exécution d'une instruction.
- M : énergie consommée par le cœur durant un défaut de cache instruction.

Les deux derniers termes de l'équation 5 permettent d'estimer la consommation due à certains aléas possibles dans les processeurs RISC (Reduce Instruction Set Computer).

Benini et al. ont appliqué cette méthode sur une architecture paramétrable de type Lx développée par ST MicroElectronics et Hewlett-Packard. Pour leur étude, cette architecture était constituée d'un seul cluster capable d'exécuter au maximum 4 instructions en parallèle. Ce cluster possède 4 unités arithmétiques et logiques 32 bits, 2 multiplieurs 16*32 bits, une unité de chargement/enregistrement ainsi qu'une unité de branchement.

L'estimateur au niveau RTL a été validé sur un jeu de tests (filtre FIR, DCT rapide, DCT inverse rapide et DCT/IDCT) et comparé avec un outil de simulation au

niveau porte. L'erreur maximum est de 10% pour l'estimation de la consommation d'énergie du cœur.

L'estimateur au niveau instructions a été validé sur le même jeu de tests ; l'erreur maximale pour ce modèle est de 7,9% par rapport à celle obtenue avec le modèle RTL. L'estimateur au niveau instruction est donc moins précis que celui au niveau RTL mais il est plus rapide puisqu'il est capable de simuler 1,7 millions d'instructions longues par seconde contre 160 par seconde pour le modèle RTL.

Cette méthode est très intéressante dans le cas où le développeur voudrait estimer la consommation d'énergie d'un système comprenant un cœur de processeur (spécialisé ou non) ainsi que des mémoires caches. L'estimation est réalisée rapidement si le modèle au niveau instruction est utilisé (quelques centaines de secondes pour une application complexe) et l'erreur d'estimation est relativement faible (maximum 8%) par rapport à un outil d'estimation au niveau portes.

Toutefois, les applications traitées sont pour l'instant relativement simples puisqu'il s'agit de transformées en cosinus discret et de filtres à réponse impulsionnelle finie. Il aurait été intéressant d'avoir les résultats d'estimation pour des applications plus conséquentes. Le problème de la détermination des probabilités de taux de défauts de cache instructions et de données n'a pas non plus été abordé.

Nous avons vu dans ce paragraphe que des approches fonctionnelles commençaient depuis quelques années à apparaître. Ces approches permettent de réduire le temps d'obtention du modèle de consommation d'énergie ou de puissance d'une cible architecturale par rapport aux approches de type ILPA. Le principal inconvénient de ces méthodes est la difficulté à prendre en compte tous les paramètres de consommation par une simple vue fonctionnelle. De plus, les taux d'erreur sont généralement plus importants qu'avec une approche de type ILPA.

Dans le paragraphe suivant, nous allons expliquer une nouvelle approche. Cette approche a pour but de modéliser le comportement en consommation directement au niveau système.

II.2.3 Approche système

Aujourd'hui, de plus en plus d'applications sont décrites directement au niveau système puisque les concepteurs sont capables d'intégrer une application complète sur une puce (Soc). Le temps de développement de ces applications étant très important du fait de leur complexité, il est intéressant de pouvoir estimer la consommation au plus tôt dans le flot de conception. Une méthode d'estimation au niveau système permettrait donc de réduire le temps de développement.

Une première approche a été réalisée par Sinha et al. de l'Institut de Technologie du Massachusetts, partant du constat que les approches de type ILPA ne sont plus du tout applicables sur les architectures actuelles du fait du grand nombre d'instructions. De plus, suite à une étude de consommation qu'ils ont réalisée sur un processeur StrongARM d'Intel ainsi que sur un processeur SH4 d'Hitachi, ils se sont aperçus que le courant consommé par les instructions de la cible ne variait que très peu (8% de variation) **[Sinha01]**. En revanche, nous savons que le courant moyen consommé est fonction de la fréquence d'horloge et de la tension d'alimentation I=f(F,Vdd). A partir de ces constatations, un modèle simple de consommation d'énergie a été développé, ce modèle s'appelle modèle d'ordre 1.

a. Modèle d'ordre 1

Ce premier modèle ne prend en compte que la tension d'alimentation de la cible et sa fréquence de fonctionnement. L'énergie consommée est donc calculée en utilisant la formule 6 :

$$E_{tot} = V_{dd} * I_0 (V_{dd}, f) * \Delta t \quad (6)$$

- V_{dd} : représente la tension d'alimentation de la cible.
- I_0 : représente le courant moyen consommé par une instruction pour une fréquence et une tension d'alimentation données.
- Δt : représente le temps d'exécution du programme.

L'hypothèse qui est faite dans ce modèle, c'est que la puissance est constante quelle que soit l'application pour une fréquence et une tension d'alimentation données (le programme n'est pas pris en compte).

L'obtention de ce modèle est très rapide puisqu'il suffit de mesurer le courant moyen consommé par une instruction pour les valeurs de fréquence et de tension supportées par la cible (à chaque tension d'alimentation correspond une et une seule valeur de fréquence). L'estimation de la consommation d'énergie est alors très simple puisqu'il suffit de choisir la bonne valeur de I_0, correspondant à la tension d'alimentation spécifiée, et de faire un profiling du code pour obtenir la valeur du temps d'exécution.

Ils ont testé ce modèle sur un StrongARM et sur un jeu de benchmarks (FFT, FIR, DCT…). Les résultats obtenus montrent que l'erreur maximale est de 8% par rapport aux mesures. Ils se sont toutefois aperçus que l'erreur d'estimation due à l'approximation que toutes les instructions consomment la même énergie, était plus importante pour les cibles de type processeur de traitement du signal (DSP) ; donc que leur modèle d'ordre 1 n'était pas adapté. Pour pouvoir estimer la consommation de tels processeurs, ils préconisent alors de reprendre une approche de type ILPA modifiée. Il faut alors regrouper le jeu d'instructions en plusieurs classes et mesurer la consommation de chacune des classes ainsi que la consommation inter-classe.

La méthode développée par Sinha et al. montre qu'un modèle simple, prenant en compte que la tension d'alimentation et la fréquence de fonctionnement, peut donner des résultats relativement précis pour des architectures simples. Dès lors que l'architecture devient plus complexe, cette approche n'est plus intéressante puisqu'elle génère des erreurs d'estimations importantes.

II.3 Bilan des méthodes pour les processeurs

Comme nous avons pu le voir dans la partie précédente, de nombreuses équipes de recherche travaillent sur le problème de l'estimation de consommation pour processeurs. Les modèles de consommation des processeurs peuvent être obtenus à divers niveaux d'abstraction mais les méthodes d'estimations sont souvent similaires puisque, dans presque tous les cas, la consommation d'énergie ou de puissance d'un code est obtenue par sommation de la consommation de chaque instruction de ce code.

De plus, beaucoup de modèles sont basés sur une approche de type ILPA (Instruction Level Power Analysis) et ne diffèrent que très peu les uns des autres.

Dans cette partie, nous allons tout d'abord faire le bilan des méthodes existantes et présenter pour certaines méthodes les outils automatiques associés. Dans un deuxième paragraphe, nous introduirons notre méthodologie de modélisation des processeurs ainsi que notre méthode d'estimation associée.

II.3.1 Bilan des méthodes existantes

Les méthodes d'estimation existantes sont pour la plupart basées sur une approche de type ILPA développée par Tiwari et al. Nous avons vu précédemment que l'inconvénient majeur de cette méthode était le nombre de mesures à effectuer pour modéliser la consommation de la cible. Il est donc difficile de modéliser les architectures complexes à l'aide de cette méthode. Pour palier ce problème, plusieurs équipes de recherche ont fait évoluer cette méthode dans le but de diminuer le nombre de mesures à réaliser. Les tendances d'évolution actuelles sont les suivantes :

- Regrouper les instructions en classes. Les mesures sont réalisées pour chacune des classes et non plus pour chacune des instructions.
- Ajouter à l'ILPA une approche fonctionnelle à grain fin.
- Ajouter la prise en compte des influences externes (aléas).

Plusieurs travaux ont également montré qu'il était possible de remplacer les mesures de consommation par des estimations réalisées à l'aide d'outils de synthèse **[Chak99, Klass98]** ce qui permet de modéliser des cœurs de processeurs et non plus seulement des processeurs commerciaux.

D'autres équipes se sont orientées vers des approches uniquement fonctionnelles **[Brand00a, Sami00a, Ben01, Bona02]** afin de diminuer encore plus le temps d'obtention du modèle et de prendre en compte l'environnement extérieur.

Afin de faire la synthèse de toutes les méthodes explicitées dans la partie précédente nous dresserons un tableau récapitulatif dans lequel nous donnerons pour chaque méthode présentée, sa complexité, les applications testées, la cible utilisée, l'erreur d'estimation, l'outil développé ainsi que ses possibilités et ses limites.

Equipe	Méthode	Complexité	Applications	Cible	Erreur	Outil	Avantages	Inconvénients
Université de Princeton	ILPA	$O(N^2)$ à $O(N^{2k})^*$	Séquence d'instructions	Intel 486DX Fujitsu SPARClite	3%		Estimations précises	Complexité du modèle Temps de modélisation
Université de Tempe	ILPA pour cœur de processeur	$O(N)$ à $O(N^k)$	Tri de 5 nombres	Motorola 68HC11 description niveau RTL	12%		Complexité réduite	Temps de modélisation Inter-instructions, cache, ruptures de pipeline non modélisés
Université CMU	ILPA + classes d'instructions	$O(N)$ à $O(N^k)$	FIR, FFT, LMS	Motorola 56K	10%		Complexité réduite	Testée sur architecture et applications simples. Ruptures de pipeline et mémoires non modélisées
Université de Dortmund	ILPA+ consommation des bus	$O(N^2)$ à $O(N^{2k})$	Séquence de 12 instructions	ARM7	1.7%		Mémoires et bus adresses et données modélisés	Testée sur application simple. Défauts de cache et ruptures de pipeline non modélisés
Université de Waterloo	ILPA + fonctionnelle grain fin	$O(N)$ à $O(N^K)$	Benchmark de 150 instructions	DSP Texas Instruments C50	3%		Estimations précises	Défauts de cache et ruptures de pipeline non modélisés
Ecole polytechnique Milan	Fonctionnelle 3 niveaux	?	ILC16	ARM7 68000 486DX	De 1 à 20% selon le niveaux		Possibilité de compromis temps d'estimation vs rapidité	Défauts de cache et ruptures pipeline non modélisés
Ecole polytechnique Milan	Fonctionnelle	?		VLIW simplifié	12.54% si ruptures de pipeline		Complexité réduite	Problème de modélisation du pipeline. Défauts de cache non testés
Université de Bologne	Fonctionnelle 2 niveaux	?	FIR, DCT, IDCT	Lx (cœur paramétrable)	10%		2 niveaux d'estimation Modélisation des mémoires	Comparaison des résultats avec un autre outil d'estimation.
MIT	Système 2 niveaux	?	FIR, FFT, DCT	StrongARM	8% niveau 1 2% niveau 2	JouleTrack	Estimation des courants de fuite	Pas de cible VLIW et/ou pipeline

- N représente le nombre d'instruction du jeu et k le nombre d'instructions composant un mot VLIW

Tableau II. 6 Récapitulatif des méthodes d'estimation existantes.

Le tableau précédent montre bien que les méthodes de type ILPA sont les plus complexes puisqu'il faut caractériser tout le jeu d'instructions de la cible mais, elles offrent la meilleure précision d'estimation. Il est difficilement envisageable d'utiliser ce type de méthode pour des cibles actuelles du fait de leur complexité. Même les méthodes ILPA utilisant des classes d'instructions requièrent un nombre de mesures prohibitif pour des cibles de type VLIW. C'est pourquoi des équipes de recherche se sont penchées sur des approches de type fonctionnel **[Ben01], [Sami00a&b], [Brand00a&b]**.

En ce qui concerne les outils automatiques d'estimations de la consommation, très peu d'entre eux utilisent les méthodes d'estimation vues précédemment. Le schéma ci-dessous présente un certain nombre d'outils d'estimation de la consommation existants. Pour chaque outil, ce schéma donne le niveau d'abstraction du modèle de consommation en abscisse ainsi que son ou ses points d'entrée en ordonnée.

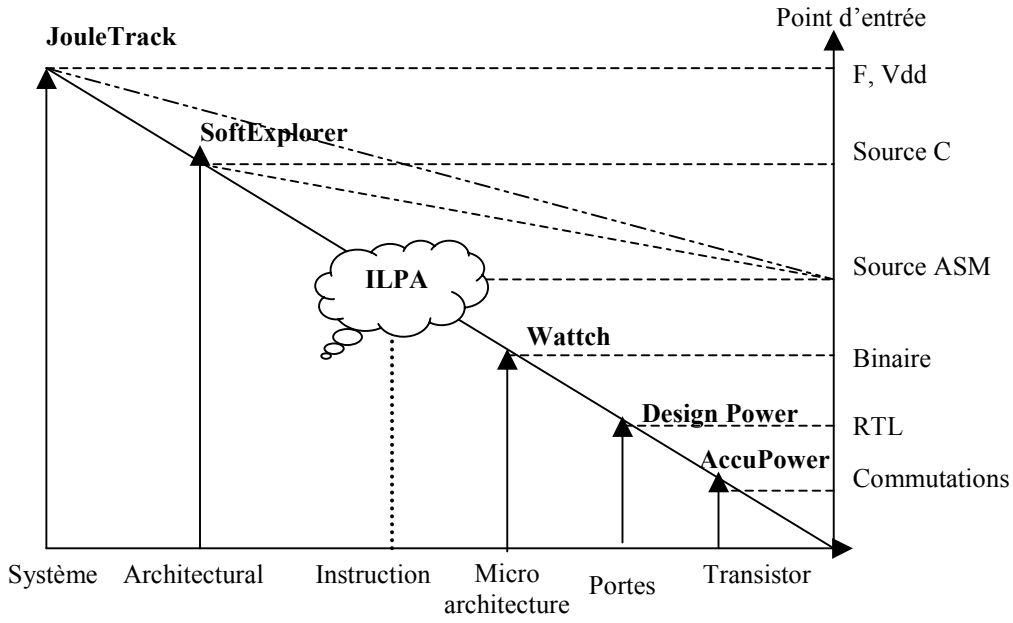

Figure II. 3 Niveau d'abstraction et points d'entrée des outils d'estimation

Les méthodes d'estimation de type ILPA sont les plus nombreuses ; pourtant, nous n'avons trouvé aucun outil automatique utilisant ce niveau d'abstraction. Nous n'avons représenté qu'un seul outil commercial (Design Power de Synopsys) sur cette figure puisqu'ils travaillent tous au même niveau d'abstraction et avec le même point d'entrée. De plus, il nous semble intéressant d'indiquer cet outil car il est souvent pris comme référence par les équipes qui travaillent sur l'estimation de consommation sur

des cœurs de processeurs. Il est toutefois difficile d'exprimer la précision de ces outils car leurs estimations ne sont pas toujours comparées à la même référence (par exemple SoftExplorer est comparé à des mesures physiques et AccuPower est comparé avec SPICE).

Les méthodes d'estimation actuelles montrent toutefois des limites puisqu'elles ne modélisent pas entièrement le comportement en consommation de l'architecture. En effet, dans la plupart des cas la consommation due à l'interaction avec l'environnement extérieure (défauts de cache, accès en mémoire externe etc…) n'est pas prise en compte. Or, les applications actuelles nécessitent de plus en plus des ressources de calculs et de mémoires importantes ce qui entraîne l'utilisation d'architectures complexes (VLIW, superscalaire, pipeline profond) et de mémoires externes. Le temps d'obtention des modèles de consommation de telles architectures par les méthodes « classiques » requièrent un temps très important, sans compter le temps d'exploitation des tables d'énergie. Il devient donc indispensable de développer une méthode d'estimation de la consommation complète, précise et qui ne requiert pas un temps de modélisation trop long.

II.3.2 Présentation de la méthodologie développée dans le cadre de notre étude

La méthodologie que nous proposons répond à quatre objectifs :

- Premièrement, elle est applicable à tous les types de processeurs que ce soit les processeurs généraux (Pentium, PowerPC) ou les processeurs spécifiques (DSP).
- Deuxièmement, sa complexité est réduite par rapport à une approche de type ILPA puisque c'est une approche de type fonctionnelle gros grain (FLPA). En effet, elle est basée sur une approche architecturale et paramétrique.
- Troisièmement, elle est complète puisqu'elle prend en compte aussi bien la consommation des unités de traitement, la consommation des mémoires internes (mémoire programme, données et caches) mais aussi la consommation liée aux aléas et à l'interaction avec l'environnement extérieure.
- Quatrièmement, elle permet d'obtenir des estimations ayant une précision du même ordre que les approches de type ILPA.

De plus, cette méthodologie a été automatisée (outil SoftExplorer) ce qui permet d'obtenir rapidement l'estimation de consommation d'un programme ainsi que des analyses du code permettant de repérer les points « chauds » du programme.

Chapitre III

Méthodologie de modélisation d'un processeur

La première partie de ce chapitre expliquera la méthodologie générale FLPA (Fonctional Level Power Analysis) que nous avons développée pour permettre la modélisation en consommation d'un processeur ; elle est basée sur une approche fonctionnelle et paramétrique. Nous expliquerons les différentes phases de la construction du modèle de consommation sur un exemple pédagogique puis, nous mettrons en application cette méthodologie sur un processeur de traitement du signal (DSP) du commerce : le TMS320C6201 de Texas Instruments. Nous définirons également les différents paramètres algorithmiques et de configuration du modèle.

Dans une seconde partie nous expliquerons la méthodologie mise en place afin de déterminer les lois de consommation mais également, comment nous avons procédé pour réaliser les mesures physiques de consommation sur la cible.

Enfin dans une dernière partie nous présenterons un exemple complet de modèle de puissance pour le DSP TMS320C6201.

III.1 L'analyse de puissance au niveau fonctionnel

Comme nous l'avons vu dans le chapitre II, la modélisation de la consommation d'un processeur est complexe puisque nous devons prendre en compte :

- Le jeu d'instructions
- Les opérandes de calculs et d'adressages
- Les différents aléas pouvant survenir

Des méthodes au niveau instruction permettent d'obtenir des estimations de consommation très précises mais, elles se heurtent à un problème de complexité. En effet, ces méthodes modélisent tout le jeu d'instructions en consommation ainsi que les effets inter-instructions. Pour des architectures « simples », la complexité de tels modèles est en $O(N^2)$ où N représente le nombre d'instructions du jeu. Pour les architectures complexes actuelles (VLIW), cette complexité est en $O(N^{2k})$ où k représente le nombre d'instructions pouvant être exécuté en parallèle. Le temps de modélisation devient alors prohibitif. Les méthodes ILPA utilisant une approche fonctionnelle à grain fin permettent de réduire la complexité du modèle de puissance et donc de réduire son temps d'obtention mais, ce gain n'est pas encore suffisant.

Un autre problème de modélisation se pose aujourd'hui. Comment modéliser les effets en consommation des aléas et des interactions avec l'environnement extérieur? En effet, les dernières générations de processeurs généraux mais également de processeurs spécifiques possèdent des pipelines de plus en plus profonds, des mémoires caches d'instructions et/ou de données ainsi que la possibilité d'exécuter plusieurs instructions en parallèle (architectures VLIW et superscalaire). Les méthodes fonctionnelles à grain fin permettent, en théorie, de modéliser ces différents paramètres mais, comme nous avons pu le voir dans le chapitre précédent, ce n'est pas aisé. En effet, les erreurs d'estimations augmentent grandement si le programme génère des défauts de cache et/ou des ruptures de pipeline. Dans **[Sami00a]**, l'erreur d'estimation atteint 12,54% si des ruptures de pipeline sont générées même sur une architecture VLIW simplifiée.

La méthodologie que nous proposons est également basée sur une approche fonctionnelle (gros grain au lieu de grain fin) pour permettre la modélisation de toutes les sources de consommation mais aussi sur une approche paramétrique ce qui permet de réduire la complexité du modèle d'estimation.

III.1.1 Présentation de la méthodologie

Dans ce paragraphe, nous allons expliquer comment est réalisée l'analyse de puissance au niveau fonctionnelle (FLPA). Cette analyse est générale et peut être appliquée sur tous les types de processeurs. La FLPA est basée sur une étude fonctionnelle et architecturale de la cible du point de vue de la consommation.

La première étape consiste à étudier le schéma bloc de l'architecture pour en comprendre le fonctionnement interne. Nous n'avons pas besoin de détails sur les architectures internes des unités de traitement mais seulement d'un schéma représentant les principales fonctionnalités de la cible.

Pour expliquer concrètement comment est appliquée la FLPA sur une cible, nous utiliserons un exemple pédagogique. La figure ci-dessous représente le schéma général d'une architecture nécessaire à l'étude. Nous appliquerons la FLPA sur cet exemple et nous prendrons pour hypothèse que ce processeur est VLIW.

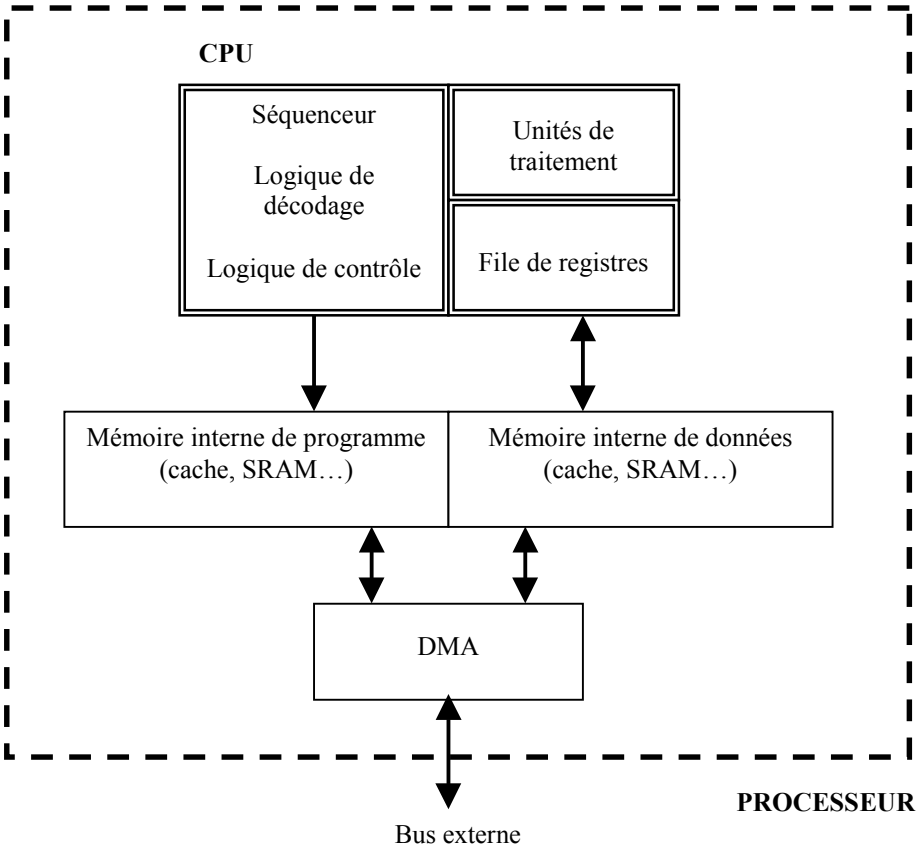

Figure III. 1 Schéma général d'une architecture d'un processeur

Ce genre de schéma bloc est généralement fourni avec la documentation du processeur même pour des processeurs commerciaux ; or, c'est ce type de processeurs que nous avons pour objectif de modéliser.

La première étape de la FLPA consiste à regrouper dans un même bloc fonctionnel les unités architecturales qui sont excitées simultanément lors de l'exécution d'un programme. Cette première étape nous donne donc une première règle de construction.

- **Règle N°1** : Deux unités architecturales appartiennent au même bloc fonctionnel si et seulement si elles sont utilisées simultanément lors de l'exécution d'un code.

Une fois cette règle appliquée, nous obtenons un nouveau schéma fonctionnel de l'architecture. Si nous reprenons l'exemple de la figure III.1, nous avons alors le découpage fonctionnel ci-dessous.

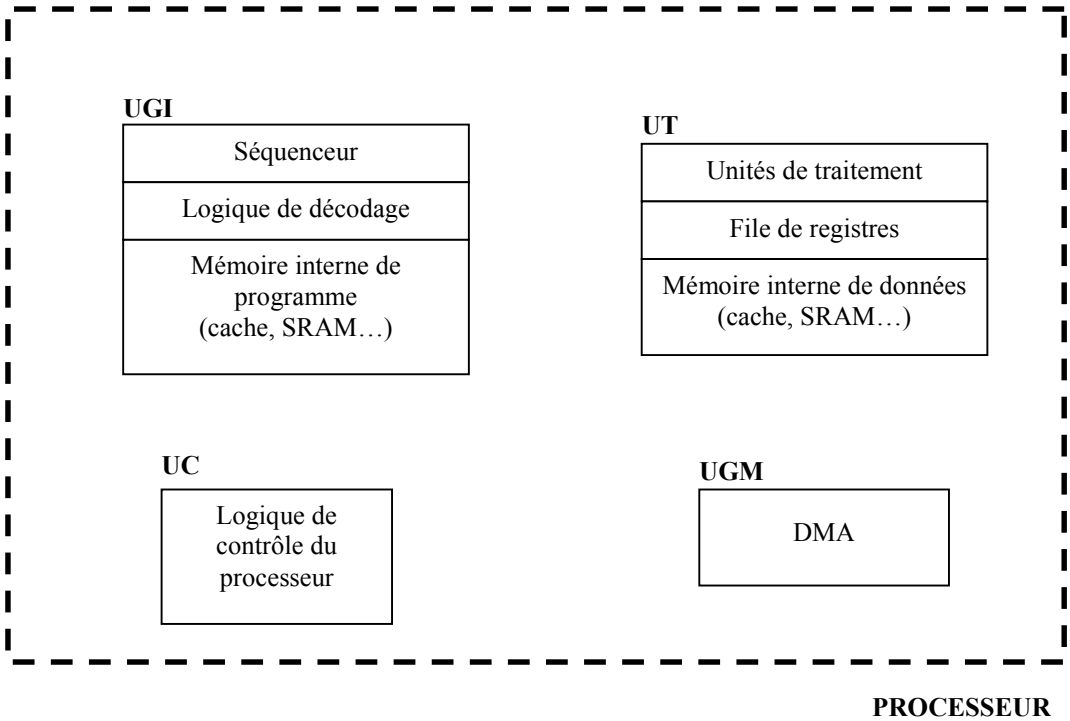

Figure III. 2 Schéma fonctionnel après application de la règle de construction N°1

Nous obtenons donc un découpage fonctionnel général constitué de quatre parties distinctes ; l'Unité de Gestion des Instructions (UGI), l'Unité de Traitements (UT), l'Unité de Gestion Mémoire (UGM) et enfin l'Unité de Contrôle (UC).

- L'UGI regroupe toutes les unités fonctionnelles relatives aux instructions i.e. la mémoire de programme, le séquenceur etc…
- L'UT regroupe toutes les unités de calculs (multiplieurs, Unités Arithmétiques et Logiques etc…), les files de registres ainsi que la mémoire de données.
- L'UC regroupe toutes les unités servant au contrôle des périphériques du processeur ainsi que les registres de configuration.
- L'UGM regroupe toutes les unités fonctionnelles servant aux accès vers la mémoire externe (Direct Memory Access, unités spécialisées d'interfaçage etc…).

La deuxième étape de la FLPA consiste à déterminer si des liens de consommation existent entre les différents blocs fonctionnels. Dans cette étape, nous pouvons également supprimer des blocs ou des sous-blocs fonctionnels si ceux-ci n'ont pas de lien de consommation avec les autres blocs ou si leur contribution dans la consommation globale est faible. Ceci nous donne une deuxième règle de construction.

- **Règle N°2** : Un bloc ou un sous-bloc fonctionnel est supprimé du modèle si et seulement si, il ne possède aucun lien d'un point de vue consommation avec un autre bloc ou sous bloc ou si sa contribution dans la consommation globale est négligeable.

L'application de la règle N°2 entraîne donc une modification du schéma fonctionnel de l'architecture : suppression du bloc fonctionnel Unité de Contrôle (cette suppression n'est pas valable, a priori, dans le cas de microcontrôleurs) et apparition des liens de consommation inter et intra-blocs. Le résultat de ces transformations est donné sur la figure III.3.

Le bloc fonctionnel Unité de Contrôle a été supprimé du modèle puisque sa contribution à la consommation totale de la cible est négligeable pour les applications qui nous intéressent. En effet, ce bloc contient tous les registres de contrôle des périphériques du processeur ainsi que ses registres de contrôle internes. Or, ces registres ne commutent quasiment jamais lors de l'exécution d'un code donc, leur poids sur la consommation globale est négligeable.

Différents liens de consommation intra et inter blocs apparaissent sur le schéma ; ces liens représentent la possibilité, lorsqu'un bloc fonctionnel est excité par une instruction, d'en exciter un autre. Ils sont exprimés sous forme de taux variant de 0 à 1 et sont fonction de l'activité générée par l'exécution du code sur la cible. L'existence de ces différents liens de consommation dépend de l'architecture cible ; **certains d'entre eux peuvent ne pas exister**. Ces liens correspondent aux paramètres algorithmiques de notre modèle.

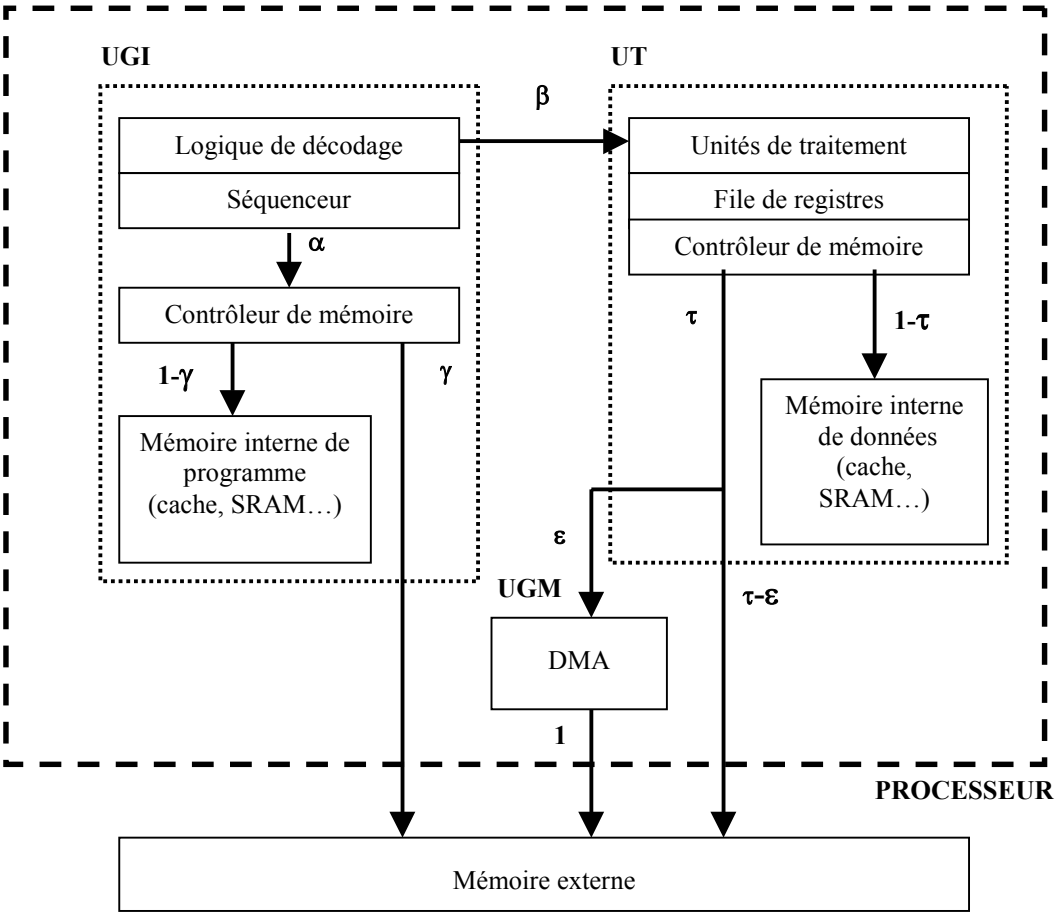

Figure III. 3 schéma fonctionnel après application de la règle de construction N°2

Ces cinq paramètres de consommation sont les suivants :

- α représente le taux de parallélisme, il est lié à l'activité de la mémoire d'instruction donc au parallélisme du programme pour une architecture VLIW ou super scalaire.
- β représente le taux d'unités de traitement utilisées ; il est lié au parallélisme du programme. En effet, plus le degré de parallélisme du code sera grand et

plus la valeur de β le sera. **Quelle que soit l'application exécutée α sera toujours supérieur ou égal à β.**

- γ représente le taux de défaut de cache du programme ; il est lié à la taille du code, à l'ordonnancement des instructions mais également à leur placement.
- τ représente le taux d'accès en mémoire externe pour les données ou le taux de défaut de cache en données si un cache de données existe. Il est lié au placement des données en mémoire mais également à l'ordonnancement des accès aux données.
- ε représente le taux d'accès aux données via le DMA ; il est lié à la configuration faite par le programmeur.

Deux paramètres importants de la consommation n'apparaissent pas dans ce schéma ; il s'agit de l'horloge (F) et de la tension d'alimentation (Vdd). En fait, ces paramètres sont, bien entendu, pris en compte mais ils sont séparés des autres puisque, premièrement, ils s'appliquent à tous les blocs fonctionnels et, deuxièmement, parce qu'ils ne dépendent pas de l'application. En effet, la fréquence d'horloge d'un processeur est générée de façon externe et dépend de la configuration faite par l'utilisateur. La tension d'alimentation, elle, dépend de la technologie employée c'est à dire de la taille des transistors. Ces paramètres sont appelés paramètres de configuration.

Dès lors, la partie découpage fonctionnel de la FLPA est terminée puisque nous avons à présent une représentation fonctionnelle et paramétrique de l'architecture cible.

L'étape suivante est la détermination des paramètres algorithmiques et de configuration pour l'architecture étudiée.

III.1.2 Les paramètres algorithmiques et de configuration

L'utilisation de processeurs pipelinés peut engendrer des dysfonctionnements durant l'exécution d'une application ce qui aura pour conséquence de modifier les valeurs des paramètres algorithmiques. Ces dysfonctionnements sont appelés aléas.

a. Les aléas de fonctionnement

Ces aléas peuvent être classés en trois catégories **[Mar96][Hen96]**:

- Les aléas structurels
- Les aléas de données
- Les aléas de contrôle

Les aléas structurels interviennent lors de conflits de ressources, quand le matériel ne peut gérer toutes les combinaisons possibles de recouvrement des instructions dans le pipeline ; par exemple, lorsqu'un conflit d'accès au bus externe survient. Dans ce cas, une ou plusieurs latences seront générées afin de permettre à toutes les unités de traitement d'obtenir leurs données. Ces aléas sont très importants lors de l'utilisation d'architectures qui ne sont pas Harvard. En effet, il y aura alors des conflits entre les accès mémoire de programme et ceux de données puisque ces accès utilisent le même bus.

Les aléas de données interviennent quand une instruction dépend du résultat d'une instruction précédente, situation que provoque le recouvrement des instructions dans le pipeline.

Les aléas de contrôle résultent de l'exécution en pipeline des branchements et des autres instructions qui modifient le compteur de programme (PC).

b. Les paramètres algorithmiques

Définition : Les paramètres algorithmiques sont des paramètres dont les valeurs évoluent en fonction du code exécuté.

Le taux de parallélisme α et le taux d'unités de traitement utilisées β

Pour expliquer clairement ces deux paramètres, nous allons prendre un exemple pédagogique dans lequel seul ces deux paramètres seront non nuls.

Supposons que notre architecture cible soit une architecture de type VLIW d'ordre 4 i.e. qu'elle est capable dans l'idéal d'exécuter 4 instructions en parallèle. Si nous reprenons le schéma III.3, nous avons donc 5 paramètres algorithmiques pour cette cible. Le découpage fonctionnel devient donc celui de la figure III.4 et les seuls paramètres algorithmiques qui restent sont le taux de parallélisme α et le taux d'unités de traitement utilisées β.

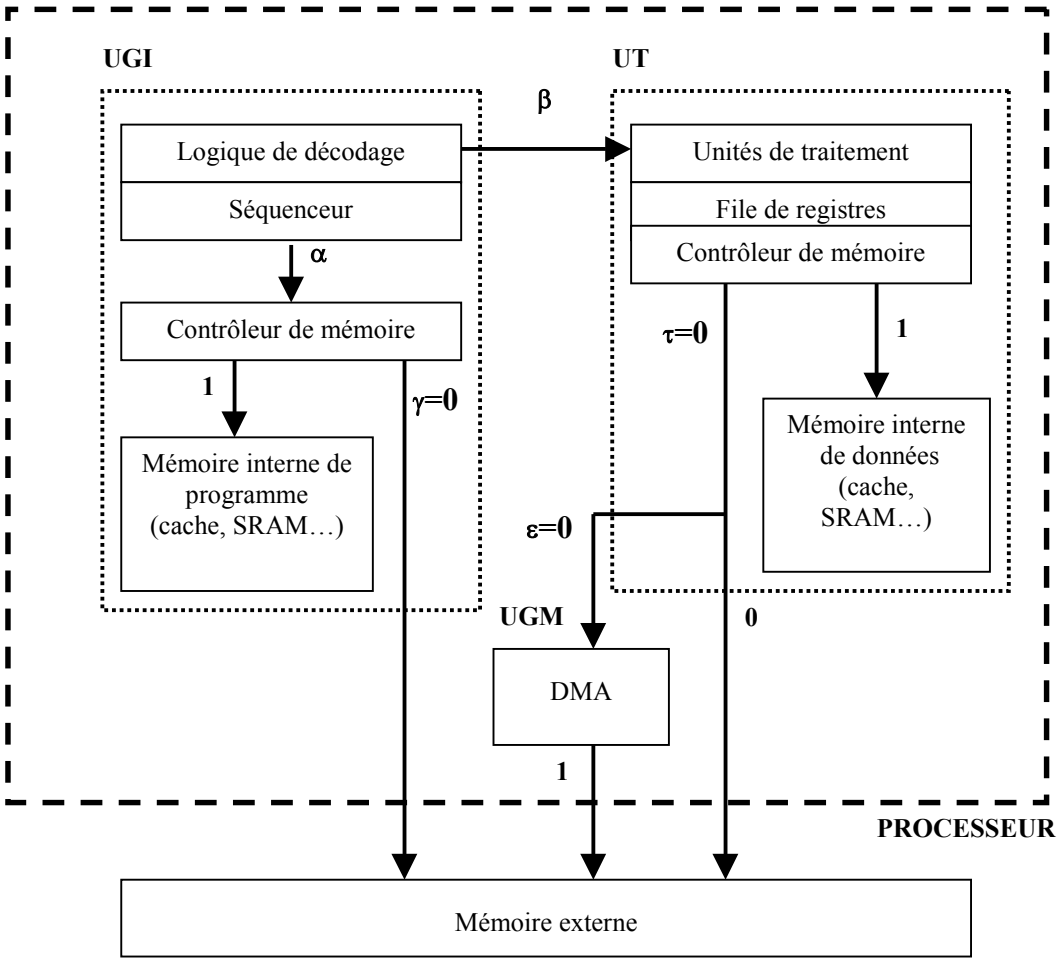

Figure III. 4 Exemple de découpage fonctionnel pour une architecture VLIW d'ordre 4

Supposons maintenant que nous ayons deux codes à exécuter sur cette cible. Le premier utilise toutes les unités fonctionnelles à chaque temps de cycle processeur et que le deuxième ne peut utiliser que 2 unités de traitement en parallèle à cause des dépendances de ressources (encore appelées aléas structurels). De plus, une des instructions du 2ème code réalisera 4 NOPs en parallèle. Dans ce cas aucune unité fonctionnelle ne sera utilisée. L'exécution de ces deux codes va générer des commutations différentes donc des valeurs de α et β également différentes. Pour expliquer clairement ce phénomène, nous allons faire une représentation temporelle du pipeline de la cible (cf. Figures III.5 et III.6).

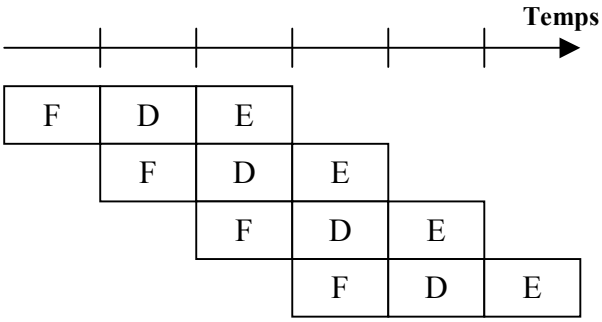

Figure III. 5 Représentation du pipeline pour le code 1

F : étage de chargement des instructions

D : étage de décodage des instructions

E : étage d'exécution des instructions

Si nous considérons l'exécution du code 1 (cf. Figure III.5), nous voyons qu'à chaque temps de cycle processeur un nouveau mot VLIW d'instructions est chargé. Dans ce cas, la valeur du paramètre α (taux de parallélisme) est de 1. De même, nous voyons qu'à chaque temps de cycle un nouveau traitement est effectué (dans notre cas, 4 opérations sont exécutées en parallèle) donc la valeur du paramètre β est également de 1.

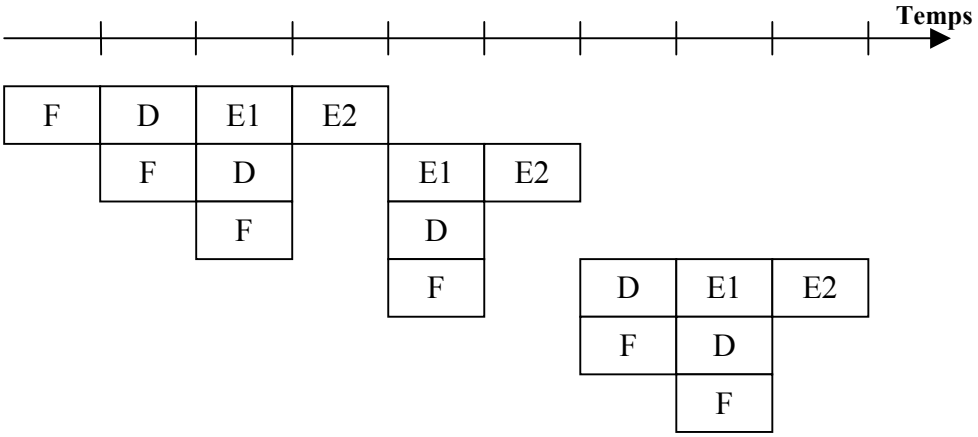

Figure III. 6 Représentation du pipeline pour le code 2

En revanche, si nous considérons l'exécution du code 2 (cf. Figure III.6) alors, nous remarquons que la valeur de α est de 0,75 alors que celle de β est de 0,375.

Nous vérifions donc ici la propriété qui est que **la valeur de β est toujours inférieure ou égale à celle de α**.

Il nous reste maintenant à étudier les paramètres algorithmiques restant i.e. le taux de défaut de cache γ, le taux d'accès aux données en mémoire externe τ ainsi que l'utilisation du DMA pour les accès externes ε.

Le taux de défaut de cache γ

Le taux de défaut de cache est considéré, dans notre modèle, comme un paramètre algorithmique car sa valeur ne dépend pas uniquement de la taille du programme mais également de son exécution.

Supposons que notre processeur possède une mémoire cache de programme à correspondance directe pouvant contenir 100 lignes de code, que notre algorithme soit composé de 150 lignes et qu'il soit itéré deux fois. Dans ce cas, l'exécution du code entraînera 300 lectures d'instructions et génèrera 150 défauts de cache.

Pour calculer le taux de défaut de cache, nous utilisons la formule suivante :

$$\gamma = \frac{NbDéfaut}{NbLecture} \quad (1)$$

NbDéfaut : nombre total de défaut de cache

NbLecture : nombre total de lecture programme

A l'aide de la formule ci-dessus, nous avons déterminé que le taux de défaut de cache de ce programme est de 0,5.

Reprenons maintenant ce même exemple mais supposons que le code s'exécute une première fois complètement et que la seconde itération n'exécute que les 100 premières lignes.

Dans ce cas, l'exécution engendrera 250 lectures d'instructions et génèrera 100 défauts de cache ; la valeur de γ sera alors de 0,25.

En conclusion, le taux de défaut de cache γ peut bien être considéré comme un paramètre algorithmique puisque sa valeur dépend en partie de l'exécution du code.

Le taux d'accès de données en mémoire externe τ

Le quatrième paramètre algorithmique à prendre en compte est le taux d'accès aux données en mémoire externe. Nous avons déterminé que ce paramètre était algorithmique puisque sa valeur dépend du nombre de fois où des données externes sont

accédées durant l'exécution du code. En effet, si l'exécution d'un code dépend des données (branchement conditionnel, nombre d'itérations d'une boucle) qui lui sont fournies alors dans un cas par exemple il devra faire 10% de ses accès en mémoire externe et dans un autre cas aucun accès. La valeur du taux d'accès en mémoire externe sera donc fonction de l'exécution du code. Cette valeur est calculée en utilisant la formule suivante :

$$\tau = \frac{NbAccèsExt}{NbAccèsTotal} \quad (2)$$

NbAccèsExt : nombre d'accès de données en externe

NbAccèsTotal : nombre total d'accès de données

Le taux d'accès aux données via le DMA ε

Le cinquième paramètre algorithmique à prendre en compte est le taux d'accès en données externes via le DMA. Ce paramètre est lié au paramètre τ. La valeur de ce paramètre est calculée en utilisant la formule suivante :

$$\varepsilon = \frac{NbAccèsDMA}{NbAccèsExt} \quad (3)$$

NbAccèsDMA : nombre d'accès via le DMA

NbAccèsExt : nombre d'accès de données en externe

A présent, tous les paramètres algorithmiques définis par la FLPA ont été étudiés et caractérisés. Il reste néanmoins un paramètre algorithmique qui n'a pas encore été pris en compte lors de l'application de la FLPA, ce paramètre est le taux de rupture de pipeline. Nous avons vu dans le chapitre état de l'art qu'une des limitations des approches fonctionnelles à grain fin était la modélisation des ruptures de pipeline. En effet, Sami et al **[Sami00a&b]** ont montré que l'erreur d'estimation pouvait être **supérieure à 10%** lorsque des ruptures de pipeline étaient générées même sur une architecture VLIW simple.

Le taux de rupture de pipeline PSR

Avant de pouvoir caractériser les ruptures de pipeline, il faut étudier la ou les causes de ces ruptures. En fait, les ruptures de pipeline sont dues aux aléas qui sont provoqués par l'exécution d'un programme sur la cible (cf. III.1.2.a). Le taux de rupture

de pipeline PSR (Pipeline Stall Rate) sera donc bien un paramètre algorithmique mais dépendra aussi de l'architecture de la cible (aléas structurels) et du compilateur. En effet, les aléas de contrôle et de données dépendent des transformations réalisées sur le code par le compilateur (déroulage de boucles etc…)

Le nombre de ruptures de pipeline est donc calculé en faisant la somme de toutes les contributions temporelles des différents aléas. Pour cela, nous utilisons la formule ci-dessous :

$$Nb\_RP = Nb\_RP\_Struct + Nb\_RP\_Données + Nb\_RP\_Contrôle \quad (4)$$

Nb_RP : nombre total de ruptures de pipeline
Nb_RP_Struct : nombre total de ruptures de pipeline dû aux aléas structurels
Nb_RP_Données : nombre total de ruptures de pipeline dû aux aléas de données
Nb_RP_Contrôle : nombre total de ruptures de pipeline dû aux aléas de contrôle

Après avoir calculé le nombre total de ruptures de pipeline, nous sommes capables de déterminer le taux de ruptures de pipeline PSR en utilisant la formule ci-dessous :

$$PSR = \frac{Nb\_RP}{Nb\_Cycle\_Exécution} \quad (5)$$

Nb_Cycle_Exécution : nombre total de cycles d'exécution du programme

A présent, tous les paramètres algorithmiques ont été caractérisés. Nous avons dénombré 6 paramètres algorithmiques permettant de modéliser le comportement global de la cible que ce soit en fonction du taux de défaut de cache, des ruptures de pipeline, du taux de parallélisme ou des accès en mémoire externe pour les données.

Dans le paragraphe suivant, nous allons étudier l'autre type de paramètres : les paramètres de configuration.

c. Les paramètres de configuration

Le comportement en consommation d'un processeur dépend aussi d'un certain nombre de paramètres dit de configuration. Parmi ces paramètres, nous retrouvons par exemple, la fréquence d'horloge, le mode mémoire… Nous allons lister de façon générale tous les paramètres de configuration qui peuvent exister.

La fréquence d'horloge F

Cette fréquence a un impact non négligeable sur la consommation globale puisque la consommation dynamique a pour paramètre la fréquence de fonctionnement.

$$P_{Dyn} = \alpha * C_{eq} * F * V_{dd}^2$$

La fréquence d'horloge est générée de façon externe au processeur. En revanche, le processeur possède souvent un système de PLL (Phase Loop Lock) couplé généralement à un multiplieur de fréquence ce qui permet d'obtenir une fréquence réglable. Plus la fréquence d'horloge sera élevée et plus la consommation sera importante. Il faut donc absolument que ce paramètre soit intégré dans notre modèle.

Le mode mémoire MM

Certains processeurs peuvent avoir différents modes de fonctionnement pour leur mémoire interne de programme et/ou de données. Cette propriété se retrouve essentiellement sur les processeurs de traitement du signal (DSP). Ceux-ci peuvent par exemple utiliser leur mémoire de programme en cache ou en SRAM (Static Random Access Memory). Or, nous savons qu'une mémoire cache consommera plus de puissance qu'une simple mémoire SRAM du fait de l'utilisation d'un contrôleur de cache ; ce contrôleur compare l'étiquette de l'instruction à exécuter avec les étiquettes des instructions déjà présentes dans son cache. Cette comparaison entraîne l'utilisation de logique supplémentaire ce qui engendre une augmentation de la consommation. De plus, lors d'un défaut de cache il faut écrire l'instruction qui a généré le défaut dans le cache ce qui augmente encore la consommation par rapport à l'utilisation d'une SRAM.

Le placement des données en mémoire DM

Le placement des données en mémoire a un impact non négligeable sur le temps d'exécution du programme ainsi que sur la consommation **[Rab96]**. L'accès à des données via la mémoire externe engendre une consommation supplémentaire du fait de l'excitation des broches d'entrées/sorties et de la consommation de cette mémoire. Le placement des données en mémoire interne est donc à privilégier.

Lorsque les données sont placées en mémoire interne, il faut quand même porter une attention particulière à leur organisation. En effet, si le code doit accéder à deux données pour réaliser un calcul et que ces deux données se trouvent dans le même banc

mémoire alors un aléa structurel est généré. Cet aléa provoque une rupture dans le pipeline et donc une modification de la consommation.

Le placement des données en mémoire est donc un paramètre à intégrer à notre modèle non seulement pour caractériser le placement en mémoire interne ou externe mais également le placement dans les bancs de la mémoire interne. En effet, actuellement le placement des données en mémoire est entièrement laissé au choix du programmeur et non réalisé durant la phase de compilation.

La mise en veille PM

De nombreux processeurs récents autorisent une mise en veille totale ou partielle de leur architecture. Le programmeur peut ainsi mettre en veille l'arbre d'horloge, les mémoires internes, les périphériques etc… Pour cela, deux techniques sont généralement utilisées :

- Le « clock gating »
- La distribution commandée de la tension d'alimentation.

Le « clock gating » revient à distribuer l'arbre d'horloge uniquement sur les unités qui sont actives lors de l'exécution du programme. Dans ce cas, la consommation de puissance dynamique est nulle ($P_{dyn} = \alpha*C*F*V_{dd}^2$). En revanche la consommation statique est non nulle puisque les unités sont alimentées. La distribution commandée de la tension d'alimentation permet, elle, de résoudre le problème de la consommation statique mais engendre un problème de volatilité des données lorsqu'elle est appliquée sur des mémoires.

La mise en place de ces fonctionnalités montre bien que la consommation est devenue un paramètre important lors de la conception d'un système.

Nous avons montré que 4 paramètres de configuration étaient à prendre en compte lors de la modélisation du comportement en consommation d'un processeur. Ces paramètres viennent se rajouter aux 6 paramètres algorithmiques définis précédemment. Au total, notre processeur sera donc caractérisé par un modèle utilisant 10 paramètres différents. Ce modèle général peut être représenté sous la forme suivante :

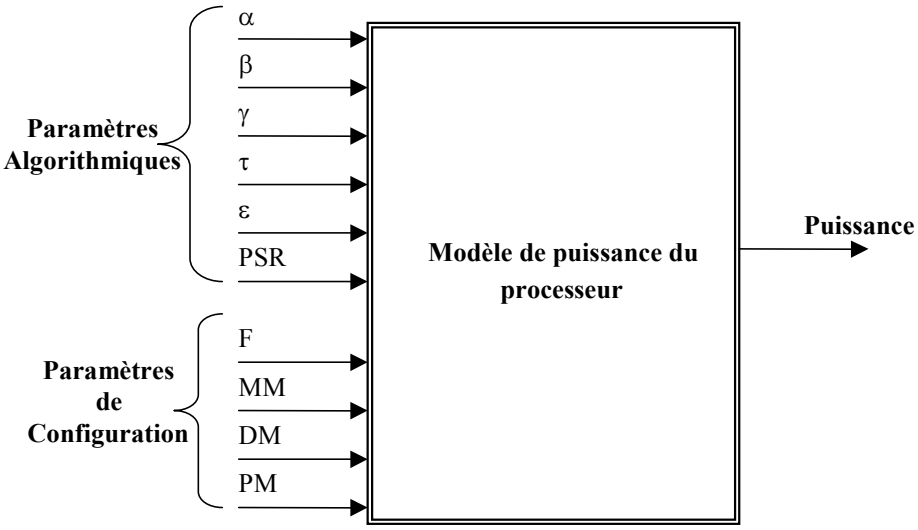

Figure III. 7 Représentation générale du modèle d'un processeur

Le modèle de puissance du processeur sera composé d'un certain nombre de lois de consommation. Ces lois seront des fonctions mathématiques paramétrées avec les différentes entrées du modèle i.e. avec les paramètres algorithmiques et de configuration. L'obtention de ces lois de consommation sera expliquée dans la seconde partie de ce chapitre.

III.1.3 FLPA appliquée au TMS320C6201

Comme exemple d'application, nous allons réaliser l'analyse de puissance au niveau fonctionnel sur un processeur de traitement du signal de Texas Instruments : le TMS320C6201. Ce processeur est un des plus complexes existant sur le marché et fait partie de la dernière génération de DSP commerciaux. Il est capable de réaliser dans l'idéal jusqu'à 1,6 milliards d'opérations par seconde. Ses principales caractéristiques sont les suivantes **[Tex99a]**:

- Tension d'alimentation 1,8 ou 2,5V selon les versions.
- Fréquence d'horloge nominale de 200MHz. Cette fréquence peut être paramétrée par l'utilisateur grâce à une PLL et un multiplieur de fréquence par 1 ou 4.
- Architecture de type Harvard (séparation de la mémoire programme et données) et VLIW d'ordre 8. Il peut donc exécuter dans l'idéal 8 instructions en parallèle.

- Son pipeline est composé au maximum de 11 étages.
- Sa mémoire de programme peut fonctionner suivant 4 modes dont un mode cache.
- Il possède un DMA qui peut être utilisé pour effectuer des transferts de données de l'extérieur vers la mémoire interne de données.
- Il possède également un EMIF (External Memory InterFace) permettant de faire l'interfaçage entre différents types de mémoire externe et le DSP.

Le schéma ci-dessous représente l'architecture du C62 telle qu'elle est fournie à l'utilisateur par Texas Instruments **[Tex99b]**.

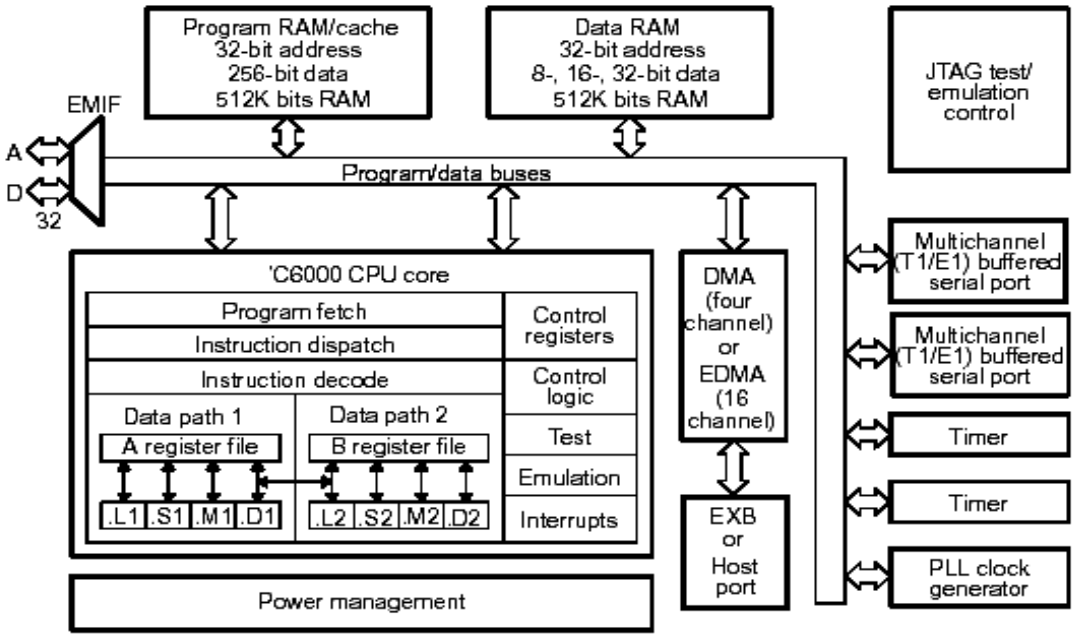

Figure III. 8 Représentation de l'architecture du C6x

Cette représentation schématique est suffisante pour nous permettre de réaliser la première étape de la FLPA. Nous allons donc appliquer la première règle de construction sur cette représentation (cf. paragraphe III.1.1). Après avoir appliqué cette règle, nous obtiendrons donc un premier schéma où les regroupements en blocs fonctionnels auront été réalisés (cf. Figure III.9).

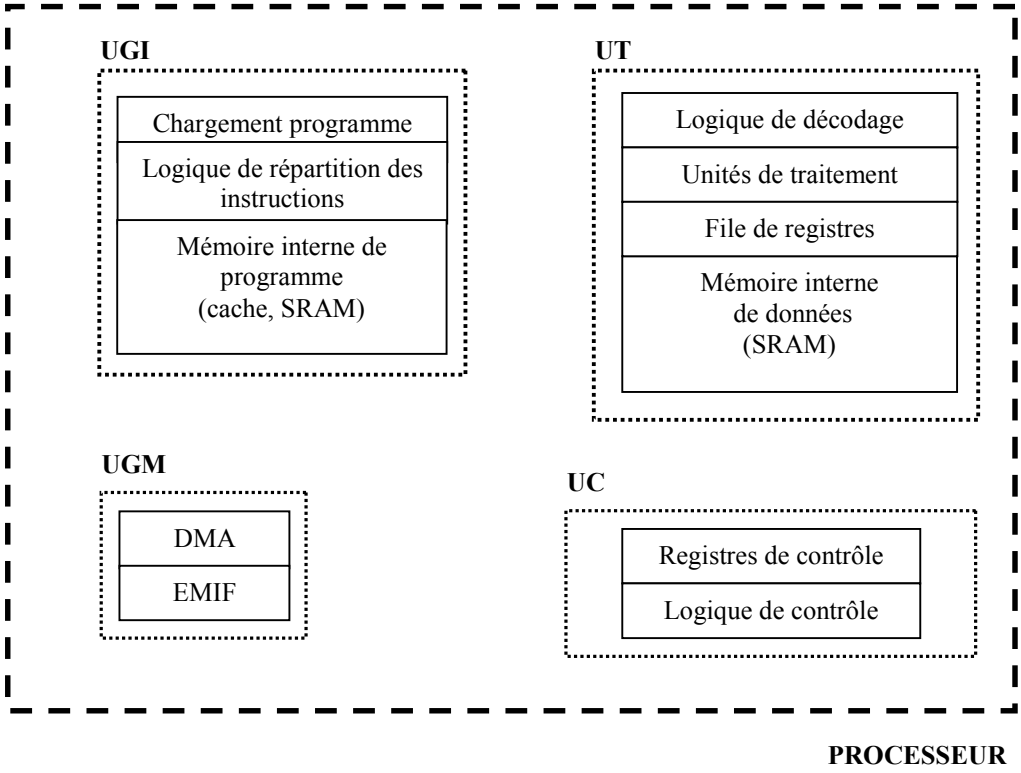

Figure III. 9 FLPA du TMS320C6201 après application de la règle de construction N°1

Ce schéma fait apparaître les différents blocs et sous-blocs fonctionnels issus de l'analyse architecturale du DSP. A présent, nous pouvons appliquer la deuxième règle de construction qui nous permettra de mettre en évidence les différents liens en consommation entre les blocs et les sous-blocs de cette architecture. Cette étape nous permettra également de supprimer certains blocs et/ou sous-blocs si leur impact sur la consommation est négligeable. La figure III.10 représente le schéma fonctionnel final pour le C62.

Sur cette figure, nous voyons que le bloc Unité de Contrôle a été supprimé ; ceci est dû au fait que la logique ainsi que les registres de contrôle engendrent une consommation de puissance négligeable par rapport à la consommation totale. Cette hypothèse de travail a été vérifiée en réalisant une campagne de mesures sur la cible. Nous voyons également que 5 paramètres algorithmiques apparaissent sur ce schéma. Un sixième paramètre algorithmique, le taux de rupture de pipeline (PSR) n'apparaît pas mais il devra toutefois être un des paramètres d'entrée du modèle du DSP. En effet, le pipeline du C62 étant relativement profond (jusqu'à 11 étages) le coût d'une rupture de pipeline est donc loin d'être négligeable.

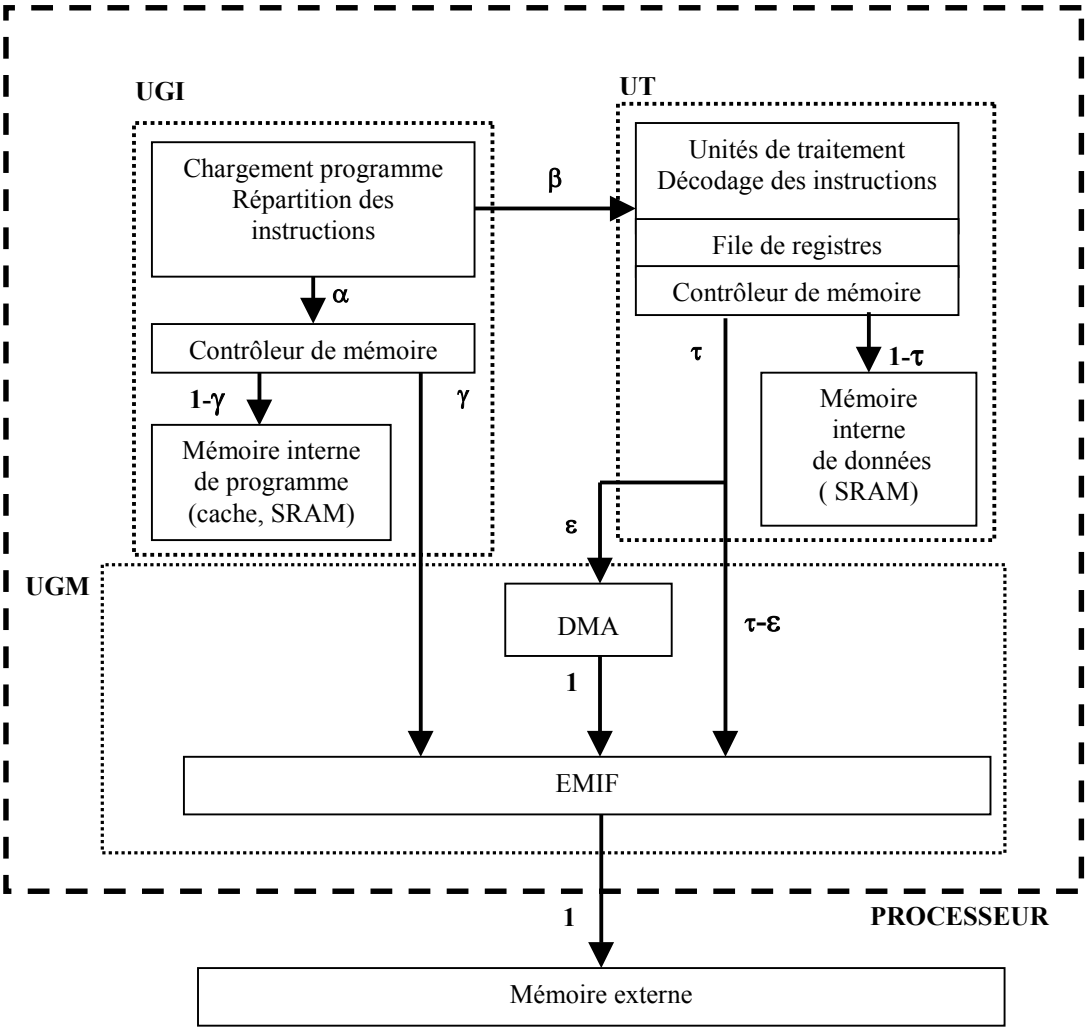

Figure III. 10 FLPA du TMS320C6201 après application de la règle de construction N°2

Pour ce processeur, les 6 paramètres algorithmiques sont les suivants :

- α : taux de parallélisme du programme
- β : taux d'unités de traitement utilisées
- γ : taux de défaut de cache
- τ : taux d'accès de données en externe
- ε : taux d'accès en externe via le DMA
- **PSR** : taux de ruptures de pipeline

Pour compléter le modèle du C62, il faut maintenant étudier son fonctionnement afin de déterminer quels sont les paramètres de configuration à prendre en compte.

Le premier paramètre de configuration est bien entendu la fréquence d'horloge F.
Le deuxième paramètre est le mode mémoire MM. Dans le cas du C62, 4 modes mémoires sont utilisables pour la mémoire de programme interne :

- Le mode **MAPPED (M)** : dans ce mode toutes les instructions sont en mémoire interne de programme donc aucun accès en mémoire externe n'est effectué.
- Le mode **CACHE (C)** : dans ce mode, la mémoire interne de programme est utilisée en mémoire cache à correspondances directes c'est à dire qu'un paquet d'instructions (8 instructions 32 bits) ne peut avoir qu'un seul emplacement physique dans le cache.
- Le mode **FREEZE (F)** : ce mode est similaire au mode cache sauf que lors d'un défaut de cache, le paquet d'instructions ayant généré le défaut n'est pas écrit dans le cache.
- Le mode **BYPASS (B)** : dans ce mode toutes les instructions sont chargées via la mémoire externe.

Le paramètre MM aura donc 4 valeurs possibles **M**, **C**, **F** ou **B** selon le mode choisi par l'utilisateur.

Le troisième paramètre de configuration est le placement des données en mémoire DM. L'utilisateur peut spécifier dans un fichier de commande à quel emplacement mémoire se trouve une donnée ; cette mémoire peut être externe ou interne. Pour la mémoire interne, nous pouvons également spécifier dans quel banc elle se trouve puisque ce processeur possède deux bancs mémoires. En réalité, ces deux bancs sont virtuels ; physiquement il n'y en a qu'un seul. Il faut donc gérer l'offset de banc pour limiter les conflits. Nos expériences nous ont montré qu'un bon placement mémoire peut améliorer de façon significative le temps d'exécution donc l'énergie consommée par une application. Par exemple sur un filtre LMS bi-voies, un placement judicieux des données en mémoire interne permet d'obtenir **un gain de 60%** au niveau du temps d'exécution du programme.

Le paramètre de configuration PM n'est, ici, pas inclus dans le modèle car le réveil du processeur se fait par un reset.

A présent, nous avons tous les paramètres nécessaires à la modélisation en puissance de ce DSP ; le schéma final est alors le suivant :

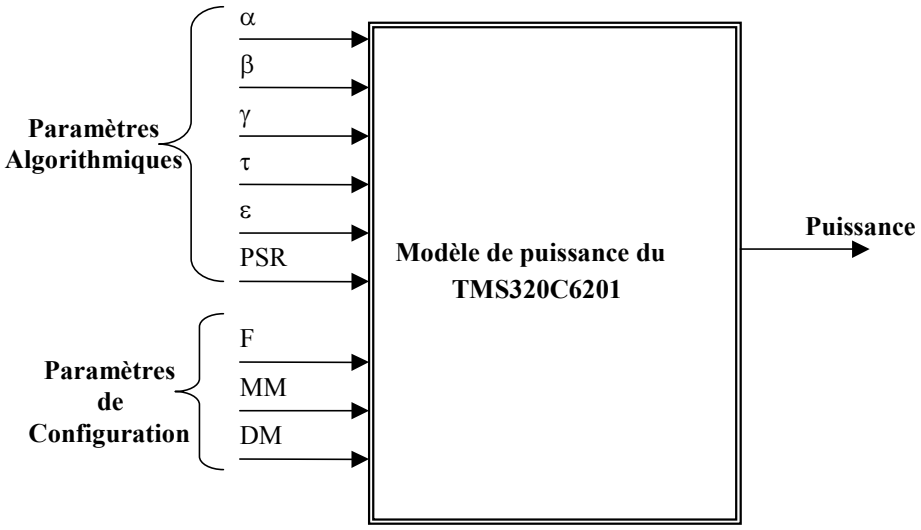

Figure III. 11 Modèle de puissance du TMS320C6201

Dans cette partie, nous avons vu comment l'analyse de puissance au niveau fonctionnel était appliquée. Cette analyse est basée sur une approche fonctionnelle et paramétrique ce qui permet de modéliser des architectures complexes (VLIW d'ordre 8) tout en ayant un modèle de puissance d'une complexité acceptable (9 paramètres). Nous avons donné une représentation générale du modèle généré ainsi que les paramètres généraux à prendre en compte lors de la caractérisation en consommation d'un processeur. Nous avons également montré comment la FLPA pouvait être appliquée sur un DSP TMS320C6201 qui est un processeur complexe.

III.2 Détermination des lois de consommation

Dans cette partie, nous allons expliquer comment les lois de consommation sont obtenues en utilisant les paramètres issus de la FLPA. Nous expliquerons dans un premier temps la méthodologie générale qui est employée puis son application sur le TMS320C6201. Les lois de consommation sont des fonctions mathématiques utilisant les paramètres algorithmiques et de configuration. La détermination de ces lois requiert la réalisation de mesures physiques pour les processeurs commerciaux mais peuvent être remplacées par des simulations au niveau portes pour les cœurs de processeurs

(utilisation d'outils commerciaux : Design Power…). Ces mesures (ou ces simulations) vont nous permettre d'analyser le comportement en consommation de la cible en fonction des variations des paramètres de son modèle de puissance.

III.2.1 Méthode générale d'obtention des lois de consommation

La méthodologie d'obtention des lois de consommation peut être représentée par la figure ci-dessous:

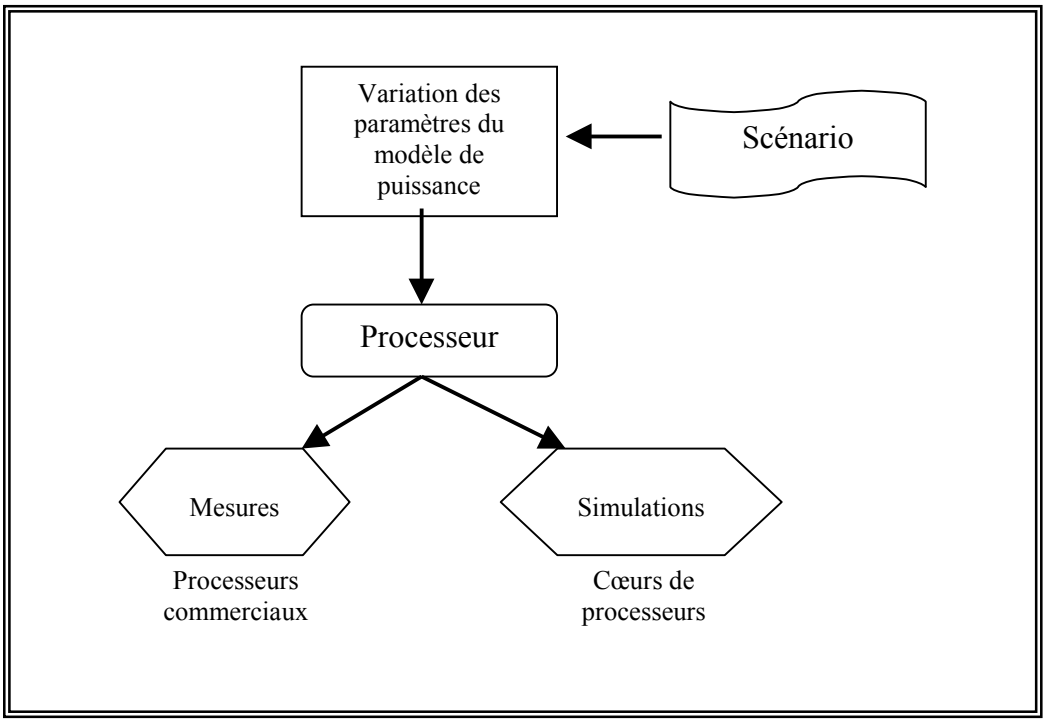

Figure III. 12 Schéma de la méthodologie générale d'obtention des lois de consommation

Quel que soit le type de cible, la variation des paramètres du modèle de puissance est réalisée à l'aide de programmes assembleurs (quelques lignes) appelés scénario. L'utilisation de l'assembleur nous permet de savoir a priori quelles sont les parties du processeur qui seront excitées ; nous pouvons donc faire varier les valeurs des paramètres selon nos besoins.

Chaque scénario est composé d'une ou plusieurs instructions répétées à l'aide d'une boucle ce qui nous permet de réaliser la mesure du courant moyen consommé. En effet, il nous faut une fenêtre temporelle suffisamment grande pour avoir une représentation correcte du courant moyen. La boucle est réalisée par un saut en fin de scénario ; ce saut génère une rupture dans le pipeline et donc une modification de la

consommation. Pour que l'impact de ce saut sur la consommation soit négligeable, nous déroulons en partie ou complètement le scénario.

Les mesures physiques sont réalisées à l'aide d'une sonde de courant (ou d'un ampèremètre) branchée sur l'alimentation du processeur. Cette technique, dont la validité a été démontrée, est similaire à celle employée par Tiwari et al. **[Tiw96b]**. Les mesures ainsi réalisées nous permettent d'obtenir une représentation de la consommation de courant de la cible en fonction des variations des valeurs des paramètres (cf. figure ci-dessous).

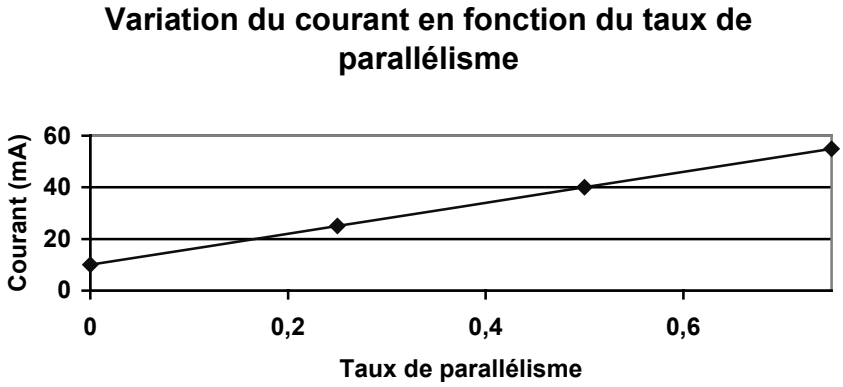

Figure III. 13 Exemple de variation de courant en fonction de la valeur d'un paramètre du modèle de puissance

Une fois que toutes les variations de courant en fonction des paramètres ont été mesurées et mise sous forme de graphiques, il est alors aisé de les modéliser sous forme d'équations mathématiques. Ces fonctions seront les lois de consommation de notre cible.

III.2.3 Application au TMS320C6201

Pour mieux comprendre le principe de la méthode, nous allons montrer comment nous l'avons appliquée sur le DSP TMS320C6201. Nous avons vu précédemment que la FLPA pour ce DSP engendre un modèle ayant 6 paramètres algorithmiques (α, β, γ, τ, ε et PSR) et 3 paramètres de configuration (F, MM, DM).

Dans un premier temps, nous allons expliquer les composantes, le calcul et l'influence du taux de ruptures de pipeline sur les autres paramètres algorithmiques de ce DSP.

Le taux de ruptures de pipeline PSR

Dans ce processeur, le taux de rupture de pipeline permet de regrouper deux paramètres algorithmiques : le taux d'accès de données en externe τ ainsi que le paramètre placement des données en mémoire DM. En effet, ces deux paramètres n'ont pour effet que de générer des ruptures de pipeline donc, ils peuvent être intégrés dans le PSR.

Le principal effet des ruptures de pipeline est d'augmenter le temps d'exécution d'un programme puisqu'elles génèrent des latences supplémentaires. Le PSR pour ce processeur possède trois composantes : les ruptures liées aux défauts de cache, aux conflits de bancs et celles liées aux accès de données en mémoire externe. Le nombre de ruptures de pipeline est donc calculé en utilisant la formule suivante :

$$Nb\_RP = Nb\_CB + (Nb\_DC * Tps_c) + (Nb\_AME * Tps) \quad (7)$$

Nb_RP : nombre total de ruptures de pipeline donné en nombre de cycles processeur.
Nb_CB : nombre de conflits de bancs.
Nb_DC : nombre de défauts de cache.
Tps_c : latence de résolution d'un défaut de cache.
Nb_AME : nombre d'accès de données en mémoire externe.
Tps : latence d'accès en mémoire externe. (exemple pour notre carte Tps est égal à 8 si $F_{DSP} = F_{mémoire}$ ou Tps est égal à 16 si $F_{DSP} > F_{mémoire}$).

Les temps de latence sont fonction du type de mémoire externe utilisée. Ceux qui sont donnés ici sont seulement représentatifs des résultats obtenus pour notre carte. Le taux de rupture de pipeline PSR est déterminé en utilisant la formule suivante (cf. III.1.2.a):

$$PSR = \frac{Nb\_RP}{Nb\_Cycle\_Exécution}$$

Le principal effet des ruptures de pipeline est de diminuer les valeurs intrinsèques du taux de parallélisme et du taux d'unités de traitement utilisées d'un programme. C'est pour cela que le PSR devra apparaître en facteur avec α et β dans les lois de consommation. En conclusion, lors de l'exécution d'un code, les valeurs réelles de α et β seront toujours inférieures ou égales à ses valeurs intrinsèques.

$$\mathbf{PSR} * \alpha \leq \alpha \qquad \mathbf{PSR} * \beta \leq \beta$$

a. La loi de consommation de l'horloge

Le comportement en consommation du DSP dû à la variation de la fréquence d'horloge est le plus simple à caractériser puisque aucun scénario n'est nécessaire. En effet, il suffit pour cela de faire varier la valeur de cette fréquence dans un registre de contrôle et/ou de changer le quartz qui génère de façon externe l'horloge. La figure ci-dessous montre sous forme graphique les résultats obtenus.

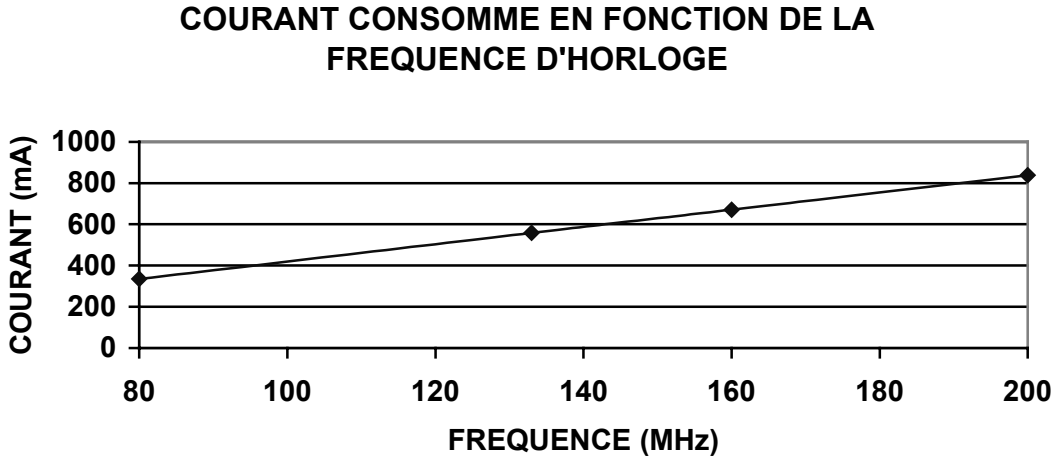

Figure III. 14 Représentation du comportement en consommation en fonction de la fréquence

Les mesures physiques réalisées nous permettent d'obtenir le courant consommé par l'horloge en fonction de sa fréquence. Nous pouvons constater que le courant consommé est linéaire avec la fréquence ce qui est conforme à la théorie. A partir de cette représentation, il est facile de déterminer la loi mathématique qui représente ce comportement. Dans le cas du C62 la consommation de l'horloge est donnée par la formule suivante :

$$I_{Horloge}(mA) = 4.19 * F(MHz) \quad (6)$$

Cette loi mathématique peut être vérifiée puisque Texas Instruments **[Tex98]** donne dans un document la consommation de l'horloge pour ce processeur : cette valeur est de **4.21* F**. L'erreur entre notre valeur et celle fournie par Texas peut s'expliquer par les erreurs dues aux appareils de mesures.

Pour déterminer les autres lois de consommation, il faudra utiliser des scénarii permettant de faire varier les valeurs des paramètres. Il n'est pas toujours possible de faire varier ces paramètres de façon indépendante ; par exemple nous ne pouvons faire

varier le nombre d'unités de traitement utilisées sans exciter l'Unité de Gestion des Instructions puisque c'est elle qui réalise la lecture des instructions. Il nous sera donc impossible de mesurer indépendamment la consommation de chacune des unités fonctionnelles ; il faudra dans ce cas utiliser une identification différentielle.

Pour déterminer la stratégie de mesures permettant d'identifier les lois de consommation, reprenons l'analyse fonctionnelle de ce DSP.

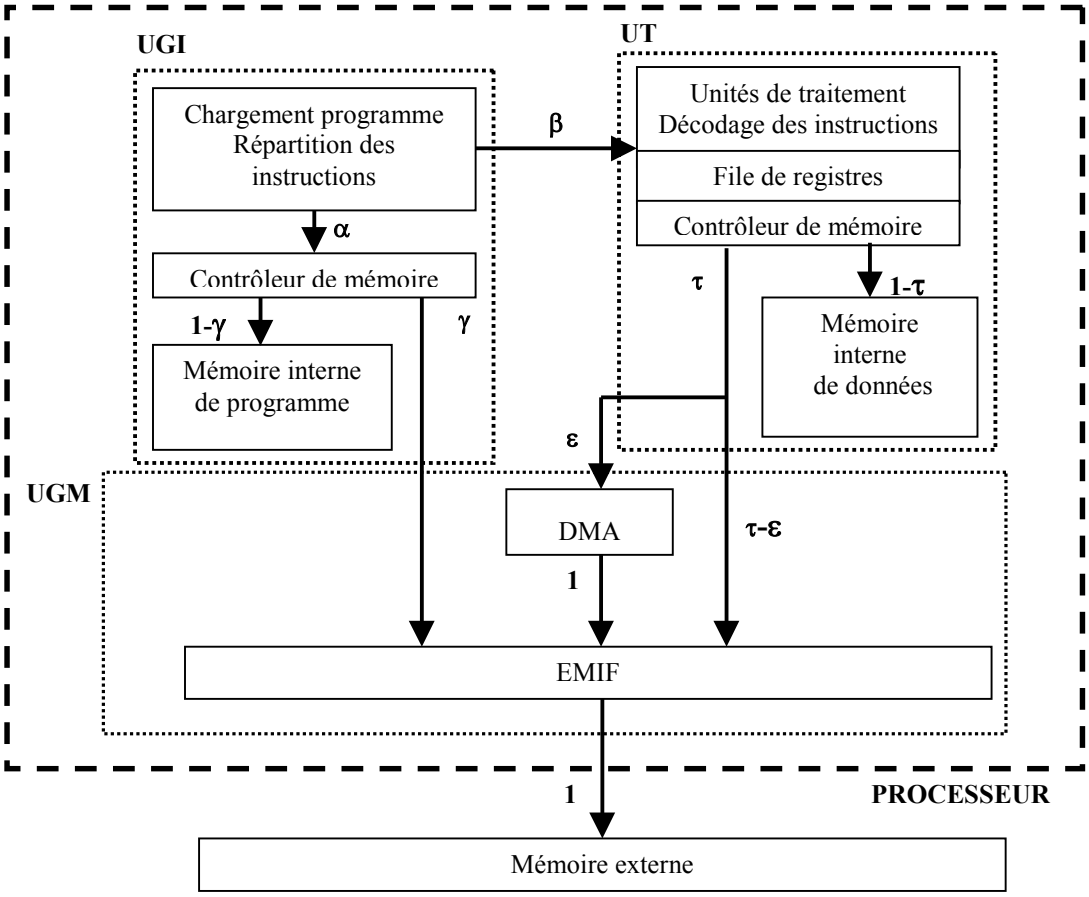

Figure III. 15 FLPA appliquée au TMS320C6201

La stratégie idéale est de suivre la propagation des instructions dans le pipeline ; c'est à dire de commencer par caractériser l'UGI puis l'UT et enfin l'UGM. Cette méthode nous permettra d'utiliser une identification incrémentale et différentielle.

b. La loi de consommation de l'UGI

Commençons donc par étudier la consommation en puissance de l'UGI. Les différent sous-blocs, composant cette unité fonctionnelle, ainsi que les liens en consommation sont donnés dans la figure ci-dessous.

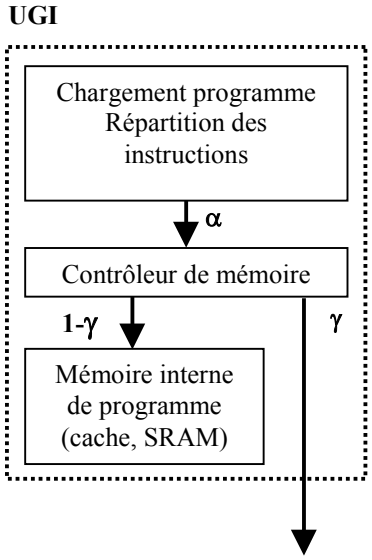

Figure III. 16 Composition de l'Unité de Gestion des Instructions

Nous voyons que cette unité fonctionnelle est composée des étages de pipeline chargement et répartition des instructions, du contrôleur de mémoire programme ainsi que de la mémoire de programme interne. Deux liens de consommation apparaissent également : le taux de parallélisme α et le taux de défauts de cache γ. Le lien γ n'existe que si la mémoire de programme interne du processeur est utilisée en mode CACHE ou en mode FREEZE (cf. III.1.3). Le lien α n'existe pas si la mémoire de programme interne du processeur est utilisée en mode BYPASS puisque toutes les instructions sont chargées de la mémoire externe. Dans ce cas une seule instruction peut être chargée à la fois ce qui réduit le parallélisme intrinsèque du code écrit. En conclusion, le comportement en consommation de l'UGI dépend bien sûr des paramètres algorithmiques mais surtout du paramètre de configuration mode mémoire MM. Il faut donc commencer par caractériser le comportement en consommation de l'UGI dans chacun de ces modes.

Nous aurons une loi de consommation pour chacun des 4 modes mémoires utilisables pour ce DSP. Pour les modes MAPPED et BYPASS, nous utiliserons le même scénario qui fera seulement varier le taux de parallélisme du programme (nous vérifierons ainsi que le paramètre α n'a aucune influence sur la consommation en BYPASS). Pour être sûr de ne pas exciter les unités de traitement et la mémoire interne ou externe de données, nous utiliserons dans ce scénario les instructions NOP. Le scénario utilisé est présenté dans la figure ci-dessous.

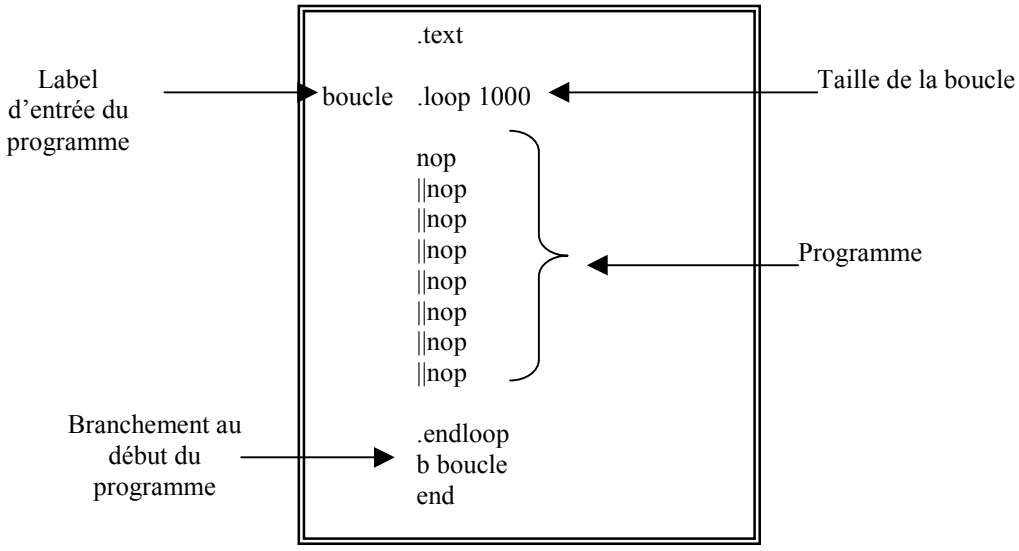

Figure III. 17 Exemple de scénario utilisé

Le symbole « || » permet de spécifier que l'instruction doit être exécutée en parallèle avec la précédente. Ceci nous permet donc de faire varier le taux de parallélisme α de 0 (toutes les instructions sont séquentielles) à 1 (toutes les instructions sont parallèles). Comme nous connaissons la consommation de l'horloge, nous pouvons déduire des mesures réalisées la consommation de l'UGI simplement en soustrayant à ces mesures la consommation de l'horloge. Nous avons pu, grâce à ce scénario, déterminer la loi de consommation de l'UGI en mode MAPPED et BYPASS.

Pour le mode CACHE et FREEZE, deux paramètres algorithmiques sont à prendre en compte, le taux de parallélisme α ainsi que le taux de défauts de cache γ. Pour faire varier α, il suffit, comme nous l'avons vu précédemment, de faire varier le nombre d'instructions en parallèle.

Pour faire varier le taux de défauts de cache, il suffit d'augmenter la taille de la boucle. Chaque trame qui dépasse du cache génère 2 défauts car le cache du C6x est à correspondance directe. Lorsque le scénario aura une taille de code égale à 1,5 fois celle du cache alors le taux de défauts sera égal à 100%. Lors de nos expériences, nous avons fait varier γ de 0 à 1 et, pour chacune des valeurs de γ, nous avons également fait varier α de 0 à 1. La loi de consommation pour chacun des modes mémoires est donnée dans le tableau ci-dessous.

MODES MEMOIRE	LOIS DE CONSOMMATION
MAPPED	$I = 5.21\alpha(1-PSR)F + 42.4\alpha(1-PSR) + 7.6$
BYPASS	$I = 5.68F + 39$
CACHE	$I = (8.55F + 184)\alpha\ (1-PSR)^{[-0.1249Log(\gamma)\ -0.002276\ ]}$ $(\gamma>0)$ $I = 4.36\alpha(1-PSR)F + 4.09F + 187.83\alpha(1-PSR) + 53.445$ $(\gamma=0)$
FREEZE	$I = (9.07F + 118)\alpha(1-PSR)^{[-0.14Log(\gamma)-0.0011]}$ $(\gamma>0)$ $I = 4.43\alpha(1-PSR)F + 4.72F + 203.1\alpha(1-PSR) - 38.52$ $(\gamma=0)$

Tableau III. 1 Lois de consommation de l'UGI en fonction du mode mémoire

Les résultats de ce tableau montrent que nous avons des lois linéaires pour les modes MAPPED et BYPASS ; ces lois sont tout à fait conformes à la théorie.

En revanche, les lois pour le mode CACHE et FREEZE possèdent deux plages : Une lorsque $\gamma=0$ et l'autre lorsque $\gamma>0$. Lorsqu'aucun défaut de cache n'est généré alors le comportement en consommation est similaire à celui en mode MAPPED à ceci près que des comparaisons d'étiquettes sont réalisées. A l'inverse, si des défauts de cache sont générés alors un surcoût de consommation est engendré par l'écriture de la mémoire de programme, en plus de celui généré par la comparaison d'étiquettes.

La deuxième plage de variation (pour des valeurs de $\gamma>0$) a une forme non linéaire. Ce phénomène s'explique par le fait que le nombre de ruptures de pipeline engendré dans ce cas croît de façon exponentielle avec le taux de défauts de cache. En effet, chaque défaut de cache engendre des accès à la mémoire externe et comme chaque accès à la mémoire externe génère des ruptures de pipeline alors chaque défaut fera augmenter le nombre de ruptures de pipeline. La figure ci-dessous montre la variation du PSR en fonction du taux de défauts de cache et ceci pour différentes valeurs du taux de parallélisme α.

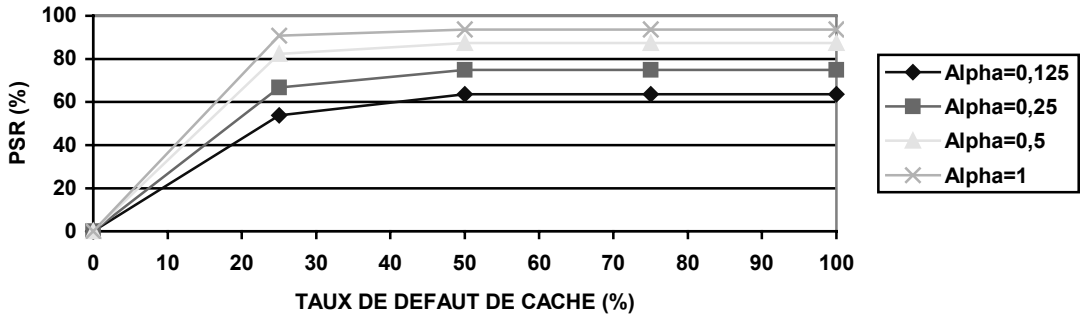

Figure III. 18 Variation du PSR en fonction du taux de défauts de cache et du taux de parallélisme

Lors de cette étude, nous nous sommes aperçus que le courant consommé par le DSP était constant quelle que soit la valeur du taux de parallélisme lorsque le taux de défaut de cache était supérieur ou égal à 0,5. En effet, le PSR influe sur le taux de parallélisme intrinsèque α (PSR * $\alpha \leq \alpha_{intrinsèque}$); notre étude a montré que la valeur (PSR*α) était identique quelle que soit la valeur α intrinsèque du programme lorsque $\gamma \geq 0,5$ (cf. Figure III.18).

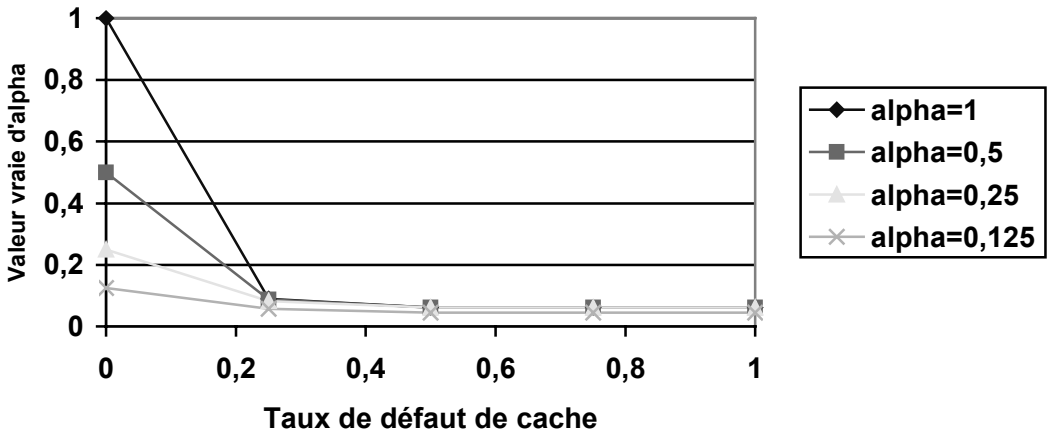

Figure III. 19 Variation du PSR en fonction du taux de défauts de cache

Les valeurs de α étant donc identiques, il est normal que les courants consommés soient identiques.

c. La loi de consommation de l'UT

Le second bloc fonctionnel à modéliser est l'Unité de Traitement. Les différents sous-blocs constituant cette unité ainsi que les liens en consommation sont donnés dans la figure III.20.

Ce bloc fonctionnel est constitué de la mémoire de données interne, de son contrôleur, de la file de registres, des unités de calculs (multiplieurs, UAL) ainsi que de la logique de décodage des instructions. En fait, le décodage des instructions se fait directement à l'intérieur des unités de calculs après que les instructions y aient été assignées. Nous voyons également que cette unité possède un paramètre entrant, le taux d'unités de traitement utilisées β et deux paramètres sortant le taux d'accès de données en externe τ et le taux d'accès en données via le DMA ε. Ce dernier paramètre sera pris en compte lors de l'étude de l'Unité de Gestion Mémoire. Pour modéliser ce bloc fonctionnel, il faut aussi tenir compte du paramètre de configuration placement mémoire

DM. En effet, ce paramètre permet de savoir si les données sont en mémoire externe ou en mémoire interne de données. Si elles sont en mémoire interne, ce paramètre permet également de savoir dans quel banc mémoire sont stockées ces données.

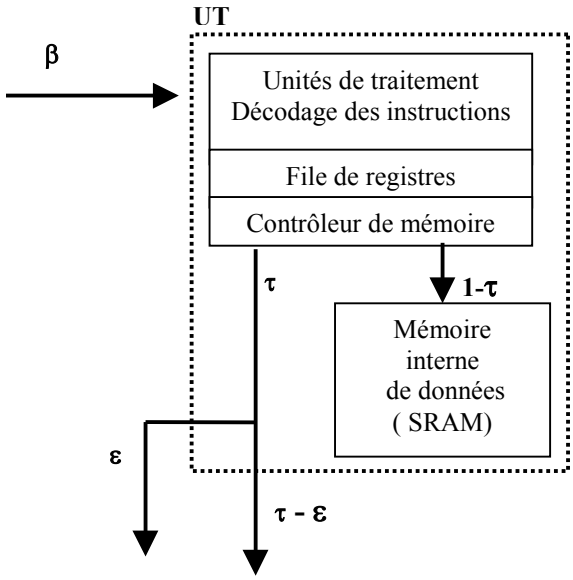

Figure III. 20 Composition de l'Unité de Traitement

Pour caractériser le comportement en consommation de ce bloc en fonction du taux d'unités de traitement utilisées β, il suffit de faire varier ce taux de 0 à 1. Cette variation est obtenue en utilisant un scénario similaire à celui employé pour caractériser l'UGI, mais en remplaçant les NOP par des opérations de calcul si nous voulons une valeur de β différentes de zéro. Par exemple pour avoir β égal à 0, il suffit de n'utiliser que des NOP dans le scénario et pour avoir β égal à 1, il faut utiliser toutes les unités de calculs de l'architecture en parallèle.

Nous ne faisons ici aucune différence entre les consommations des différentes unités de traitement car nous avons pris pour hypothèse que cette différence serait très faible par rapport à la consommation globale du circuit. Nous avons vérifié cette hypothèse par une campagne de mesures ; les résultats ont montré que la différence maximale sur la consommation globale était de **2%** entre l'utilisation d'additions ou de multiplications dans un code. Notre étude a également montré que les valeurs des données n'avaient quasiment aucune influence sur la consommation des instructions. L'écart maximal de consommation entre les différents types de données est de **0,32%**.

Les deux autres paramètres à prendre en compte sont τ et DM. En fait, le paramètre τ est lié au paramètre DM. En effet, si aucun accès n'est réalisé en mémoire

externe pour charger des données alors le paramètre τ sera nul. Le paramètre DM ne caractérisera alors plus, dans ce cas, que les conflits de bancs internes.

Comme nous l'avons vu précédemment, pour ce processeur, les paramètres τ et DM peuvent être regroupés dans le paramètre taux de rupture de pipeline PSR.

En conclusion, le bloc fonctionnel Unité de Traitement est donc caractérisé par une loi de consommation qui a pour paramètres β et PSR.

UNITE DE TRAITEMENT
$I_{UT} = 0{,}64 * \beta\ (1 - PSR) * F$

Tableau III. 2 Loi de consommation de l'unité de traitement

d. Loi de consommation de l'UGM

Il ne reste alors qu'à étudier le comportement en consommation du bloc fonctionnel Unité de Gestion Mémoire. Les différents sous blocs constituant cette unité ainsi que les liens de consommation sont donnés dans la figure ci-dessous.

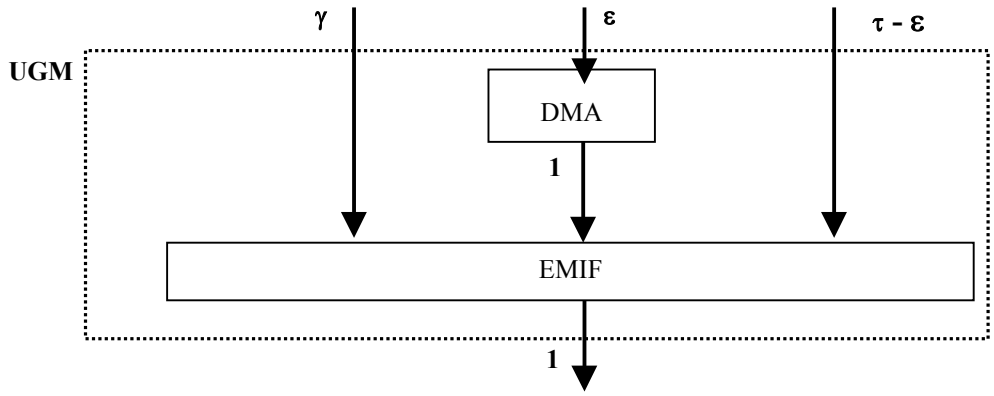

Figure III. 21 Composition du bloc fonctionnel Unité de Gestion Mémoire

Ce bloc fonctionnel est constitué du DMA ainsi que de l'EMIF. L'EMIF (External Memory InterFace) possède deux liens de consommation, le taux de défaut de cache γ et $(\tau - \varepsilon)$ qui représente le taux d'accès de données en mémoire externe qui n'est pas réalisé via le DMA. En fait, ces deux liens ont déjà été pris en compte lors de l'étude du comportement en consommation de l'UGI et de l'UT. En effet, à chaque fois

qu'un accès en mémoire externe doit être fait, l'EMIF doit être sollicité. L'UGI peut faire des accès en externe si elle est utilisée en mode CACHE ou FREEZE. L'UT peut aussi réaliser des accès externes lorsque les données sont en mémoire externe. C'est pour ces raisons que nous avons intégré l'EMIF dans les lois de consommation de l'UGI et de l'UT.

Il ne reste donc plus qu'à étudier le comportement en consommation du DMA en fonction du paramètre ε. Pour cela, nous avons utilisé un scénario dans lequel nous utilisons le DMA pour charger un tableau de données dans la mémoire interne. Ce scénario fait varier la taille du tableau à charger. Nos expériences nous ont montré que le courant absorbé par le DMA était constant quelle que soit la taille du tableau mais variait en fonction de la fréquence de fonctionnement du DSP ainsi qu'en fonction de la largeur des données (8, 16, 32 bits). L'horloge des mémoires externes est fournie par le DSP or, celui-ci peut fonctionner jusqu'à 200MHz ce qui n'est pas le cas des mémoires. Pour résoudre ce problème le DSP génère une horloge mémoire égale à la moitié de sa fréquence de fonctionnement lorsque celle-ci est supérieure à celle des mémoires externes, et égale à sa fréquence autrement.

La loi de consommation qui caractérise le DMA est donc fonction de la largeur des données (**L**)et de la fréquence d'horloge du DSP (F). Cette loi aura deux plages de fonctionnement une si la fréquence du DSP est égale à la fréquence des mémoires externes l'autre si sa fréquence est supérieure.

DMA	
$\mathbf{F_{DSP} > F_{Mémoire}}$	$\mathbf{I = (0.077LF + 2.12F + 2.05L + 94.72)\ \varepsilon}$
$\mathbf{F_{DSP} \leq F_{Mémoire}}$	$\mathbf{I = (-0.083LF + 4.9F + 24.93L – 476.16)\ \varepsilon}$

Tableau III. 3 Loi de consommation du DMA

A présent, nous avons une caractérisation globale en consommation du DSP TMS320C6201. Le modèle de puissance de ce processeur est composé de 5 lois différentes, ces lois sont présentées dans le tableau récapitulatif ci-dessous.

MODES MEMOIRE	LOIS DE CONSOMMATION
MAPPED	$I = 5.21\alpha(1-PSR)F + 4.19F + 42.40\alpha(1-PSR) + 7.6 + 0.64\beta(1-PSR)F$
BYPASS	$I = 9.87F + 39 + 0.64\beta(1-PSR)F$
CACHE	$I = (8.55F + 184)\alpha\ (1-PSR)^{[-0.1249Log(\gamma)\ -0.002276\ ]} + 0.64\beta(1-PSR)F \quad (\gamma>0)$ $I = 4.36\alpha(1-PSR)F + 4.09F + 187.83\alpha(1-PSR) + 53.45 + 0.64\beta(1-PSR)F \quad (\gamma=0)$
FREEZE	$I = (9.07F + 118)\alpha(1-PSR)^{[-0.14Log(\gamma)-0.0011]} + 0.64(1-PSR)F \quad (\gamma>0)$ $I = 4.43\alpha(1-PSR)F + 4.72F + 203.1\alpha(1-PSR) - 38.52 + 0.64\beta(1-PSR)F \quad (\gamma=0)$
DMA	
$I = (0.077LF + 2.12F + 2.05L + 94.72)\ \varepsilon\ (F_{DSP} \leq F_{mémoire})$ $I = (-0.083LF + 4.9F + 24.93L - 476.16)\ \varepsilon\ (F_{DSP} > F_{mémoire})$	

Tableau III. 4 Lois de consommation du TMS320C6201

III.3 Conclusion

Dans ce chapitre, nous avons présenté une méthodologie de modélisation en consommation d'un processeur : la FLPA. Cette méthodologie est basée sur une approche fonctionnelle et paramétrique ce qui permet de réduire grandement le nombre de mesures à réaliser donc le temps de modélisation de la cible. La FLPA a été appliquée avec succès sur deux processeurs de traitement numérique du signal le TMS320C6201 et le TMS320C5510 (les résultats pour ce processeur sont disponibles dans **[Far02]**). Le premier est un processeur VLIW d'ordre 8 qui permet de réaliser jusqu'à 1,6 milliards d'opérations par seconde. Ce processeur est utilisé pour des applications requérant un nombre de traitements très important. Le deuxième est la dernière génération de processeurs basse consommation de Texas Instruments. Il est plutôt destiné aux applications embarquées ayant des contraintes de consommations fortes.

Le temps nécessaire à la modélisation en consommation du C62 est d'environ 45 jours. Ce temps peut paraître important mais il est bien inférieur au temps nécessaire avec les approches classiques existantes. Pour donner une idée du gain, nous pouvons comparer nos résultats avec ceux donnés dans **[Bona02]**. Dans cet article, Bona et al. ont modélisé un processeur Lx d'ordre 4 par une approche fonctionnelle à grain fin. Le temps de modélisation nécessaire dans ce cas est de 108 jours ; l'utilisation de notre méthode entraîne donc un gain de temps de 63%. Pour des processeurs plus simples comme par exemple le C55, ce temps est encore plus faible puisqu'il nous a fallu environ 30 jours pour le modéliser.

Les lois de consommation composant le modèle de puissance sont fonction de paramètres algorithmiques et de configuration. Les paramètres de configuration étant fixés par l'utilisateur, l'obtention de leur valeur est simple. Les valeurs des paramètres algorithmiques dépendent, elles, de l'algorithme ; nous expliquerons donc dans le chapitre suivant comment sont déterminées les valeurs de ces paramètres.

Chapitre IV

Estimation de la consommation d'un algorithme

La première partie de ce chapitre expliquera la méthode d'obtention des valeurs des paramètres algorithmiques à partir du code assembleur. Nous expliquerons les différentes étapes de ce processus en utilisant l'exemple du DSP TMS320C6201.

La deuxième partie de ce chapitre expliquera comment il est possible d'obtenir directement, à partir du source C, les valeurs de certains paramètres algorithmiques en utilisant des modèles de prédiction. Nous utiliserons des exemples pédagogiques pour expliquer clairement cette méthode. Nous donnerons l'intérêt d'une telle méthode par rapport à celle développée dans la première partie.

Enfin, la dernière partie de ce chapitre sera consacrée à la présentation de l'outil d'analyse de la consommation SoftExplorer développé au cours de cette thèse.

IV.1 Méthode au niveau assembleur

Nous avons vu dans le chapitre précédent que les lois de consommation étaient fonction de paramètres algorithmiques et de configuration. Les valeurs des paramètres de configuration sont fournies par l'utilisateur ; en revanche, les valeurs des paramètres algorithmiques dépendent de l'application à exécuter. Ces valeurs peuvent être déterminées à partir du code assembleur généré par le compilateur.

Nous allons tout d'abord expliquer comment les différents paramètres algorithmiques sont calculés.

IV.1.1 Calcul des paramètres algorithmiques

Nous avons vu dans le chapitre I que notre contexte d'application serait les algorithmes de traitement du signal et de l'image. Ces algorithmes sont constitués de boucles dont les bornes peuvent être connues à l'avance (déterminisme d'exécution) ou dépendantes des données (exécution non déterministe).

Si les tailles de ces boucles sont figées, alors chacune d'entre elles sera étudiée comme étant un bloc linéaire d'instructions (BLI). Chacun de ces BLI possédera donc ses propres paramètres algorithmiques appelés paramètres locaux. Une fois que toutes les valeurs locales auront été déterminées, nous pourrons calculer les paramètres globaux de l'application en réalisant une moyenne pondérée des valeurs locales.

Si les tailles de boucles sont dépendantes des données, alors il faudra déterminer un jeu de vecteurs d'entrées permettant de caractériser au mieux l'exécution de ces boucles. La première solution pour déterminer le jeu de vecteurs, est de choisir un jeu représentant les données typiques utilisées par ces applications. Une deuxième solution est de prendre un jeu de vecteurs qui va maximiser la taille des boucles ; nous aurons alors une représentation du pire cas de l'exécution. Quelle que soit la solution retenue, nous étudierons chacune des boucles comme étant un BLI. Nous appliquerons donc la même méthode que pour les boucles bien formées.

Nous avons vu dans le chapitre III que les lois de consommation sont fonction de 5 paramètres algorithmiques ; nous devons donc déterminer les valeurs de ces 5 paramètres pour réaliser l'estimation d'une application. Pour chacun d'eux, nous présenterons un exemple pédagogique appliqué sur le TMS320C6201.

a. Détermination des paramètres locaux

Le premier paramètre local à déterminer est le taux de parallélisme α de chaque BLI. Ce paramètre n'existe que pour les architectures VLIW ou super scalaires. Dans le cas d'architectures de ce type, le paramètre local α est calculé de la façon suivante :

$$\alpha = \frac{NbFP}{NbEP} \quad \textbf{(1)}$$

NbFP : représente le nombre de paquets de mots d'instructions chargés lors de l'exécution du programme encore appelé paquet de chargement. Un paquet de chargement peut être constitué d'un ou plusieurs paquets d'exécution.
NbEP : représente le nombre de paquets d'exécution du programme.

Le deuxième paramètre algorithmique local à calculer est le taux d'unités de traitement utilisées β. Ce paramètre, tout comme le taux de parallélisme, n'existe que pour des architectures VLIW ou super scalaires. Dans le cas de ce type d'architecture, le paramètre local β est calculé de la façon suivante :

$$\beta = \frac{1}{Nb\_Max\_UT}\frac{NbUT}{NbEP} \quad \textbf{(2)}$$

NbUT : représente le nombre d'unités de traitement utilisées lors de l'exécution du programme.
Nb_Max_UT : représente le nombre maximum d'unités de traitement utilisables en parallèle.

b. Détermination des paramètres globaux

Pour un programme constitué de plusieurs BLI, il suffit de calculer le taux de parallélisme ainsi que le taux d'unités de traitement utilisées de chacun des blocs linéaires d'instructions constituant le programme et ensuite d'en faire une moyenne pondérée par le nombre de fois que ce BLI est itéré. Le taux de parallélisme local d'un BLI est calculé en utilisant la formule (1) et le taux de parallélisme global du programme est calculé en utilisant la formule ci-dessous :

$$\alpha_{global} = \frac{\sum(\alpha_{locaux} \times NbIT)}{\sum NbIT} \quad \textbf{(3)}$$

NbIT : représente le nombre d'itérations de chaque BLI

Comme précédemment nous devrons calculer le taux d'unités de traitement utilisées β de façon locale (pour chaque BLI) et ensuite calculer le β global du programme à l'aide de la formule ci-dessous :

$$\beta_{global} = \frac{\sum(\beta_{locaux} \times NbIT)}{\sum NbIT} \quad \textbf{(4)}$$

Exemple de calculs des paramètres locaux et globaux

Prenons un exemple de code assembleur où deux boucles sont utilisées. La première est itérée 4 fois et la seconde 6 fois ; ce programme est présenté ci-dessous.

```
Début
          Add a0, a1, a2
          ||mpy b0, b1, b2
          ||add a3, a4, a5
          ||sub b3, b4, b5
          ||nop 4

boucle1   add a1, a2, a0
          ||mpy b0, b4, b7
          ||sub a8, a3, a6
          ||mpy a4, a5, a9
          ||add b2, b3, b10
          sub 1, a14
          cmp a14, a15
[ !a15]   b boucle1

boucle2   add a0, a1, a2
          sub b0, b1, b2
          ||mpy a3, a5, a4
          ||mpy b3, b7, b8
          add a8, a9, a0
          sub 1, b14
          cmp a14, a15
[ !b15]   b boucle2

          end
```

Figure IV. 1 Exemple de programme assembleur sur le TI C6x

La première étape est de calculer les paramètres locaux du programme. Ici, nous aurons 3 paramètres locaux : un pour chacune des boucles et un autre pour le début du

programme. Les taux de parallélisme locaux sont calculés en utilisant la formule suivante :

$$\alpha_{local} = \frac{NbFP}{NbEP}$$

Le paramètre NbFP correspond au nombre de paquets de chargement utilisés par le programme. Ce processeur charge 8 instructions simultanément puisque c'est un VLIW d'ordre 8. Un paquet de chargement FP correspondra donc à 8 instructions consécutives dans le code.

Le paramètre NbEP correspond au nombre de paquets d'exécution EP utilisés par le programme. Chaque paquet de chargement du programme peut être constitué de 1 à 8 paquets d'exécution en fonction du parallélisme du programme.

Reprenons maintenant notre exemple de programme et calculons son nombre de FP et son nombre EP.

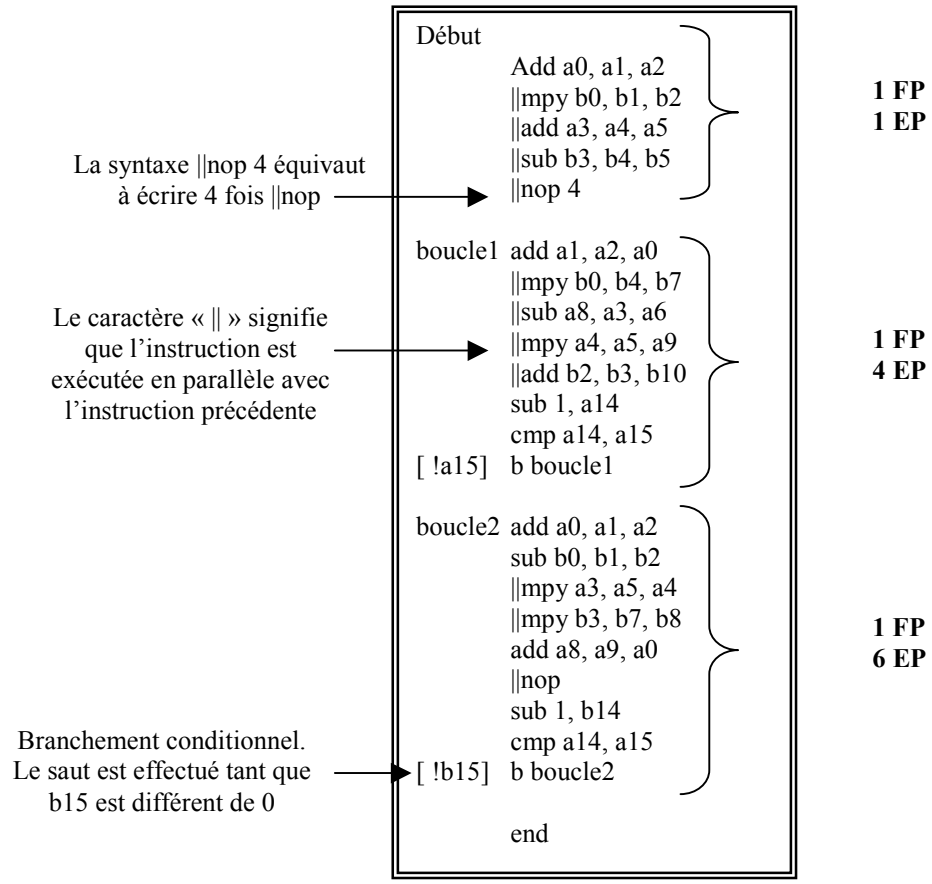

Figure IV. 2 Détermination du nombre d'EP et d'FP d'un programme

A présent, nous pouvons calculer les trois α locaux du programme en utilisant l'équation 1.

$$\alpha_1 = \frac{1}{1} = 1$$

$$\alpha_2 = \frac{1}{4} = 0.25$$

$$\alpha_3 = \frac{1}{6} = 0.167$$

Une fois les différents paramètres locaux calculés, nous pouvons déterminer le taux de parallélisme global du programme en utilisant l'équation 2.
Ce qui donne, dans notre cas, le résultat suivant :

$$\alpha_{global} = \frac{(1 * 1) + (0.25 * 4) + (0.167 * 6)}{11} = 0.273 \quad \textbf{(5)}$$

Nous allons maintenant montrer comment est calculé le taux d'unités de traitement utilisées β. Tout comme le taux de parallélisme, ce taux est calculé à partir de paramètres locaux. Les valeurs de ces paramètres sont déterminées en utilisant la formule suivante :

$$\beta_{local} = \frac{1}{Nb\_Max\_UT} \frac{NbUT}{NbEP}$$

Dans le cas du C62, le paramètre Nb_Max_UT est égal à 8 puisqu'au maximum 8 unités de traitement peuvent être utilisées simultanément. Le paramètre Nb_UT représente, le nombre d'unités de traitement utilisées par le programme ; le paramètre Nb_EP à, lui, déjà été déterminé lors du calcul du paramètre α. Reprenons donc notre exemple de programme et calculons son nombre d'UT. Comme les branchements utilisent une unité de traitement pour calculer l'adresse du saut, ces instructions seront prises en compte lors du calcul du nombre d'UT.

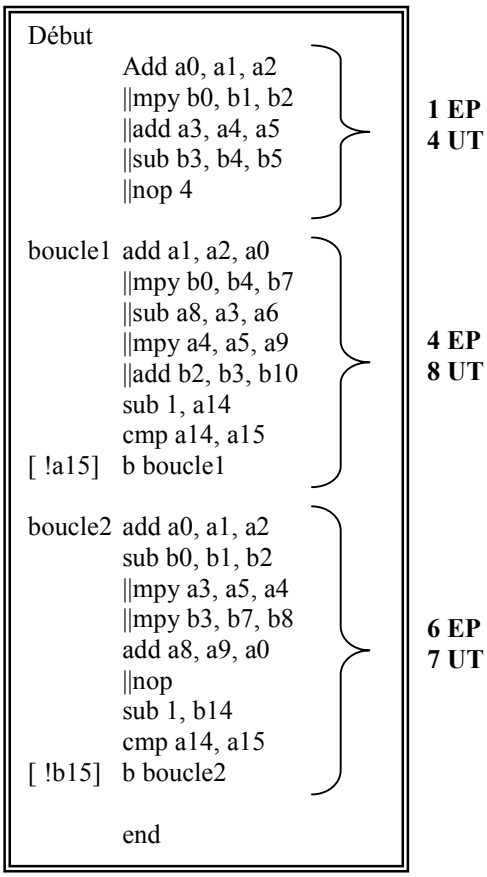

Figure IV. 3 Détermination du nombre d'EP et d'UT d'un programme

Nous pouvons alors calculer les 3 β locaux du programme en utilisant l'équation 3.

$$\beta_1 = \frac{1}{8} \times \frac{4}{1} = 0.5$$

$$\beta_2 = \frac{1}{8} \times \frac{8}{4} = 0.25$$

$$\beta_3 = \frac{1}{8} \times \frac{7}{6} = 0.146$$

Une fois ces paramètres locaux déterminés, nous pouvons calculer le taux d'unités de traitement utilisées global du programme en utilisant l'équation 4 ; ce qui donne dans notre cas le résultat suivant :

$$\beta_{global} = \frac{(1 * 0.5) + (4 * 0.25) + (6 * 0.146)}{11} = 0.216 \quad \textbf{(6)}$$

c. Détermination du taux d'accès de données via le DMA ε

Le troisième paramètre algorithmique à déterminer est le taux d'accès de données via le DMA. Le DMA devant être configuré par l'utilisateur, celui-ci peut donc connaître le nombre de données qui vont être accédées au cours du programme ainsi que leurs adresses. En effet, la configuration du DMA se fait par le rajout d'une fonction assembleur qui spécifie l'adresse de début et de fin de la structure à charger. Connaissant également le nombre total de données qui seront accédées en mémoire externe grâce au « mapping » mémoire, nous pouvons facilement calculer le taux d'accès de données via le DMA ε. Cette valeur est calculée en utilisant la formule suivante :

$$\varepsilon = \frac{Nb\_Données\_DMA}{Nb\_Total\_Données} \quad (7)$$

Nb_Données_DMA : représente le nombre de données accédées via le DMA

Nb_Total_Données : représente le nombre total de données accédées en mémoire externe

Exemple de détermination du taux d'accès de données via le DMA

Dans le paragraphe précédent, nous avons expliqué comment, de façon générale, était calculé ε. Dans le cas du C62, nous utilisons l'outil Code Composer fourni par Texas Instruments. Code Composer est l'environnement de développement pour les DSP de Texas Instruments **[Tex01]**. Il regroupe différentes fonctionnalités comme :

- 1 éditeur de code C ou ASM
- 1 compilateur/éditeur de liens
- 1 simulateur
- 1 débuggeur

Le simulateur permet, par profiling dynamique de l'algorithme, de déterminer le nombre d'accès de données en mémoire externe. Pour cela, il suffit de placer un point d'arrêt en fin de programme et de lancer une exécution du code. Connaissant le nombre d'accès de données via le DMA puisque nous l'avons programmé, il ne reste plus qu'à appliquer l'équation 7 pour déterminer le taux d'accès de données via le DMA.

La programmation du DMA se fait par l'écriture d'une fonction en assembleur. La première étape de ce programme est de définir l'adresse source à laquelle doit

débuter le transfert (adresse en mémoire externe). La deuxième étape consiste à définir l'adresse de destination (adresse en mémoire interne), la taille de la structure ainsi que la largeur des mots à transférer (8,16 ou 32 bits). Enfin, la dernière étape permet d'autoriser le début de transfert. La figure ci-dessous présente un exemple de la programmation du DMA.

```
.global _asmdma ; déclaration du fonction externe
_asmdma :

        * adresse source du transfert
        mvkl    0x000,A1
        mvkh    0x0040,A1
        mvkl    0x0010,A2
        mvkh    0x0184,A2
        stw     A1,*A2

        * adresse de destination
        mvkl    0x0400, A1
        mvkh    0x8000,A1
        mvkl    0x0018,A2
        mvkh    0x0184,A2
        stw     A1,*A2
        mvkl    0x0050,A1
        mvkh    0x0100,A1
        mvkl    0x0000,A2
        mvkh    0x0184,A2
        stw     A1,*A2

        * autorisation de transfert
        mvkl    0x0051,A1
        mvkh    0x0100,A1
        stw     A1,A2

        * retour au programme principal
        b        B3
        nop     5
```

Figure IV. 4 Exemple de fonction assembleur permettant la programmation du DMA

d. Détermination du taux de défauts de cache γ

Le quatrième paramètre algorithmique à calculer est le taux de défaut de cache γ. Généralement les algorithmes de TdSI sont constitués de boucles imbriquées. Le taux de défaut de cache dépendra bien entendu de la taille du programme mais également de la position de ces boucles dans le programme ainsi que de leur nombre d'itérations (cf. chapitre III.1.2.a).

Le taux de défaut de cache sera donc calculé en utilisant un « profiling » du code pour pouvoir déterminer le nombre de défauts de cache générés par son exécution. Dans le cas où la taille des boucles dépend des données utilisées par le programme, le test de tous les cas possibles n'est pas envisageable; le taux de défaut de cache utilisé pour l'estimation sera, dans ce cas, une valeur moyenne.

Exemple de détermination du taux de défauts de cache

Nous avons vu dans le paragraphe précédent comment, de façon générale, était calculé le taux de défaut de cache. Dans le cas du TMS320C6201, nous utilisons l'outil Code Composer fourni par Texas Instruments pour le déterminer. En effet, cet outil offre en autre, comme fonctionnalité, de permettre le calcul du nombre de défauts de cache. Pour cela, il suffit de placer un point d'arrêt à la fin du programme et ensuite d'exécuter le programme jusqu'à ce point. L'outil nous donne alors le nombre de réussites, de défauts de cache ainsi que le nombre de paquets de chargement engendré par l'exécution du programme (cf. **[Tex01]**). Une fois que ces valeurs ont été déterminées, il est alors aisé de déterminer le taux de défaut de cache du programme en utilisant l'équation ci-dessous :

$$\gamma = \frac{NbDéfaut}{NbLecture}$$

Pour les algorithmes dont la taille des boucles dépend des données, nous utiliserons soit un jeu d'entrées typiques soit une valeur maximale afin de définir leur taux de défauts de cache.

e. Détermination du taux de ruptures de pipeline PSR

Le dernier paramètre algorithmique à calculer est le taux de ruptures de pipeline PSR. Sa valeur est calculée en utilisant la formule suivant :

$$PSR = \frac{Nb\_RP}{Nb\_Cycle\_Exe} \quad (8)$$

Nb_RP : représente le nombre de ruptures de pipeline générés par l'exécution du code ; il est exprimé en nombre de cycles processeur.

Nb_Cycle_Exe : représente le nombre de cycles d'exécution du programme.

Nous avons vu dans le chapitre III (cf. III.1.2.a) que ce paramètre avait trois composantes : les ruptures dues aux aléas structurels, aux aléas de contrôle et celles dues aux aléas de données. Nous avons également montré qu'il suffisait de faire la somme des ruptures de pipeline dues à ces trois types d'aléas pour déterminer le nombre de total de ruptures de pipeline. Au niveau assembleur, il est possible d'obtenir directement le nombre total de ruptures de pipeline à l'aide des outils constructeurs. Ces outils réalisent un profiling de l'algorithme afin de déterminer son comportement d'exécution sur la cible ; ils peuvent ainsi repérer les ruptures de pipeline générées et déterminer le temps d'exécution de l'application. Le calcul de la valeur du PSR se fait, alors, en utilisant la formule 8.

En revanche, ces outils ne permettent généralement pas de connaître la contribution de chaque type d'aléas.

Exemple de détermination du taux de ruptures de pipeline

Pour calculer ce paramètre, il suffit de connaître le nombre de ruptures de pipeline ainsi que le temps d'exécution du programme. L'outil Code Composer nous permet d'obtenir le nombre de ruptures de pipeline exprimé en nombre de cycle processeur ainsi que le temps d'exécution. Nous utilisons le même procédé que précédemment pour obtenir ces valeurs. Ensuite, il ne reste plus qu'à appliquer la formule 8 afin de déterminer le PSR.

Une fois que les différents paramètres algorithmiques ont été calculés, il suffit d' insérer les valeurs de ces paramètres dans les lois de consommation définies lors de l'élaboration du modèle de puissance de la cible pour estimer la consommation du programme. Ces lois nous permettent de déterminer la consommation moyenne de courant. Connaissant la tension d'alimentation ainsi que le temps d'exécution du programme, nous pouvons également estimer la puissance ainsi que l'énergie moyenne consommées par le programme sur la cible considérée en utilisant les formules suivantes :

$$P_{moyen} = I_{estimé} * Vdd \quad (9)$$

$$E_{moyen} = P_{moyen} * Texe \quad (10)$$

f. Exemple d'estimation de la consommation d'un programme pour le TMS320C6201

Les différentes valeurs des paramètres étant, à présent, calculées, il ne reste plus qu'à les insérer dans les lois de consommation du TMS320C6201 pour pouvoir estimer le courant moyen, la puissance et l'énergie moyenne consommés par l'exécution de l'application sur cette cible.

Reprenons le programme utilisé en exemple et calculons son courant, sa puissance et son énergie consommés. Nous utiliserons pour cet exemple les paramètres de configuration suivants :

- Fréquence d'horloge de 200MHz.
- Mode mémoire utilisé : Mapped. Dans ce mode toutes les instructions sont en mémoire interne de programme.

Supposons également que 20% des accès externes se font via le DMA sur des données de largeur 16 bits, ce qui implique que ε est égal à 0,2 et que L est égal à 16. Supposons également que le taux de rupture de pipeline engendré par l'exécution de ce code soit de 35% (PSR égal à 0,35) et que le temps d'exécution soit égal à 20µs. La tension d'alimentation de ce processeur est de 2,5V.

Nous avons précédemment déterminé que le taux de parallélisme pour ce programme était de 0,273 et que son taux d'unités de traitement utilisées était de 0,216 (cf. chapitre IV.1.1 équations 5 et 6). Dans le chapitre III, nous avons vu que la loi de consommation pour ce DSP dans ce mode mémoire était la suivante :

$$I_{estimé} = 5.21\alpha(1\text{-}PSR)F + 4.19F + 42.401\alpha(1\text{-}PSR) + 7.6 + 5.12\beta(1\text{-}PSR)F$$

En utilisant les paramètres déterminés, nous obtenons donc le résultat suivant :
$I_{estimé}$ = [5.21*0.273*(1-0.35)*200] + [4.19*200] + [42.401*0.273*(1-0.35)] +7.6 + [5.12*0.216*(1-0.35)*200] = **1182mA**

A cette consommation, il faut rajouter la consommation due au DMA ; pour cela, nous utilisons la formule vue dans le chapitre précédent.

$$I_{DMA} = (-0.083LF + 4.906F + 24.93L - 476.16)\varepsilon \quad (car\ F_{DSP} > F_{mémoire})$$

Ce qui donne, pour notre exemple, le résultat suivant :
I_{DMA} = (-0.083*16*200 + 4.906*200 + 24.93*16 – 476.16) * 0.2 = **128mA**

Il ne nous reste plus qu'à faire la somme de ces deux courants estimés pour connaître l'estimation du courant total consommé, soit ici un courant de **1310mA**. Une fois ce courant déterminé, nous pouvons estimer la puissance et l'énergie consommé par ce programme de la manière suivante :

$$\mathbf{P = V_{dd} * I_{Total\_Estimé} = 2,5 * 1310 = 3275mW}$$
$$\mathbf{E = P * T_{EXE} = 3,275 * 20\ 10^{-6} = 65,5\mu J}$$

Nous avons vu dans cette partie comment étaient déterminées les valeurs des paramètres algorithmiques. Cette détermination repose sur l'analyse du code assembleur généré par le compilateur et sur des « profiling » dynamiques de ce code en utilisant les outils constructeurs. La détermination des paramètres pourrait se faire manuellement vue sa simplicité mais devient un peu fastidieuse pour des applications conséquentes ; c'est pourquoi nous l'avons automatisé. Cette automatisation est une des fonctionnalités de l'outil SoftExplorer que nous présenterons dans la dernière partie de ce chapitre.

IV.2 Méthode au niveau algorithme C

Nous avons vu dans le paragraphe précédent que les paramètres algorithmiques pouvaient être calculés à partir du code assembleur généré par le compilateur. Ceci implique donc d'avoir à disposition les outils constructeurs permettant de compiler le code mais également de faire du « profiling » dynamique pour obtenir certains paramètres (taux de défaut de cache et taux de ruptures de pipeline). Or, ce que veulent les utilisateurs, c'est obtenir une estimation de la consommation d'une application au plus tôt au cours du développement

Le fait de compiler le code n'est pas vraiment un handicap si le processeur devant être utilisé pour exécuter l'application est fixé. En revanche, si c'est l'application qui est fixée et qu'il faut déterminer le meilleur processeur sur lequel doit être exécutée cette application pour respecter les contraintes de consommation ; alors cette compilation devient pénalisante. En effet, dans ce cas, il faut que le concepteur possède tous les outils constructeurs des différentes cibles qu'il veut tester. Il serait alors plus

intéressant d'avoir une méthode qui permette d'obtenir des estimations de consommation sans devoir compiler le code.

Durant cette thèse, nous avons développé une telle méthode d'estimation ; elle est basée sur des modèles de prédiction qui permettent, sans compilation, d'obtenir la valeur des différents paramètres du modèle de puissance d'une cible. Elle suppose toutefois que le concepteur a, à sa disposition, les différents modèles de puissance des cibles qu'il veut tester. La méthode C n'étant pas automatisée, les modèles de prédiction ne sont appliqués que sur les boucles constituant le programme afin de réduire la complexité d'application de la méthode. L'erreur engendrée par cette simplification est limitée puisque plus de 80% des traitements sont réalisés dans les cœurs de boucles pour les applications de traitement du signal et de l'image qui représentent notre contexte d'applications.

Nous allons tout d'abord définir les différents modèles de prédiction que nous avons développés. Nous présenterons ensuite comment, à partir du source C, nous pouvons estimer le taux de ruptures de pipeline du programme. L'application de cette méthode sur des applications réelles sera présentée dans le chapitre suivant.

IV.2.1 Modèles de prédiction

Au niveau assembleur, nous calculions les valeurs réelles des paramètres algorithmiques ; au niveau C, nous ne pouvons que prédire ces valeurs puisque nous ne compilons pas le code. Les modèles de prédiction sont donc basés sur des hypothèses de comportement du compilateur. **[Val01]** a montré que la consommation dépendait fortement des options de compilation choisies ; il faut donc que notre modèle de prédiction soit capable de prendre en compte ces options. C'est pour cette raison que nous avons développé les quatre modèles de prédiction suivants :

- Modèle SEQ
- Modèle MAX
- Modèle MIN
- Modèle DATA

Nous allons reprendre ces 4 modèles un à un et expliquer les propriétés de chacun d'entre eux.

Modèle SEQ

Pour ce modèle, nous supposons que toutes les instructions seront exécutées de façon séquentielle même pour des architectures VLIW ou super scalaires. Quel est l'intérêt d'un tel modèle pour les architectures parallèles ? Nous avons dit que nous voulions prendre en compte les différentes options de compilation possibles or, lorsqu'un code est compilé sans aucune optimisation (i.e. sans options) le code généré est purement séquentiel même pour des architectures parallèles.

En conclusion, ce modèle est utilisé lorsque l'architecture cible est purement séquentielle i.e. qu'elle ne peut réaliser qu'un seul calcul par cycle ou alors que le code est compilé sans aucune option.

Modèle MAX

Dans ce modèle, nous supposons que les possibilités de l'architecture sont utilisées au mieux (parallélisme maximum). Ce modèle nous permet d'obtenir la borne supérieure de consommation (courant & puissance) d'un algorithme pour des architectures parallèles ainsi que la borne inférieure du temps d'exécution.

Modèle MIN

Dans ce modèle, nous supposons que les chargements et/ou les enregistrements ne sont jamais effectués en parallèle. En effet, nous nous sommes aperçus que le compilateur ne parallélisait pas toujours ce type d'instructions. En revanche, les opérations de calculs sont, elles, toutes réalisées simultanément dans la limite des possibilités architecturales. Ce modèle nous permet d'obtenir une borne minimale de consommation d'un algorithme pour des architectures parallèles.

Modèle DATA

Ce dernier modèle est moins restrictif que le précédent puisqu'il prend en compte le placement des données en mémoire. En effet, les chargements et/ou enregistrements en parallèle sont autorisés si, et seulement si ils se font sur des données différentes. Les opérations de calcul sont, elles, réalisées en parallèle comme pour le modèle précédent. Ce modèle a été créé pour raffiner la prédiction de consommation. En effet, dans les trois précédents modèles, nous ne nous sommes jamais préoccupés ni

des données et ni de leurs placements. Or, ce placement a une grande influence sur la consommation d'une application.

En appliquant un de ces 4 modèles de prédictions, nous obtenons donc une image virtuelle du code compilé, ce qui nous permet de déterminer les valeurs des différents paramètres algorithmiques de l'application.

Pour mieux comprendre l'utilisation de ces modèles de prédictions, nous allons appliquer chacun d'entre eux sur un exemple simple et calculer les valeurs des paramètres algorithmiques. Dans cet exemple, nous supposerons, pour calculer ces valeurs, que le processeur cible est le DSP TMS320C6201 et que les deux tableaux de données se trouvent dans des bancs mémoires internes différents.

```
For (i=0 ; i<100 ; i++)
{
Y = X[i] * (H[i] + H[i+1] + H[i-1]) + Y
}
```

Figure IV. 5 Exemple de code C

Dans cet exemple, un calcul est itéré 100 fois à l'aide d'une boucle ; ce calcul est constitué de 4 chargements et de 4 opérations de calcul. Appliquons à présent chacun des modèles de prédiction sur cet exemple.

Modèle SEQ

Nous avons vu précédemment que dans ce modèle toutes les instructions étaient exécutées de façon séquentielle. L'image du code assembleur obtenue avec ce modèle est la suivante :

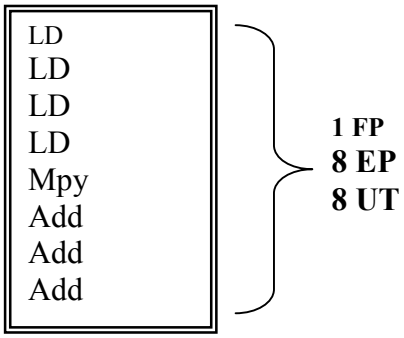

Figure IV. 6 Résultat obtenu avec le modèle SEQ

Cet exemple étant constitué de 8 instructions, nous n'avons donc qu'un seul paquet de chargement (FP). De plus comme toutes les instructions sont exécutées séquentiellement, nous avons alors 8 paquets d'exécution (EP) et puisque chacune des instructions utilise une unité de traitement (UT), nous avons donc 8 unités de traitement utilisées.

Connaissant ces différents paramètres, nous pouvons dès lors prédire le taux de parallélisme ainsi que le taux d'unités de traitement utilisées de ce programme. Les résultats sont les suivants :

$$\alpha = \frac{1}{8} = 0,125 \quad \beta = \frac{1}{8} \times \frac{8}{8} = 0,125$$

Modèle MAX

Dans ce modèle, nous supposons que les possibilités de l'architecture sont utilisées au mieux (parallélisme maximum). Dans le cas du C62, il est capable de réaliser 2 chargements ainsi que 6 autres opérations en parallèle. L'image du code assembleur avec ce modèle est le suivant :

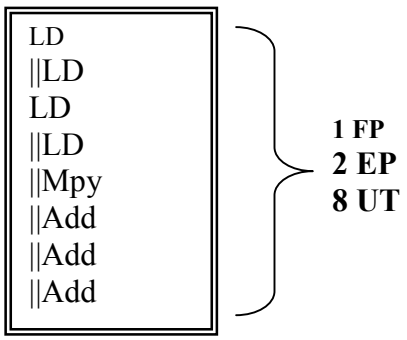

Figure IV. 7 Résultat obtenu avec le modèle MAX

Le taux de parallélisme et le taux d'unités de traitement utilisées pour ce modèle de prédiction sont alors les suivants :

$$\alpha = \frac{1}{2} = 0,5 \qquad \beta = \frac{1}{8} \times \frac{8}{2} = 0,5$$

Modèle MIN

Dans ce modèle, les chargements et/ou enregistrements ne peuvent pas être exécutés en parallèle ; en revanche, les opérations de calcul peuvent l'être. L'application de ce modèle sur l'exemple donne alors l'image du code assembleur suivante :

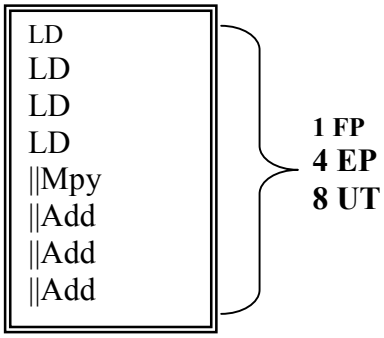

Figure IV. 8 Résultat obtenu avec le modèle MIN

Les paramètres α et β pour ce modèle de prédiction sont donc les suivants :

$$\alpha = \frac{1}{4} = 0,25$$

$$\beta = \frac{1}{8} \times \frac{8}{4} = 0,25$$

Modèle DATA

Dans ce modèle, nous avons vu que les chargements et/ou enregistrements pouvaient se faire en parallèle si, et seulement si ils s'effectuent sur des données différentes. Dans notre exemple, les structures de données X et H sont dans des bancs différents ; il est donc possible de réaliser un accès dans la structure X en parallèle avec un accès dans H. L'application de ce modèle sur notre exemple donne l'image suivante :

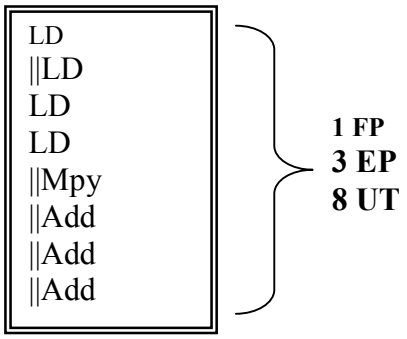

Figure IV. 9 Résultat obtenu avec le modèle DATA

Les valeurs des paramètres α et β pour ce modèle sont les suivantes :

$$\alpha = \frac{1}{3} = 0,33$$

$$\beta = \frac{1}{8} \times \frac{8}{3} = 0,33$$

Pour cet exemple, nous n'avons calculé que la valeur du taux de parallélisme ainsi que celle du taux d'unités de traitement utilisées. Les prédictions du taux de rupture de pipeline ainsi que du taux de défaut de cache seront expliqués dans le paragraphe suivant.

A l'aide de cet exemple pédagogique, nous voyons que les valeurs des paramètres algorithmiques sont différentes en fonction du modèle de prédiction utilisé. Plus les valeurs de α et β sont grandes et plus la puissance engendrée par l'exécution du code sera importante. Grâce à cette propriété, nous pouvons donc conclure que le modèle MAX modélisera la consommation de puissance la plus élevée (la borne maximale de consommation). Le modèle MIN donnera la borne inférieure de consommation et le modèle DATA donnera l'estimation de consommation la plus réaliste. Ces différentes remarques ont été validées et les résultats seront présentés dans le chapitre applications (chapitre V).

Si l'architecture cible n'est capable de réaliser qu'un seul traitement par cycle alors un seul des 4 modèles existe ; c'est le modèle SEQ.

IV.2.2 Méthode de prédiction du PSR

Les 4 modèles de prédictions que nous avons présentés dans le paragraphe précédent ne permettent pas de prédire tous les paramètres algorithmiques. En effet, ces modèles prédisent les valeurs des paramètres α et β mais pas celles des paramètres PSR et γ. Or, nous avons expliqué dans le chapitre III que le PSR avait un impact important sur la consommation de puissance et d'énergie tout comme le taux de défaut de cache. Il est donc indispensable de pouvoir prédire ces paramètres directement à partir du source C.

Nous avons vu dans le chapitre III (cf. III.1.2.a) que le taux de rupture de pipeline était calculé de la façon suivante :

$$PSR = \frac{NbRP}{Nb\_Cycle\_Exécution}$$

La première étape est de déterminer le nombre de ruptures de pipeline NbRP. Ce paramètre est calculé comme étant la somme de trois composantes dues aux aléas structurels, de données et de contrôle. Nous prendrons pour hypothèse de travail que les aléas de contrôle sont marginaux ; ce qui est vrai pour les applications de traitements du

signal et de l'image. Nous n'aurons plus alors que deux types d'aléas qui contribueront aux ruptures de pipeline.

Il ne reste alors qu'à analyser le code C, le placement des données en mémoire ainsi que le nombre de défauts de cache pour déterminer le nombre de ruptures de pipeline. Comme le nombre de ruptures de pipeline dépend également du processeur sur lequel va être exécuté le code, il est nécessaire d'étudier sa documentation pour connaître le coût de chaque aléa.

Méthode générale de la prédiction du PSR

Les aléas structurels

Le nombre de ruptures de pipeline dû aux aléas structurels est calculé en utilisant la formule suivante :

$$Nb\_RP_{AS} = \sum (AS \times Trésolution_{AS}) \qquad \textbf{(11)}$$

Nb_RP$_{AS}$: Nombre de ruptures de pipeline dû aux aléas structurels
AS : Aléas Structurels
T**résolution$_{AS}$** : Temps de résolution d'un aléas structurel en nombre de cycle processeur.

Les aléas de données

Le nombre de ruptures de pipeline dû aux aléas de données est calculé en utilisant la formule suivante :

$$Nb\_RP_{AD} = \sum (AD \times Trésolution_{AD}) \qquad \textbf{(12)}$$

Nb_RP$_{AD}$: Nombre de ruptures de pipeline dû aux aléas de données
AD : Aléas de Données
Trésolution$_{AD}$: Temps de résolution d'un aléas de données en nombre de cycle processeur

Les aléas de contrôle

Le nombre de ruptures de pipeline dû aux aléas de contrôle est calculé en utilisant la formule suivante :

$$Nb\_RP_{AC} = \sum (AC \times Trésolution_{AC}) \qquad \textbf{(13)}$$

Nb_RP$_{AC}$: Nombre de ruptures de pipeline dû aux aléas de contrôle
AC : Aléas de contrôle
Trésolution$_{AC}$: Temps de résolution des aléas de contrôle en nombre de cycle processeur.

Pour déterminer le nombre total de ruptures de pipeline généré par l'exécution de l'application, il suffit de la somme de toutes les contributions (aléas structurels + aléas de données + aléas de contrôle).

Pour mieux comprendre la prédiction du PSR à partir du code C, nous allons appliquer cette méthode sur un exemple de type filtrage à réponse impulsionnelle finie (FIR) exécutée sur le TMS320C6201. Nous comparerons pour chaque type d'aléas les valeurs prédites avec les valeurs obtenues à l'aide des outils Texas.

Méthode de prédiction du PSR appliquée au TMS320C6201

Les aléas structurels

Dans le cas du C62, le seul type aléa structurel possible est dû aux défauts de cache. Chaque trame (8 instructions) générant un défaut de cache génère également des ruptures de pipeline ; le temps nécessaire à ce DSP pour résoudre un défaut de cache est de 8 cycles processeurs.

Supposons que notre FIR soit constitué de 2506 trames d'instructions. La mémoire cache de ce processeur peut contenir au plus 2048 trames; ce cache étant à correspondance directe, chaque trame génèrera deux défauts de cache. Dans notre cas, 1492 trames génèreront des défauts de cache ; le nombre de ruptures de pipeline associé à ces défauts de cache peut être calculé avec la formule suivante :

$$Nb\_RP_{\gamma} = NbTrame \times Taccès$$
$$Nb\_RP_{\gamma} = 1492 \times 8 = 11936 \quad cycles$$

La valeur mesurée avec les outils TI (Texas Instruments) donne 12486 cycles de ruptures de pipeline soit une erreur de **4.25%** entre notre prédiction et la mesure.

Les aléas de données

Les aléas de données pour ce DSP sont de deux types : les aléas dus aux conflits de bancs mémoire interne et les aléas dus aux accès en mémoire externe. Nous allons calculer le nombre de ruptures de pipeline pour chacun de ces aléas.

Supposons que notre filtre utilisent 3607 échantillons, il devra donc avoir 3607 coefficients. Supposons également que ces coefficients et ces échantillons soient dans le même banc mémoire. Il y aura donc des conflits de bancs à chaque calcul. Dans le cas de ce DSP, chaque conflit de banc est levé en 1 temps de cycle, nous pouvons donc

facilement calculer le nombre de ruptures de pipeline associé à cet aléa en utilisant la formule suivante :

$$PSR_{CB} = Nb\_Conflits \times 1 = 3607 \quad cycles$$

La valeur fournie par l'outil TI est de 3699 cycles soit une erreur de **0.22%** entre notre prédiction et la mesure.

Supposons maintenant que tous les échantillons et tous les coefficients soient en mémoire externe. Il faudra alors faire deux accès en mémoire externe à chaque calcul. En étudiant la documentation du processeur, nous savons qu'il faut 16 temps de cycle pour charger une donnée via la mémoire externe. Nous pouvons alors calculer le nombre de ruptures de pipeline engendré par cet aléa en utilisant la formule suivante :

$$PSR_{\tau} = NbEXT \times 16 = 7214\ (3607\ éch + 3607\ coef) \times 16 = 115424 \quad cycles$$

L'outil TI nous donne dans ce cas une valeur de 112088 cycles soit une erreur de **2.98%** entre notre prédiction et la mesure.

Pour valider cette méthode, nous l'avons appliquée sur un certain nombre d'applications de traitement du signal et de l'image. Pour chaque application, nous donnons la valeur du PSR mesurée par l'outil TI, la valeur de notre prédiction ainsi que le pourcentage d'erreur. Les résultats obtenus sont donnés dans le tableau ci-dessous.

APPLICATION	PSR MESURE	PSR ESTIME	ERREUR
FFT	0.64	0.6043	**5.64%**
LMS	0.25	0.24	**4%**
DWT_1	0.0027	0.00269	**0.7%**
DWT _2	0.755	0.7131	**5.55%**
DWT_3	0.765	0.7259	**5.1%**
EFR_Vocoder	0.2252	0.219	**2.75%**

Tableau IV. 1 Prédiction du PSR sur des applications de traitement du signal

L'algorithme DWT-1 utilise une image 64*64 stockée en mémoire interne de données. L'algorithme DWT-2 et DWT-3 utilisent respectivement une image 64*64 et 512*512 stockées en mémoire externe.

Les résultats pour ces différentes applications montrent que notre méthode de prédiction est précise ; l'erreur maximale entre la valeur fournie par l'outil TI et nos prédictions est de **5,64%** et l'erreur moyenne est de **3,39%**.

Une fois que tous les paramètres algorithmiques ont été prédits, il ne reste plus qu'à les insérer dans les lois de consommation de la cible pour pouvoir estimer la consommation de courant et de puissance de l'application. L'estimation de l'énergie consommée n'est pas actuellement possible en utilisant cette méthode. En effet, pour pouvoir estimer l'énergie, il faut connaître le temps d'exécution de l'application or, actuellement ce temps n'est pas prédit par cette méthode.

Le temps nécessaire à la prédiction des paramètres algorithmiques est relativement faible puisque nous n'étudions pas le code C dans son ensemble mais, seulement les boucles qui le constituent. Le nombre de lignes de code à étudier est donc restreint. Dans le tableau ci-dessous nous montrons pour des applications concrètes le nombre total de lignes de code C, le nombre de lignes assembleur générées par le compilateur ainsi que le nombre de lignes de code étudié.

APPLICATIONS	Nombre total de lignes de code C	Nombre de lignes de code assembleur	Nombre de lignes de code étudié	
FFT 1024	77	408	10	13%
LMS bi voies 1024	30	408	4	13%
DWT 64*64	46	714	17	37%
EFR VOCODER	118	1323	37	31%
MPEG-1	2267	8488	30	1%

Tableau IV. 2 Nombre de lignes de code à étudier en utilisant la méthode C

En moyenne, avec cette méthode, nous devons étudier 19% du source C. Donc, même si pour l'instant cette méthode n'a pas été automatisée, il est possible de l'appliquer sur des algorithmes relativement conséquents. Nous verrons dans le chapitre application (chapitre V) les précisions obtenues lorsque cette méthode est utilisée pour prédire la consommation d'une application.

IV.3 Présentation de l'outil Soft_Explorer

Dans cette partie, nous allons présenter l'outil automatique qui a été réalisé au cours de cette thèse. Cet outil permet, entre autres, d'estimer la consommation de courant, de puissance et d'énergie d'une application en utilisant la code assembleur généré par le compilateur ou par le programmeur. Nous allons maintenant présenter les différentes fonctionnalités de cet outil.

IV.3.1 Estimation de la consommation

Cet outil utilise le code assembleur pour estimer la consommation d'une application. Nous avons vu que, pour réaliser cette estimation, il fallait déterminer les valeurs des paramètres algorithmiques du modèle de puissance de la cible.

L'estimation avec SoftExplorer nécessite l'utilisation de l'environnement de développement fourni par Texas Instruments. La méthode globale d'estimation est représentée sur la figure ci-dessous :

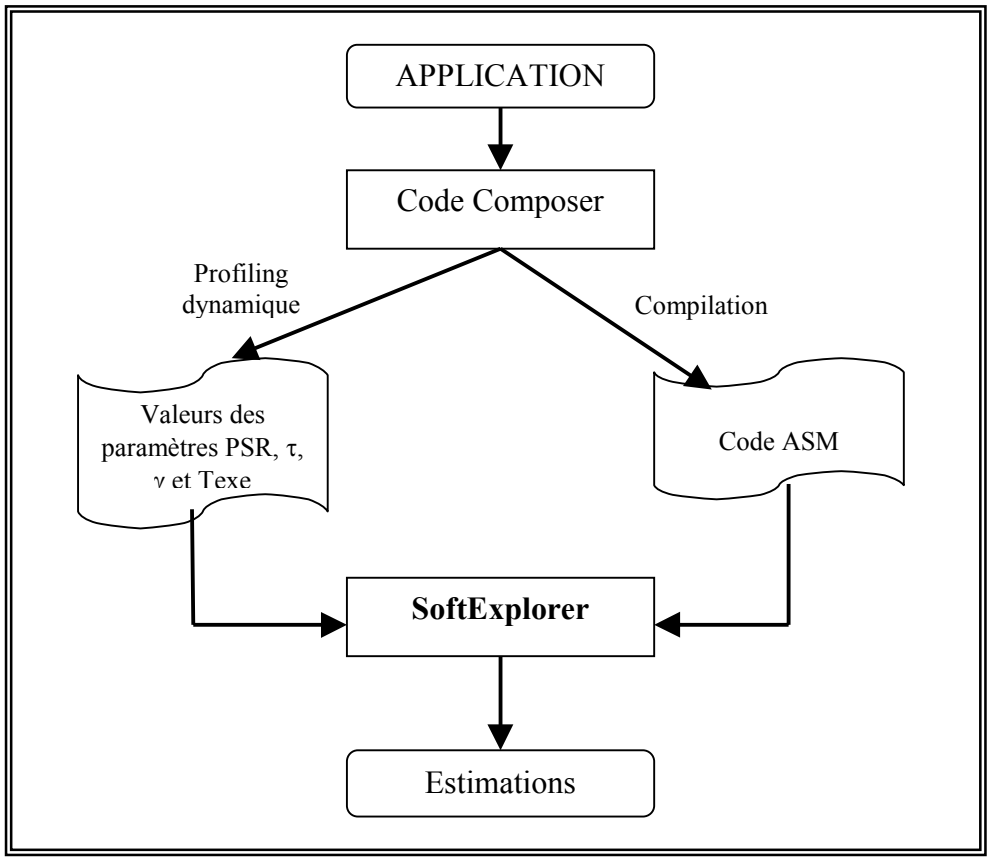

Figure IV. 10 Flot global d'estimation

L'utilisation de l'outil Code Composer permet, par profiling dynamique, de déterminer les valeurs du taux de ruptures de pipeline PSR, du taux d'accès en mémoire externe τ, du taux de défauts de cache γ et du temps d'exécution Texe de l'application. Ces différentes valeurs serviront d'entrée à SoftExplorer ; ce sera à l'utilisateur de saisir ces valeurs dans les champs de l'interface graphique.

Il ne reste alors qu'à déterminer le taux de parallélisme α et le taux d'unités de traitement utilisées β. Ces deux taux sont constitués de paramètres locaux et globaux. En effet, généralement un programme est constitué de plusieurs blocs linéaires d'instructions ; à chacun de ces blocs correspond un α et un β local. Le taux de parallélisme et le taux d'unités de traitement utilisées global d'un programme sont calculés en faisant une moyenne pondérée des différents paramètres locaux (cf. III.1.2.a).

Pour déterminer les paramètres locaux l'outil utilise un analyseur syntaxique qui étudie le code assembleur généré par le compilateur afin de repérer les diverses boucles du programme et de déterminer leur nombre d'itérations. Pour cela, l'analyseur syntaxique repère les commentaires "Kernel" généré par le compilateur lorsqu'il traduit les boucles en code assembleur. Le compilateur donne aussi, en commentaire, un certain nombre d'informations concernant ces boucles comme, par exemple le nombre d'itérations minimum et maximum. Ces informations nous permettent de déterminer le poids de chacune des boucles dans le programme ; cette information sera utilisée lors du calcul des paramètres globaux. Pour chacun des cœurs de boucle, l'analyseur calcule le nombre de paquets de chargement Nb_FP, le nombre de paquets d'exécution Nb_EP ainsi que le nombre d'unités de traitement utilisées Nb_UT. L'outil calcule alors les valeurs des paramètres de α et β locaux. Une fois tous les cœurs analysés, l'outil détermine les valeurs de α et β globales. SoftExplorer intègre alors ces valeurs dans la loi de consommation afin d'estimer la consommation de courant, de puissance et d'énergie de l'application. La loi de consommation est déterminée en fonction du mode mémoire du DSP choisi par l'utilisateur lorsque celui-ci entre les valeurs des paramètres de configuration. Nous avons développé cet analyseur en utilisant l'outil Parser Generator de Bumble-Bee Software.

La Figure IV.11 représente l'interface graphique à partir de laquelle l'utilisation entre les valeurs des paramètres de configuration et spécifie quel est le code à analyser.

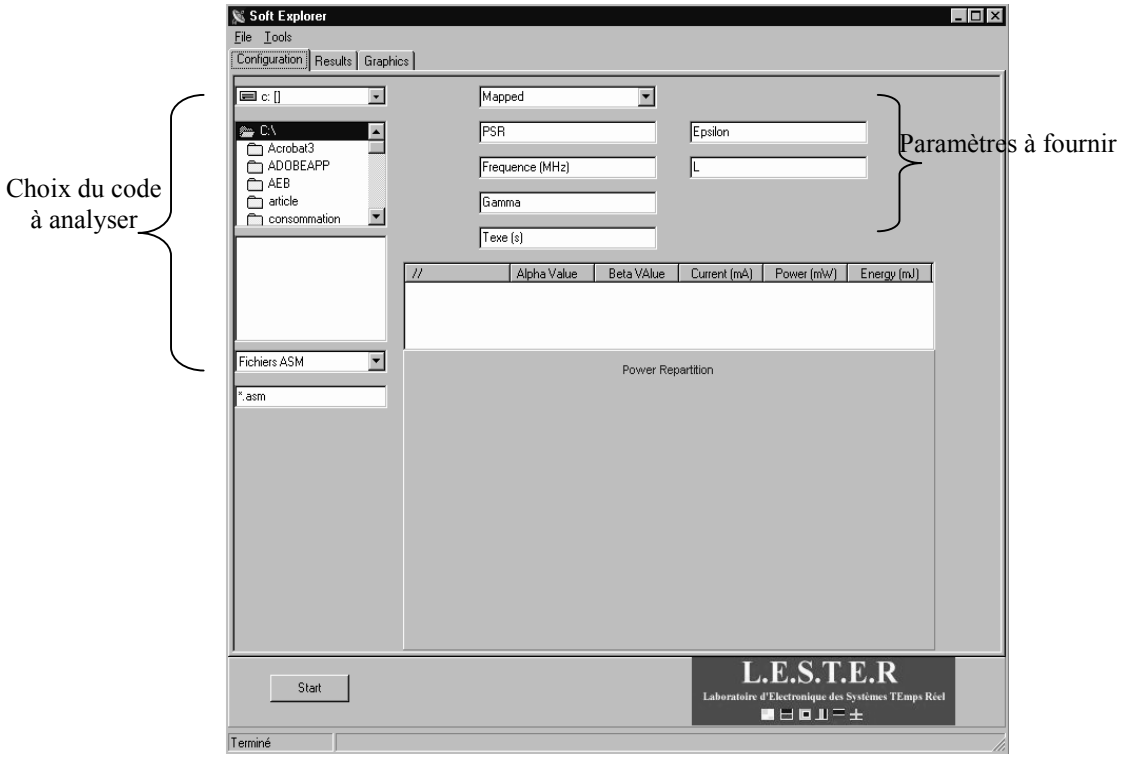

Figure IV. 11 Interface graphique de configuration de l'outil

L'outil permet de donner l'estimation de la consommation (courant, puissance, énergie) d'une application mais également sa répartition. C'est à dire que nous fournissons les consommations dues à l'unité de gestion des instructions, aux unités de traitement, à l'horloge interne du processeur ainsi que celle due au DMA. La Figure IV.12 montre l'interface graphique donnant l'estimation de la consommation ainsi que sa répartition.

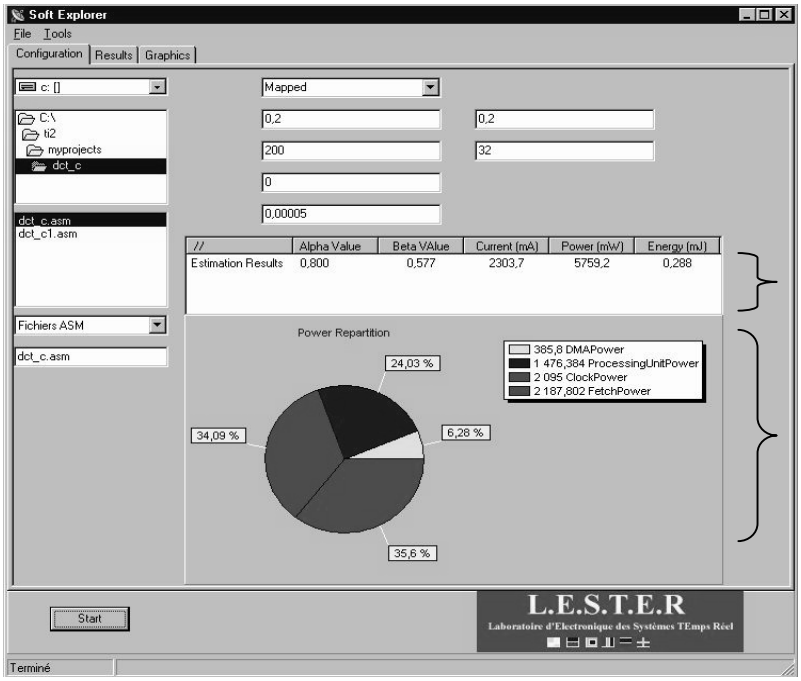

Figure IV. 12 Interface graphique donnant l'estimation de la consommation ainsi que sa répartition

IV.3.2 Etude des boucles d'un programme

Lors de l'estimation de la consommation d'un programme, il peut être intéressant de déterminer quels sont les « points chauds » d'un point de vue consommation. Les programmeurs étudient généralement leur code afin de déterminer quelles sont les parties du code qui pénalisent le temps d'exécution. La consommation devenant aujourd'hui un paramètre aussi important que le temps d'exécution pourquoi ne pas analyser le code afin d'en déterminer les parties les plus consommatrices?

Comme chacune des boucles du programme est étudiée grâce à l'analyseur syntaxique, nous pouvons facilement calculer la consommation de puissance de ces boucles ainsi que leur temps d'exécution. La figure ci-dessous montre les résultats obtenus sur une application utilisant dix boucles différentes.

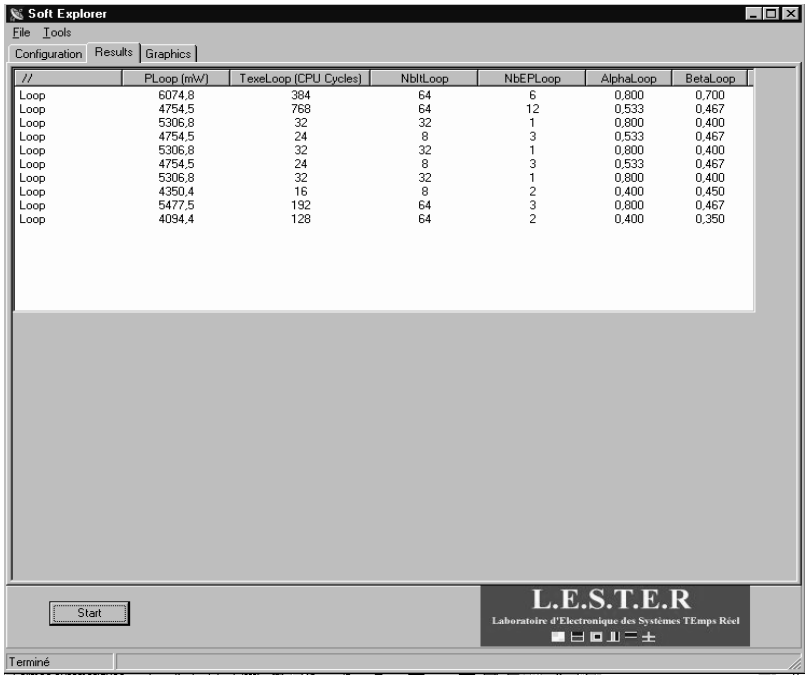

Figure IV. 13 Etude des boucles d'un programme

Cette interface permet à l'utilisateur de visualiser les caractéristiques de chaque boucle constituant le programme. Pour chacune d'elle l'outil fournit la puissance consommée, le temps d'exécution, le nombre d'itération, le nombre de paquets d'exécution ainsi que le taux de parallélisme et le taux d'unités de traitement utilisées. A l'aide de ces informations, il est alors possible de détecter les boucles du code les plus pénalisantes. Il pourra donc focaliser ces optimisations seulement sur les parties importantes au lieu d'essayer d'optimiser la totalité de son programme.

L'outil fournit également à l'utilisateur une représentation graphique de la consommation de puissance de chaque boucle ainsi que la durée de cette consommation. La Figure IV.14 montre le résultat pour l'exemple utilisé précédemment. Sur cet exemple, nous voyons bien que les boucles 4, 6 et 8 consomment plus de puissance que la boucle 2 mais que cette puissance est consommée sur un laps de temps court à l'inverse de la deuxième boucle. Si le programmeur veut optimiser sa consommation d'énergie, il devra alors en priorité améliorer le temps d'exécution de la deuxième boucle. Si au contraire, il veut optimiser sa consommation de puissance, il devra se focaliser plus particulièrement sur la première boucle.

Figure IV. 14 Représentation graphique de la consommation de puissance des boucles d'un programme

Une première version de cet outil est disponible sur le site Internet du laboratoire : http://lester.univ-ubs.fr:8080 rubrique Tools.

IV.4 Conclusion

Nous avons vu dans ce chapitre comment étaient déterminés les paramètres algorithmiques d'une application. Cette détermination peut se faire en utilisant soit le code assembleur soit le source C.

L'application de la méthode utilisant comme point d'entrée le code assembleur implique la possession des outils constructeurs de développement puisque le code source doit être compilé afin de permettre le calcul des différents paramètres.

En revanche, l'utilisation de la méthode C permet, elle, de s'affranchir des outils constructeurs puisque le code n'a pas besoin d'être compilé. Cette méthode repose sur l'utilisation de modèles de prédiction. Ces modèles permettent également de prendre en compte les différentes optimisations de compilation susceptibles d'être utilisées par le programmeur. Le principal avantage de cette méthode est d'estimer la consommation

dès les premières phases de conception. Son principal inconvénient est que nous prédisons seulement la puissance qui sera consommée mais, pour l'instant, pas l'énergie. En effet, notre prédiction du temps d'exécution de l'algorithme est entachée de trop d'erreur pour avoir une prédiction fiable (jusqu'à 74% d'erreur pour une application MPEG). La principale source d'erreur vient du fait que nous ne prédisons le temps d'exécution que pour les boucles du programme. Pour obtenir une prédiction fiable, il faudrait analyser tout le code C. Cela sera possible dès que l'automatisation de la méthode aura été réalisée car l'étude, même d'un code C, devient fastidieux pour des applications conséquentes.

Nous avons également présenté dans ce chapitre l'outil automatique SoftExplorer qui a été développé au cours de cette thèse. Cet outil est basé sur une analyse du code assembleur et permet non seulement d'estimer la consommation (courant, puissance et énergie) d'une application mais aussi de repérer les « points chauds » du point de vue de la consommation. Pour les applications dont l'exécution est non déterministe, l'outil utilise la borne maximale d'exécution afin de réaliser l'estimation de consommation.

Chapitre V

Validation sur des applications de TdSI

Dans la première partie de ce chapitre, nous validerons notre modèle de puissance sur plusieurs applications de traitement du signal et de l'image. Nous validerons également la méthode C sur ces mêmes applications. Les résultats fournis, pour la méthode utilisant le code assembleur, sont ceux déterminés à l'aide de l'outil automatique Soft_Explorer.

Dans la deuxième partie de ce chapitre, nous validerons notre méthode d'analyse de puissance au niveau fonctionnel (FLPA) en présentant les résultats obtenus pour un second processeur le TMS320C5510 de Texas Instruments.

Enfin dans une dernière partie, nous présenterons des résultats d'adéquation algorithme/architecture pour différents types d'applications.

V.1 Validation du modèle de puissance

Dans cette partie, nous allons valider le modèle de puissance que nous avons déterminé pour le DSP TMS320C6201 de Texas Instruments. Pour cela, nous comparerons des mesures physiques avec les résultats de nos estimations utilisant le code assembleur (avec l'outil Soft_Explorer) mais aussi le code C.

V.1.1 Validation du modèle au niveau assembleur

La validation du modèle de puissance de ce DSP a été effectuée sur différents types d'applications : des filtres numériques, des applications de compression d'images, une application de codage de la voix ainsi que sur une application de décompression vidéo.

Pour chacune de ces applications, nous avons réalisé des mesures physiques du courant moyen consommé. Connaissant la tension d'alimentation du processeur ainsi que le temps d'exécution de chacun des programmes, nous avons pu déterminer la puissance et l'énergie consommées. Les estimations de consommation ont été réalisées en utilisant l'outil Soft_Explorer que nous avons présenté dans le chapitre précédent.

Les premières applications que nous avons utilisées sont des filtres numériques : un filtre à réponse impulsionnelle finie (FIR) et un filtre adaptatif par la méthode des moindres carrés (LMS). Chacun de ces filtres à une longueur de 1024 points et le LMS gère en plus deux voies. Les filtres LMS sont utilisés pour réaliser de l'annulation d'écho acoustique appliqués dans le domaine de la téléconférence en bande élargie **[Bja93]**. Nous avons également appliqué notre méthode sur une transformée de Fourier rapide 1024 points.

Nous avons ensuite utilisé des applications de compression d'images : une transformée en cosinus discrets (64*64) et une transformée en ondelette paramétrable. Cette transformée peut compresser des images de taille variable ; le nombre de niveaux de résolution est également paramétrable.

Enfin, nous avons appliqué notre méthode sur deux applications plus conséquentes : un codeur de voix EFR (Enhanced Full Rate) pour GSM et une chaîne de décompression MPEG-1. Pour donner une idée de la complexité de ces applications,

l'EFR Vocoder est constitué de 1323 lignes de codes assembleur et le décodeur MPEG de 8488 lignes.

L'application codeur de voix pour GSM est issue de MediaBench qui est un recueil d'applications tests développé à l'Université de Californie **[Leec97]**. Les sources de la chaîne de décompression MPEG1 sont disponible sur Internet à l'adresse suivante : http://www.mpeg.org.

Le tableaux V.1 donne les résultats pour ces différentes applications. Pour chaque algorithme, nous donnons le placement des données en mémoire (DM), le mode mémoire utilisé (MM), les mesures et les estimations de puissance et d'énergie ainsi que l'erreur entre ces mesures et nos estimations.

Application	PM	MM	MESURES		ESTIMATIONS		Erreur (%)
			P (W)	E	P (W)	E	
FIR	Int	M	3.023	78µJ	2.94	75.8µJ	***-2.8***
FFT	Int	M	2.65	3.68mJ	2.677	3.72mJ	***+1***
LMS	Int	B	4.97	8.87J	5.144	9.18J	***+3.5***
LMS	Int	C	5.66	956mJ	5.558	939mJ	***-1.8***
DCT	Int	M	5.986	383mJ	5.898	377mJ	***+1.5***
DWT-1	Int	M	3.755	8.712mJ	3.83	8.89mJ	***+2***
DWT-2	Ext	M	2.55	23.46mJ	2.483	22.84mJ	***-2.63***
DWT-3	Ext	M	2.55	1.47J	2.465	1.42J	***-3.33***
EFR	Int	M	2.642	10.7mJ	2.625	10.63mJ	***-0.6***
MPEG	Int	M	5.82	235.1µJ	5.59	225.7µJ	***-4***

Tableau V. 1 Résultats d'estimation de la consommation de puissance

Les données sont placées soit en mémoire interne de données (Int) soit en mémoire externe (Ext). Afin de valider toutes les lois de consommation, nous avons également utilisé différents modes pour la mémoire de programme interne :

- M : Mapped
- B : Bypass
- C : Cache

Pour la transformée en ondelettes, nous avons utilisé 2 tailles d'images différentes ainsi que deux placements en mémoire. Les algorithmes DWT-1 et DWT-2 utilisent une image de 64*64 mais, l'image est en interne pour l'algorithme DWT-1 et en externe pour le DWT-2. L'algorithme DWT-3 utilise une image 512*512 placée en mémoire externe. Les mesures et les estimations que nous avons faites ne prennent pas en compte la consommation de la mémoire externe.

Les résultats montrent que l'erreur entre nos estimations (puissance et énergie) et les mesures est au **maximum de 4%** et que l'erreur **moyenne** est de **2,3%**. Nous

pouvons également remarquer que le mode mémoire a une forte influence sur la consommation d'énergie. En effet, dans le cas de l'application LMS nous avons une **augmentation** de la consommation de puissance de **8%** lorsque le mode CACHE est utilisé. En revanche, l'utilisation de ce mode entraîne une **diminution** de l'énergie de **89.77%.**

En conclusion, nous pouvons dire que notre modèle est validé aux vues des valeurs d'erreurs de nos estimations par rapport aux mesures physiques. De plus, nos estimations oscillent autour des mesures, nous sommes donc sûrs que toutes les consommations ont été prises en compte.

L'estimation avec l'outil Soft_Explorer est quasi instantanée ; dans la plupart des cas, moins d'une seconde CPU est nécessaire pour réaliser les estimations. A ce temps d'estimation il faut, toutefois, rajouter le temps nécessaire au profiling dynamique réalisé en utilisant Code Composer. Sur les applications utilisées, ce temps n'excède jamais plus de quelques secondes.

V.1.2 Validation du modèle au niveau C

Nous avons présenté dans le chapitre précédent une méthode permettant de prédire la consommation de puissance directement à partir du code C. L'avantage de cette méthode est qu'elle ne nécessite aucun outil constructeur puisque le code n'a pas besoin d'être compilé. Une telle méthode est intéressante lorsque le concepteur veut faire de l'adéquation algorithme/architecture. En effet, si la consommation de puissance est une de ses contraintes, il peut rapidement, à l'aide de cette méthode, prédire la consommation de son algorithme pour différentes cibles. Cela suppose toutefois qu'il ait en sa possession les modèles de puissance de ces cibles.

Le tableau V.2 donne les résultats de prédiction de la puissance pour des applications similaires à celles utilisées dans le paragraphe précédent et cela pour tous les modèles de la méthode. Ces prédictions sont comparées aux mesures physiques réalisées sur la cible. Comme précédemment, cette méthode a été appliquée sur le TMS320C6201. Le placement des données en mémoire ainsi que le mode mémoire pour chaque application est le même que dans le tableau précédent (Tableau V.1). Les mesures ont été réalisées pour des codes optimisés en performance.

Applications	Mesures	Estimations				
		SEQ	MAX	MIN	DATA	
	P(W)	P (W)	P (W)	P (W)	P (W)	Erreur (%)
FIR	4.5	2.745	4.725	3.015	4.725	***+5***
FFT	2.65	2.36	2.97	2.57	2.58	***-2.6***
LMS-1	4.97	5.02	5.12	5.07	5.12	***+3***
LMS-2	5.66	2.55	6	4.76	6	***+5.9***
DWT-1	3.755	2.82	4.24	3.27	3.53	***-6***
DWT-2	2.55	2.295	2.63	2.4	2.46	***-3.5***
DWT-3	2.55	2.27	2.61	2.37	2.45	***-3.9***
EFR	5.0775	2.54	5.636	3.86	5.13	***+1***
MPEG	5.823	2.665	6.380	3.927	5.353	***-8***
Erreur moyenne		***28.2%***	***7.3%***	***15.2%***	***4.4%***	

Tableau V. 2 Résultats de la prédiction de la consommation de puissance

Ces résultats montrent bien que le modèle de prédiction DATA est le plus précis puisque son **erreur maximum** est de **8%** et que son **erreur moyenne** est de **4.4%**. En effet, ce modèle de prédiction tient compte du placement des données en mémoire, ce qui n'est pas le cas pour les autres modèles. Or, Roy et al. ont montré que la consommation variait de façon importante en fonction du placement des données en mémoire **[Roy96]**.

L'intérêt des modèles MAX et MIN est de donner respectivement la borne supérieure et inférieure de consommation de l'application au concepteur. De plus, la méthode au niveau C doit pouvoir prendre en compte les différentes options de compilation susceptibles d'être utilisées. Pour montrer que tel est le cas, le tableau V.3 montre les résultats de prédiction de la consommation de puissance pour 3 applications : une transformée en ondelette 512*512, un codeur de voix EFR pour GSM et un décodeur MPEG1. Nous avons mesuré la consommation de puissance de chacune de ces applications pour 3 options de compilation : sans optimisation, avec optimisation de la taille du code et avec optimisation de la performance (temps d'exécution).

Ces résultats montrent bien qu'en fonction du type d'optimisation utilisée, le modèle de prédiction le plus précis n'est pas le même. En effet, si le code est compilé sans optimisation, alors le modèle le plus adéquat pour prédire la consommation de puissance sera le modèle SEQ avec une erreur moyenne de **2.86% ;** dans ce cas le code compilé est purement séquentiel. Si le code est compilé pour une optimisation de sa taille, alors le modèle le plus précis sera le modèle MIN (erreur moyenne **3.26%**) et

enfin si la compilation est faite pour optimiser les performances, le modèle le plus adéquat sera le modèle DATA (erreur moyenne **4.3%**).

ALGORITHME			MESURES	ESTIMATIONS			
	PM	**MM**	**P(W)**	**SEQ**	**MAX**	**MIN**	**DATA**
SANS OPTIMISATION							
DWT	Ext	M	**2.26**	***2.27*** (+0.4%)	2.61	2.37	2.45
EFR	Int	M	**2.37**	***2.54*** (+6.7%)	5.64	3.86	5.13
MPEG	Int	M	**2.71**	***2.67*** (-1.48%)	6.38	3.92	5.35
OPTIMISATION CODE							
DWT	Ext	M	**2.34**	2.27	2.61	***2.37*** (+1.28%)	2.45
EFR	Int	M	**3.64**	2.54	5.64	***3.86*** (+6%)	5.13
MPEG	Int	M	**4.02**	2.67	6.38	***3.92*** (-2.49%)	5.35
OPTIMISATION PERFORMANCE							
DWT	Ext	M	**2.55**	2.27	2.61	2.37	***2.45*** (-3.92%)
EFR	Int	M	**5.08**	2.54	5.64	3.86	***5.13*** (+0.98%)
MPEG	Int	M	**5.82**	2.67	6.38	3.92	***5.35*** (-8%)

Tableau V. 3 Résultats pour différentes options de compilation

Une application peut être constituée de plusieurs fonctions C. Chacune de ces fonctions n'a pas le même impact sur son exécution ; il peut donc être intéressant de compiler chacune de ces fonctions avec des options différentes. Il est alors indispensable d'estimer la consommation de puissance de ces fonctions en tenant compte de leurs options de compilation. En appliquant le modèle de prédiction adéquat sur les différentes fonctions C, nous sommes capables de prédire la consommation de puissance de chacune de ces fonctions mais également de la totalité de l'application. Ceci montre l'intérêt d'avoir plusieurs modèles de prédiction permettant de prendre en compte toutes les options de compilation.

V.2 Validation de la méthode d'analyse au niveau fonctionnel

Pour démontrer la validité de notre méthode d'analyse de la puissance au niveau fonctionnel (FLPA), nous allons exposer les résultats obtenus pour un autre processeur le TMS32C5510 de Texas Instruments. Ce processeur est la dernière génération de DSP basse consommation de ce constructeur ; il possède les caractéristiques suivantes **[Tex00a], [Tex00b], [Tex01], [Tex02]**:

- Tension d'alimentation de 1.7V
- Fréquence de fonctionnement paramétrable via une PLL jusqu'à 160MHz.
- 2 multiplieurs accumulateurs
- 2 UALs
- Instructions de taille variable pour densifier le code
- Mise en veille automatique des mémoires et des périphériques
- Un DMA et un EMIF pour effectuer les accès externes
- 3 modes mémoire pour la mémoire interne de programme : Mapped, Cache (associatif 2 voies), Bypass.

V.2.1 FLPA et modèle de puissance du TMS320C5510

Nous avons appliqué sur ce DSP la méthode FLPA présentée dans le chapitre III, ce qui nous a permis d'obtenir un découpage fonctionnel de l'architecture ainsi que tous ses liens de consommation. La FLPA de ce processeur est donnée sur la figure ci-dessous :

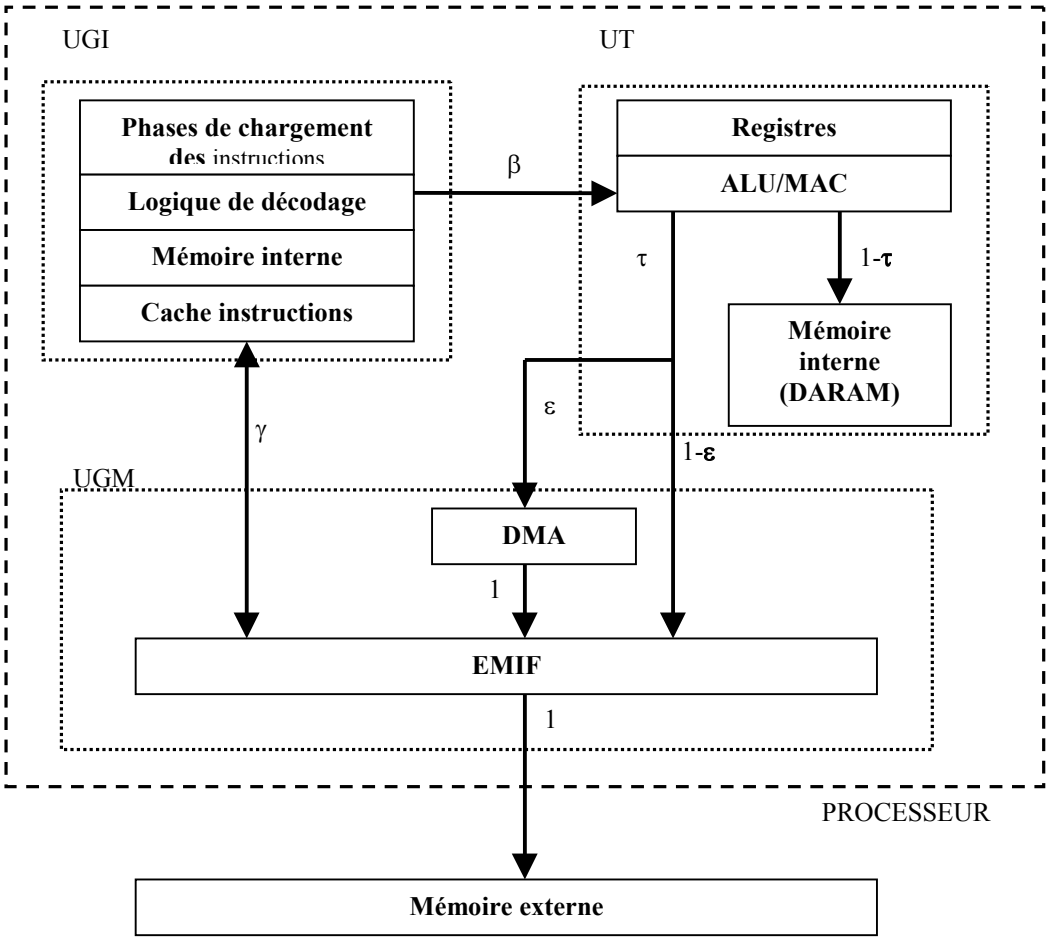

Figure V. 1 FLPA du TMS320C5510

L'analyse de puissance au niveau fonctionnel fait apparaître pour ce processeur 4 paramètres algorithmiques :

- β : taux d'unités de traitement utilisées.
- γ : taux de défaut de cache.
- τ : taux d'accès de données en mémoire externe.
- ε : taux d'accès de données via le DMA.

A ces 4 paramètres algorithmiques, il faut rajouter 6 paramètres de configuration :

- F : Fréquence d'horloge.
- MM : mode mémoire.
- DM : placement des données en mémoires.
- L : taille des données à charger via le DMA.
- NR : nombre de RAMSET utilisé.

- PM : Zones désactivées.

Le modèle de puissance du TMS320C55 peut donc être représenté sous la forme suivante :

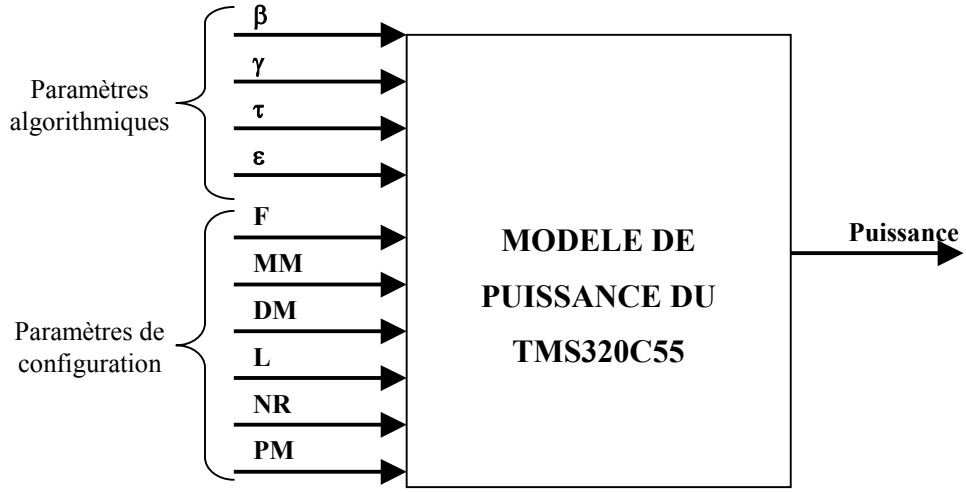

Figure V. 2 Modèle de puissance du TMS320C55

Ici, nous ne détaillerons pas l'obtention des lois de consommation ni la procédure de mesures pour ce processeur car la méthode est similaire à celle utilisée pour le TMS320C62. De plus amples informations concernant l'application de la FLPA ainsi que les lois de consommation pour ce processeur sont disponibles dans **[Far02]**.

Le temps de modélisation de ce processeur a été d'environ 1 mois ; cette étude a été réalisée par un stagiaire de DUT GEII **[Far02]** ce qui montre bien que cette méthode est rapide et aisée à mettre en œuvre.

V.2.2 Validation du modèle sur des applications

A l'heure actuelle, le modèle de ce DSP a été validé seulement sur 3 applications mais, celles-ci sont relativement complexes puisqu'il s'agit d'une transformée en ondelette, un codeur EFR de voix pour GSM et enfin pour un décodeur MPEG-1.

Pour chaque application, nous avons réalisé des mesures physiques sur la cible et des estimations en utilisant le code assembleur. Dans chacun des cas, le programme et les données sont en mémoire interne et la fréquence d'horloge est de 160MHz. Les résultats pour l'estimation de puissance sont donnés dans le tableau ci-après.

APPLICATION	MESURES		ESTIMATIONS		Erreur
	P (mW)	E (mJ)	P (mW)	E (mJ)	
DWT 64*64	370.6	1.71	385.7	1.78	***+4%***
EFR Vocoder	424.15	1.33	434.7	1.36	***+2.49%***
MPEG-1	385.9	1.58	400.3	1.64	***+3.7%***

Tableau V. 4 Résultats d'estimation de la consommation de puissance

Ces résultats montrent que notre estimation est précise puisque **l'erreur maximale** entre les mesures et nos estimations est de **4%** et que **l'erreur moyenne** est de **3.16%**.

En conclusion, l'analyse de puissance au niveau fonctionnel que nous avons développée est une approche intéressante puisque le temps d'obtention du modèle de puissance est court (de l'ordre d'un mois) et permet d'obtenir des estimations de consommation précises (environ 3% d'erreur pour le C55 et environ 2% d'erreur pour le C62). De plus, cette méthode est non seulement applicable sur des processeurs VLIW mais également sur des processeurs plus simples (voir les résultats obtenus pour le C55) ; elle permet donc de traiter différents types de cibles.

V.3 Adéquation algorithme/architecture

Dans cette partie, nous expliquerons comment notre méthode d'estimation de la consommation peut être utilisée pour réaliser de l'adéquation algorithme/architecture. En effet, jusqu'à présent, les contraintes utilisées lors de l'adéquation étaient des contraintes de temps et/ou de surface. Or, aujourd'hui la conception de systèmes requiert également la prise en compte de la consommation de puissance et/ou d'énergie. Il faut donc disposer d'une métrique précise permettant de prendre en compte ses facteurs. Nous nous proposons ici d'utiliser notre méthode d'estimation comme métrique pour la consommation.

Le but de l'adéquation est de faire correspondre au mieux l'architecture à l'algorithme utilisé ou l'algorithme à l'architecture choisie. Dans cette partie, nous étudierons ces deux possibilités pour montrer que notre méthode d'estimation est utilisable dans les deux cas.

V.3.1 L'architecture est fixée

L'architecture sur laquelle doit être exécutée l'application peut être fixée par le concepteur. En effet, il n'est pas rare que des sociétés utilisent pendant un certain nombre d'années le même processeur pour amortir le coût d'investissement. Dans ce cas, il faut toujours essayer d'adapter au mieux l'algorithme de l'application à cette cible.

Supposons que nous voulions réaliser une transformée en ondelettes sur une image et que nous ayons comme contrainte non seulement le temps d'exécution de cette application mais également sa consommation. Le taux de compression de l'image sera fonction du niveau de résolution de la transformée en ondelettes. Or, chaque niveau de résolution supplémentaire engendre un surcoût de consommation ; il faut donc pouvoir estimer rapidement la consommation en fonction de ce niveau.

Nous avons réalisé une telle étude pour une transformée en ondelettes sur une image 64*64. Le processeur cible pour cette étude est le TMS320C6201 fonctionnant à 200MHz. L'estimation de la consommation de puissance et d'énergie a été réalisée pour cinq niveaux de résolution en utilisant notre outil automatique. Le tableau V.5 expose les résultats que nous avons obtenus.

Niveau de résolution	Temps d'exécution (ms)	Puissance (mW)	Energie (mJ)
DWT 64*64			
1 Niveau	1.73	3785	6.55
2 Niveaux	2.19	3775 ***(-0.26%)***	8.27 ***(+26%)***
3 Niveaux	2.32	3765 ***(-0.53%)***	8.73 ***(+33%)***
4 Niveaux	2.35	3757 ***(-0.74%)***	8.83 ***(+35%)***
5 Niveaux	2.37	3757 ***(-0.74%)***	8.9 ***(+36%)***

Tableau V. 5 Résultats de consommation d'une DWT pour différents niveaux de résolution

Ces résultats montrent que la puissance consommée ne varie que très peu avec l'augmentation du nombre de niveaux de résolution (légère décroissance). Ceci s'explique par le fait que le nombre d'accès mémoire augmente ce qui entraîne une augmentation du nombre de ruptures de pipeline. Cette augmentation génère une diminution du parallélisme du programme donc de sa consommation de puissance. En revanche, l'augmentation du taux de ruptures de pipeline entraîne une augmentation du

temps d'exécution de l'application donc une augmentation de la consommation d'énergie.

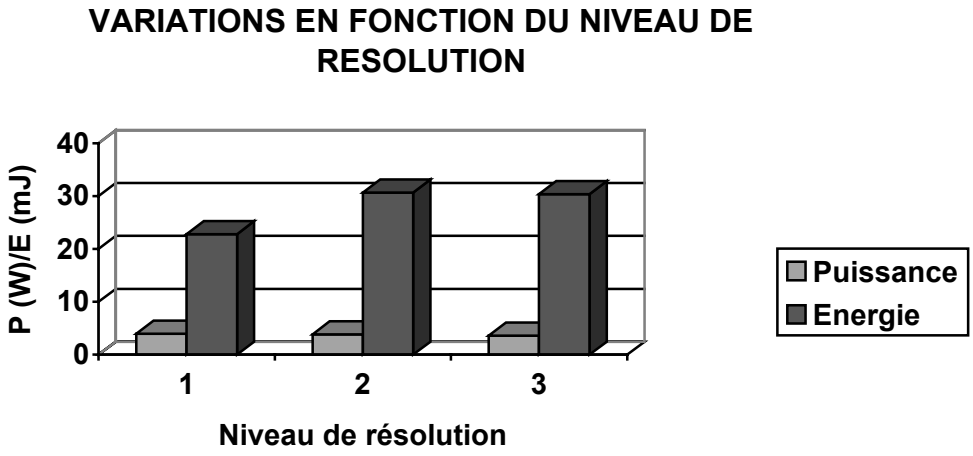

Figure V. 3 Variations de la puissance et de l'énergie en fonction du niveau de résolution

Ces résultats montrent bien que le choix du niveau de résolution de la transformée n'est pas trivial car l'évolution de la consommation en fonction du niveau de résolution n'est pas aisée à déterminer. En effet, nous avons pu voir que la consommation de puissance diminue légèrement avec l'augmentation du nombre de résolution. En revanche, l'énergie consommée, elle, augmente. La puissance et l'énergie n'évoluant pas dans le même sens, il faut donc prendre en compte la contrainte la plus forte pour le système.

La transformée en ondelettes pouvant travailler sur plusieurs tailles d'image, il faut également être capable de déterminer quelle est la taille la mieux adaptée aux contraintes de consommation. En effet, est-il plus intéressant du point de vue de la consommation de travailler sur une image entière ou de la segmenter en sous-images ? Répondre à cette question nécessite d'estimer la consommation de puissance de cette transformée en fonction de la taille de l'image. Nous avons réalisé une telle étude pour 4 tailles d'image différentes : 64*64, 128*128, 256*256 et 512*512. Nous avons également utilisé le niveau de résolution comme paramètre. Les résultats obtenus lors de cette étude sont présentés dans le tableau V.6.

Taille de l'image	Temps d'exécution	Puissance	Energie
1 Niveau de résolution			
64*64	1.73ms	3785mW	6.55mJ
128*128	27.65ms	2545mW	70.37mJ
256*256	110.19ms	2557.5mW	281.81mJ
512*512	439.93ms	2550mW	1.12J
2 Niveaux de résolution			
64*64	2.19ms	3775mW	8.27mJ
128*128	34.62ms	2552.5mW	88.37mJ
256*256	137.84ms	2562.5mW	353.21mJ
512*512	550ms	2555mW	1.4J
3 Niveaux de résolution			
64*64	2.32ms	3780mW	8.77mJ
128*128	36.38ms	2555mW	92.95mJ
256*256	144.8ms	2567.5mW	371.77mJ
512*512	577.77ms	2555mW	1.476J
4 Niveaux de résolution			
64*64	2.35ms	3797.5mW	8.92mJ
128*128	36.84ms	2552.5mW	94mJ
256*256	146.57ms	2562.5mW	375.58mJ
512*512	584.73ms	2557.5mW	1.495J
5 Niveaux de résolution			
64*64	2.37ms	3767.5mW	8.93mJ
128*128	36.97ms	2555mW	94.46mJ
256*256	147.05ms	2562.5mW	376.82mJ
512*512	586.54ms	2557.5mW	1.5J

Tableau V. 6 Résultats pour différentes tailles d'image et différents niveaux de résolution

Ces résultats montrent que la puissance consommée varie très peu avec le nombre de niveaux de résolution ; par contre, la taille de l'image a une influence. En effet, lorsque l'image a une taille de 64*64, celle-ci se trouve en mémoire interne alors que pour les autres tailles, l'image est en mémoire externe.

Le fait d'être en mémoire externe engendre des ruptures de pipeline qui modifient les valeurs intrinsèques des paramètres algorithmiques de l'application d'où la diminution de la puissance consommée. Ceci est également due au fait que nous ne prenons pas en compte la consommation de la mémoire externe. En revanche, il n'en est pas de même avec l'énergie. En effet, plus la taille de l'image est grande et plus l'énergie consommée est importante, ceci est dû au fait que le temps d'exécution de l'application augmente très fortement avec l'augmentation de la taille de l'image.

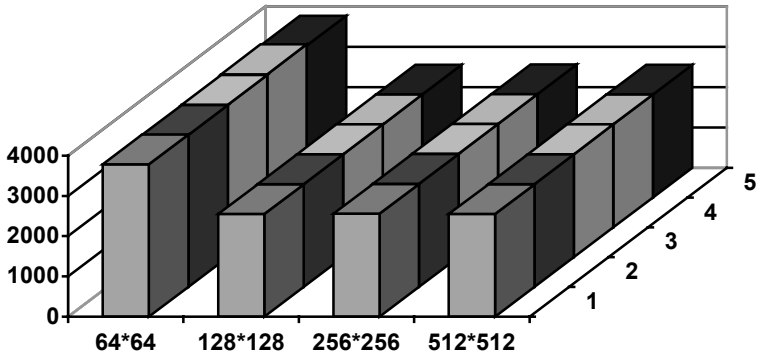

Figure V. 4 Variations de la puissance en fonction de la taille d'image et du niveau de résolution

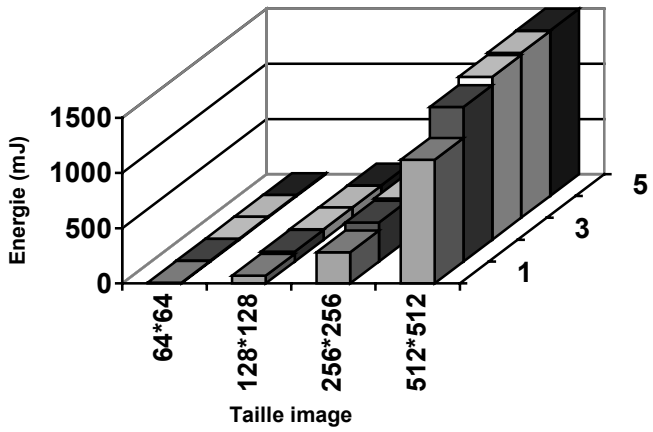

Figure V. 5 Variations de d'énergie en fonction de la taille d'image et du niveau de résolution

Lorsque l'image est en mémoire externe, le temps d'exécution est multiplié par 4 à chaque fois que la taille d'image est doublée. Ceci explique la forte croissance de l'énergie lorsque la taille de l'image augmente.

Le niveau de résolution n'est pas la seule possibilité d'adapter l'algorithme à l'architecture : nous pouvons également modifier cet algorithme tout en conservant ces propriétés. Dans l'étude suivante, nous allons reprendre l'application de transformée en ondelettes et ceci toujours pour la même cible et la même taille d'image. Dans le premier algorithme (Algo-1), les indices de boucles sont toujours les mêmes quel que soit le niveau de résolution. Pour cela, nous réarrangeons l'image de manière à ce que

les pixels à traiter soient toujours dans la partie supérieure gauche de l'image (cf. Figure V.6). Chaque pixel traité nécessite donc 4 accès mémoire: 2 accès pour le filtrage et 2 accès pour le réarrangement.

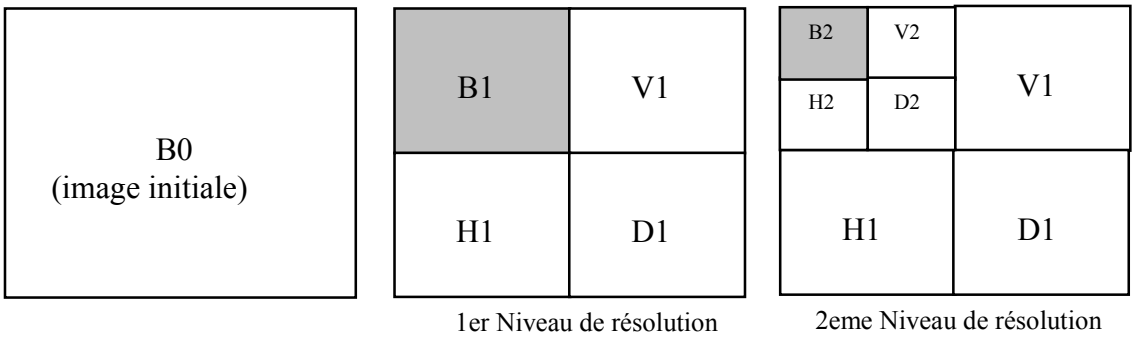

Figure V. 6 Décomposition d'une image par la transformée en ondelettes sur 2 niveaux

Pour l'algorithme modifié (Algo-2), les indices de boucles évoluent en fonction du niveau de résolution, ce qui permet d'éviter de faire un réarrangement de l'image à traiter. Il ne faut plus que 2 accès mémoire au lieu de 4.

Le tableau ci-dessous expose les résultats sur une image 64*64 pour ces deux algorithmes et ceci pour les trois niveaux de résolution.

Algorithme	Temps d'exécution (ms)	Puissance (mW)	Energie (mJ)
1 Niveau de résolution			
Algo-1	5.72	3982	22.78
Algo-2	1.73	3885 ***(-2.5%)***	6.72 ***(-70.5%)***
2 Niveau de résolution			
Algo-1	7.93	3865	30.65
Algo-2	2.19	3875 ***(+0.26%)***	8.49 ***(-72.3%)***
3 Niveau de résolution			
Algo-1	8.41	3611	30.37
Algo-2	2.32	3880 ***(+7.45%)***	9 ***(-70.67%)***

Tableau V. 7 Résultats pour 2 algorithmes de DWT et pour différents niveaux de résolution

Ces résultats montrent que l'Algo-2 consomme plus de puissance que l'Algo-1 sauf pour le niveau de résolution le plus faible. En revanche, l'énergie consommée par cet algorithme est très inférieure à celle consommée par l'algorithme non modifié (environ -70%). Si la contrainte la plus sévère est le temps d'exécution ou l'énergie consommée alors l'emploi de l'algorithme modifié est recommandé mais, si c'est la consommation de puissance qui est la plus critique alors l'emploi de l'algorithme non modifié est à privilégier.

V.3.2 L'algorithme est fixé

Nous avons montré dans la partie précédente qu'un algorithme pouvait être paramétré ou modifié pour s'adapter au mieux à une architecture fixe. Nous pouvons également avoir le cas inverse où l'algorithme est fixé et dans ce cas, il faut déterminer la cible architecturale la plus adéquate pour l'exécuter. Ce cas peut se présenter si l'application a déjà fait l'objet d'un développement ; il serait alors coûteux de refaire ce développement et de réaliser les mesures de consommation pour plusieurs architectures. Notre méthode peut alors être utilisée pour estimer rapidement la meilleure cible architecturale permettant de respecter les contraintes de consommation.

Nous avons réalisé une étude de ce type pour 4 algorithmes différents : un filtre FIR, une transformée en ondelettes, un codeur de voix EFR pour GSM ainsi que pour un décodeur MPEG-1 (ces algorithmes sont les mêmes que ceux utilisés précédemment). Nous allons comparer les résultats obtenus sur ces applications pour les deux modèles de puissance que nous avons développés au cours de cette thèse i.e. pour le TMS320C6201 et pour le TMS320C5510. Le tableau ci-après donne les résultats d'estimations de la consommation de puissance et d'énergie pour ces 4 applications.

Application	Cible	Temps d'exécution (µs)	Puissance (W)	Energie (µJ)
FIR	***C55***	55.46	0.465	25.79
	C62	25.8	2.358 ***(+407%)***	60.84 ***(+136%)***
DWT	***C55***	4615	0.386	1781.4
	C62	2320	3.83 ***(+892%)***	8885.6 ***(+399%)***
EFR	***C55***	3140	0.435	1366
	C62	4050	2.625 ***(+503%)***	10631 ***(+678%)***
MPEG-1	***C55***	4090	0.4	1636
	C62	40.37	5.59 ***(+1297%)***	225.67 ***(-86%)***

Tableau V. 8 Résultats pour différents processeurs

Ces résultats montrent que le C55 consomme toujours moins de puissance que le C62, jusqu'à **–892%**. Ceci s'explique par le fait que ce processeur est dit « basse consommation ». En effet, sa tension d'alimentation est de 1.7V au lieu de 2.5V pour le C62. De plus, sa fréquence d'horloge est inférieure à celle du C62 : 160MHz au lieu de 200MHz. Le nombre d'instructions générées par le compilateur du C62 est toujours plus grand que pour le C55 puisque le C62 possède une architecture LOAD/STORE. Le

compilateur doit donc rajouter du code permettant le chargement et l'enregistrement des données via ou vers la mémoire interne et/ou externe.

En revanche, nous pouvons remarquer que les temps d'exécution sur le C55 sont supérieurs à ceux du C62 sauf dans le cas de l'EFR Vocoder. Pour cette application, le code exécuté par le C62 génère un grand nombre de ruptures de pipeline ce qui entraîne un ralentissement. Comme le temps d'exécution est inférieur pour ce processeur, cela entraîne une diminution de sa consommation d'énergie. C'est pour cette raison que la surconsommation d'énergie est inférieure à la surconsommation de puissance.

Pour des applications nécessitant un grand nombre de calculs et peu de transferts mémoire comme le MPEG-1, le C62 est même moins consommateur d'énergie que le C55 qui est pourtant « Low Power ». Dans le cas de cette application, le compilateur du C62 peut utiliser au maximum le parallélisme de l'architecture ce qui explique la valeur de puissance élevée (5,59W). Par contre, le parallélisme important du code entraîne une diminution importante du temps d'exécution donc une énergie faible. Ces résultats montrent bien qu'il est difficile de choisir a priori quelle est la cible la mieux adaptée à une application. Il est donc indispensable de posséder une métrique rapide et fiable de la consommation pour permettre le choix le plus approprié.

V.4 Conclusion sur les résultats

Tout au long de ce chapitre, nous avons présenté un ensemble de résultats d'estimations et de prédictions de la consommation pour un certain nombre d'applications. Le spectre de ces applications était vaste puisque nous avons traité des algorithmes de filtrage numérique, de compression d'image, de codeur de voix ainsi qu'une application de décompression vidéo.

Dans un premier temps, nous avons validé notre modèle de puissance du TMS320C6201 sur ces applications et ceci pour deux niveaux d'abstraction : au niveau assembleur et au niveau C. Au niveau assembleur, notre modèle a été validé en utilisant l'outil automatique **Soft_Explorer** développé au cours de cette thèse. Ces résultats ont montré que notre modèle de puissance permettait d'estimer la consommation de puissance avec une erreur moyenne de :

- **2.3% au niveau assembleur**

- **4.4% au niveau C.**

Le niveau C nous permet de prédire la consommation de puissance engendrée par l'exécution d'une application et cela sans devoir compiler le code. Ceci évite donc au concepteur d'avoir à sa disposition les outils constructeurs pour toutes les cibles qu'il veut tester. Toutefois, il est nécessaire d'avoir à disposition les modèles de puissance de chacune des cibles ce qui requiert une analyse préalable. Un autre avantage de cette méthode est de permettre la prise en compte des options de compilation qui pourraient être utilisées dans la phase finale de conception.

Dans un second temps, nous avons validé notre méthode d'analyse de puissance au niveau fonctionnel (FLPA). Pour cela, nous avons réalisé la FLPA et déterminé le modèle de puissance pour un autre DSP : le TMS320C5510 de Texas Instruments. Les résultats d'estimation sur 3 applications (DWT, EFR Vocoder et MPEG-1) montrent que notre modèle peut estimer la consommation de puissance avec une erreur moyenne de 3% par rapport aux mesures physiques.

Enfin dans une dernière partie, nous avons montré que notre méthode pouvait être utilisée dans les processus d'adéquation algorithme/architecture. En effet, jusqu'à présent les métriques utilisées pour faire de l'adéquation étaient le temps et la surface. Or, aujourd'hui il est indispensable de prendre en compte la consommation lors de ce processus ; pour cela, il faut une métrique de consommation précise et rapide. Nos résultats ont montré que l'estimation de la consommation de puissance n'était pas un paramètre suffisant et que l'énergie devait également être prise en compte. En effet, c'est l'énergie qui a un impact sur la durée de vie des batteries alors que la puissance, elle, a un impact sur le système de refroidissement et la durée de vie du composant. Notre méthode permet d'estimer ces deux paramètres pour que le concepteur puisse trouver le meilleur compromis.

Chapitre VI

Conclusion générale et perspectives

VI.1 Conclusion générale

Les travaux développés au cours de cette thèse s'intègrent dans un projet global qui a pour but d'intégrer l'aspect consommation dans le flot de conception des systèmes sur puce (SoC).

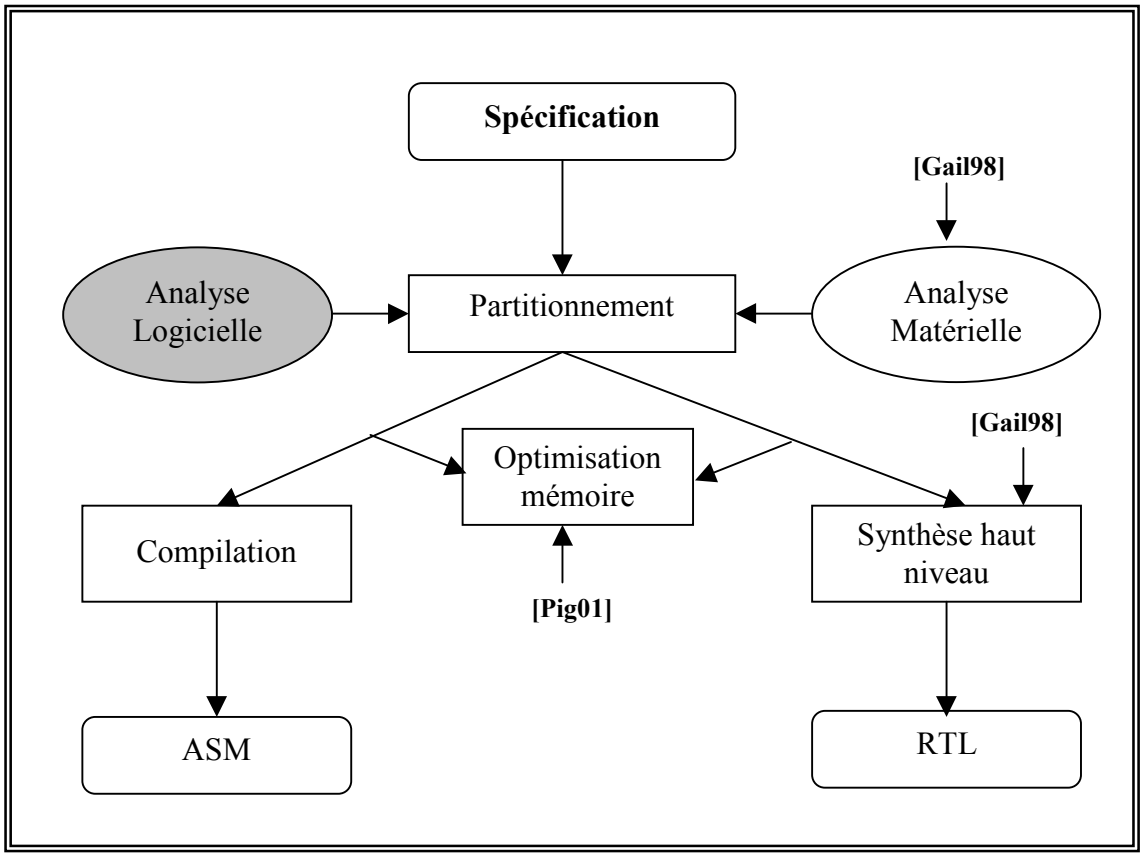

Figure VI. 1 Flot de conception des Soc

Durant cette thèse, nous nous sommes focalisés sur la partie analyse logicielle du point de vue de la consommation (les autres étapes du flot ont fait ou font l'objet de thèses). Le problème de l'estimation de la consommation est complexe puisqu'il faut modéliser non seulement le processeur mais également les interactions avec son environnement extérieur (ruptures de pipeline, défauts de cache etc.) ; les impacts de ces interactions sur la consommation deviennent prédominants du fait de l'utilisation de mécanismes architecturaux complexes. Nous devons aussi gérer la complexité de la méthode d'estimation ; le temps nécessaire à l'estimation quelle que soit l'application doit être le plus court possible.

De nombreuses méthodes d'estimations de la consommation existent surtout pour les processeurs (généraux ou spécialisés) mais la plupart d'entre elles se heurtent à un problème de complexité. En effet, ces méthodes sont généralement basées sur des approches ILPA où il est nécessaire de mesurer la consommation de tout le jeu d'instructions; la complexité du modèle varie alors de $O(N^2)$ pour des processeurs simples à $O(N^{2k})$ pour des architectures VLIW ou superscalaires. De plus, ces méthodes ne prennent généralement pas en compte la consommation due aux interactions avec l'environnement extérieur. Ces méthodes utilisent des tables de consommation pour réaliser l'estimation d'une application ; à chaque instruction ou classe d'instructions correspond une valeur de consommation dans la table. L'estimation se fait par analyse du code assembleur et pour chaque instruction il faut rechercher sa consommation en parcourant la table ou les tables ; le temps d'estimation est donc relativement important. Même les approches utilisant une modélisation fonctionnelle à grain fin n'arrivent pas à résoudre de façon satisfaisante le problème de complexité.

Le premier objectif de cette thèse était donc de développer une méthodologie d'estimation simple et précise. Nous avons montré dans ces travaux une nouvelle méthode basée sur une approche fonctionnelle à gros grains (FLPA). La FLPA permet de modéliser le comportement du processeur mais également les interactions avec l'extérieur. Pour cela nous réalisons une étude de l'architecture du point de vue de la consommation. La seule information nécessaire à cette étude est un schéma bloc de la cible. Après étude, nous obtenons un modèle utilisant 2 types de paramètres: des paramètres algorithmiques et de configuration. Notre approche permet de réduire de façon significative le temps d'obtention du modèle puisqu'il nous a fallu 45 jours pour modéliser un processeur VLIW d'ordre 8 alors que 108 jours ont été nécessaires à Sami et al. **[Sami02]** pour modéliser un processeur VLIW d'ordre 4 avec une approche fonctionnelle à grain fin.

Le deuxième objectif était valider notre modèle de consommation sur des applications complexes. Pour cela, il était indispensable de développer un outil automatique. L'outil SoftExplorer est basé sur un analyseur syntaxique qui, à partir du code assembleur, calcule les différentes valeurs des paramètres algorithmiques du modèle de puissance. L'utilisateur doit seulement fournir les valeurs des paramètres de configuration et spécifier le code à estimer en utilisant l'interface graphique de l'outil. SoftExplorer permet également de faire du "profiling" statique de l'application d'un

point de vue consommation. Nos résultats d'estimations sur des différents types d'applications de traitement du signal et d'images (FIR, LMS, FFT, DCT, DWT, EFR Vocoder, MPEG1) par rapport aux mesures réalisées montrent que:

- pour le TMS320C6201 **l'erreur moyenne est de 2,3% et au maximum de 4%**
- pour le TMS320C5510 **l'erreur moyenne est de 3,16% et au maximum de 4%**

Le temps nécessaire à l'estimation de la consommation de ces applications n'excède pas 1 seconde de temps CPU en utilisant notre outil. Le fait d'avoir modélisé deux processeurs différents permet de valider la méthodologie que nous avons développée.

Un autre objectif de cette thèse était de montrer que le niveau d'abstraction du point d'entrée du modèle pouvait être relevé. Jusqu'à présent le point d'entrée généralement utilisé était au mieux le code assembleur. Nous devons donc passer par une phase de compilation ce qui génère une latence supplémentaire. De plus, lors de la phase d'adéquation algorithme/architecture ce niveau d'abstraction nécessite d'avoir à disposition tous les outils de développement des cibles à analyser. Nous avons montré dans ces travaux qu'il était possible d'utiliser directement le code écrit en langage C pour réaliser l'estimation sans passer par une phase de compilation. A ce niveau d'abstraction, nous ne calculons plus les différentes valeurs des paramètres algorithmiques de notre modèle mais nous les prédisons. La plus grande difficulté, lors de la prédiction des valeurs de ces paramètres, étaient de prendre en compte les différentes optimisations pouvant être utilisées par le programmeur lors de la compilation. Pour résoudre ce problème, nous avons développer 4 modèles de prédictions.

- **Le modèle Max**: Exploitation maximale de l'architecture. Il permet de donner une borne maximale de consommation ainsi que la borne minimale du temps d'exécution.
- **Le modèle Min**: Les chargement et les enregistrements ne sont jamais réalisés en parallèle en revanche les opérations de calculs peuvent être faites en parallèle. Ce modèle donne la borne minimale de consommation. Il

permet également de prendre en compte les optimisations au niveau de la taille du code.

- **Le modèle DATA**: Ce modèle est le seul à prendre en compte le placement des données. En effet, les chargements et les enregistrements peuvent être réaliser en parallèle si et seulement si ils se font sur des structures différentes. Ce modèle permet de prendre en compte les optimisations en performance.
- **Le modèle SEQ**: Toutes les instructions sont réalisées de façon séquentielle. Ce modèle est intéressant pour les cibles scalaires et permet de modéliser un code compilé sans aucune optimisation.

Nous avons validé cette nouvelle approche en prédisant la consommation de plusieurs applications de traitement du signal et de l'image. Nos résultats montrent que **l'erreur maximale de prédiction par rapport aux mesures est de 8%**. La principale limitation de cette méthode est que l'énergie consommée par une application n'est pas prédite; nous ne prédisons que la puissance. En effet, la prédiction du temps d'exécution est pour l'instant entachée de trop d'erreur (jusqu'à 74%) pour permettre une prédiction de l'énergie. La résolution de ce problème passe par la réalisation d'un outil automatique permettant d'analyser la totalité du code et non plus seulement ces boucles.

Nous avons également montré dans cette thèse que notre méthode d'estimation pouvait servir de métrique pour les processus d'adéquation algorithme/architecture. En effet, si l'utilisateur possède une librairie de cibles architecturales conséquente alors, il peut aisément déterminer quelle est la cible la mieux adaptée, d'un point de vue consommation, à l'exécution de son application. Pour cela, il doit toutefois prendre en compte non seulement la consommation de puissance mais également l'énergie car elle ne varie pas forcément dans le même sens. Les résultats de notre étude pour les deux cibles dont nous disposions montre qu'il n'est pas trivial de déterminer a priori quelle est la meilleure cible architecturale. En effet, pour une application comme le décodeur MPEG-1, les résultats montrent que la cible la mieux adaptée est le C62 et non le C55 bien que ce dernier soit un DSP basse consommation. Ces résultats montrent que notre méthodologie d'estimation peut être intégrée comme métrique de consommation lors de la phase d'adéquation Algorithme/Architecture.

VI.2 Perspectives

Un certain nombre de perspectives peut être dégagé et énoncé à l'issu de la présentation de ces travaux. Ces perspectives peuvent être définies à court, moyen et long terme.

A cours terme, Nous devrons appliquer notre méthode FLPA sur des processeurs généraux pour bien montrer que notre approche est générale et qu'elle n'est pas seulement réservée aux processeurs de traitements du signal. Certains processeurs généraux peuvent utiliser des architectures VLIW ou des architectures superscalaires. Nous avons montré dans ces travaux que notre méthode était applicable sur une architecture VLIW mais qu'en est-il pour une architecture superscalaire? Ces deux architectures sont capables d'exécuter plusieurs instructions en parallèle mais dans le cas d'architecture VLIW, l'ordonnancement des instructions est réalisé par le compilateur (ordonnancement statique). En revanche, pour les architectures superscalaires l'ordonnancement des instructions est réalisé lors de l'exécution du code (ordonnancement dynamique). Avec ce type de processeur, nous devrons prendre en compte la possibilité d'exécution spéculative. Comme notre méthode n'est pas basée sur une approche ILPA, nous ne caractérisons pas le jeu d'instructions ni les effets inter-instructions. Le fait que l'ordre d'exécution des instructions diffère entre le code compilé et le code exécuté n'est donc pas un problème pour notre méthode. La modélisation de tels processeurs devrait être possible en utilisant notre méthode FLPA.

En revanche, les applications pouvant être exécutée sur les processeurs généraux ne sont pas forcément des applications requérant du calcul intensif mais du contrôle intensif. Nous pourrons donc tester si notre méthodologie est capable d'estimer d'autres applications que celles de types calculs intensifs.

A moyen terme, nous devrons automatiser la méthode développée au niveau C car elle devrait s'avérer comme un atout indéniable dans l'estimation de systèmes sur puce. Il faudra pour cela résoudre le problème de prédiction du temps d'exécution mais, ceci devrait être résolu dès lors que l'automatisation sera réalisée. Nous pourrions étudier comment notre méthode d'estimation pourrait être appliquée sur des cœurs de processeurs et/ou des Ips. En effet, de plus en plus les systèmes sont intégrés sur une

même puce (SoC) et ces SoCs sont de fait hétérogènes. Il faudra donc étudier comment notre méthode pourrait être généralisée à tous types d'architectures. Cela nécessitera de rajouter des paramètres à notre modèle initial puisque typiquement la consommation dépendra de la technologie employée. Il sera également indispensable de développer un modèle de consommation pour les mémoires puisque les applications actuelles requièrent des tailles mémoires conséquentes.

Enfin à long terme, notre méthode d'estimation de la consommation pourrait intégrer dans l'étape de partitionnement du flot de conception des Socs. Elle pourrait être utiliser comme métrique de consommation ; ce paramètre s'ajouterait aux paramètres temps et surface afin de choisir si une fonction doit être réalisée de façon logicielle ou matérielle.

Concrètement, nous pourrions envisager d'intégrer notre méthodologie dans des outils de Co-Design comme Design Trotter actuellement développé au sein du laboratoire. En effet, cet outil ne prend en compte, pour l'instant, que des contraintes de temps et de surface. Or, la consommation devient aujourd'hui un paramètre incontournable et il devient impératif que les outils de Co-Design intègrent une métrique de consommation. Nous avons montré dans cette thèse que notre méthode pouvait être utilisée dans le processus d'adéquation Algorithme/Architecture.

Bibliographie Personnelle

Publications en revues internationales

SPRINGER VERLAG : Lecture Notes in Computer Science

- **2002** : **Power Consumption Estimation of a C Program for Data-Intensive Applications**

  LNCS volume 2451 septembre 2002 ISBN 3-540-44143-3

  http://link.springer-ny.com/link/service/series/0558/tocs/t2451.htm

Eric Senn, Nathalie Julien, Johann Laurent, Eric Martin

- **2002 : Power Estimation of a C Algorithm Based on a Functional-Level Power Analysis of a Digital Signal Processor**

  http://link.springer-ny.com/link/service/series/0558/tocs/t2327.htm

LNCS volume 2327 mai 2002 ISBN 3-540-43674-X

Nathalie Julien, Johann Laurent, Eric Senn, Eric Martin

- **2002 : Power Consumption Estimation of a C Algorithm : A New Perspective for Software Design** (à paraître)

  Johann Laurent, Nathalie Julien, Eric Senn, Eric Martin

- **2002 : High Level Energy Estimation for DSP Systems** (à paraître)

  Johann Laurent, Eric Senn, Nathalie Julien, Eric Martin

IEEE TCCA Newsletter

- **2001 : High Level Power Analysis for embedded DSP Software**

  http://computer.org/tab/tcca/NEWS/jan2001/index.html

  TCCA Newsletter janvier 2001

  Johann Laurent, Nathalie Julien, Eric Senn, Eric Martin

Publications en conférences internationales

- **2002 : Estimation de la consommation d'un algorithme C intégrant l'architecture cible : une aide efficace pour la conception système**

Johann Laurent, Nathalie Julien, Eric Senn, Eric Martin Conférence IEEE JFAAA 16-18 décembre 2002 Monastir Tunisie (actes à paraître).

- 2002 : **Power Consumption Estimation of a C Program for Data-Intensive Applications**

Eric Senn, Nathalie Julien, Johann Laurent, Eric Martin Conférence IEEE PATMOS pp332- 341 11-13 septembre 2002 Séville Espagne.

- **2002 : Algorithmic Power Consumption Estimation : an Efficient Guide for the Co-Design**

http://www.acm.org/sigs/sigda/daforum/program.html

Johann Laurent Conférence IEEE/ACM **DAC, Ph D Forum** sélectionné pour un « travel grant » 10-14 juin 2002 Nouvelle Orléans USA.

- **2002 : Power Estimation of a C Algorithm on a VLIW Processor**

Nathalie Julien, Eric Senn, Johann Laurent, Eric Martin Conférence IEEE WCED worshop de la conférence ISCA 26 mai 2002 Anchorage Alaska USA.

- **2002 : Power Consumption Estimation of a C Algorithm : A New Perspective for Software Design**

Johann Laurent, Eric Senn, Nathalie Julien, Eric Martin Conférence ACM LCR pp67-72 22-23 mars 2002 Washington USA.

- **2002 : Power Estimation of a C Algorithm based on a Functional Analysis of a Digital Signal Processor**

Johann Laurent, Nathalie Julien, Eric Senn, Eric Martin Conférence ISHPC 15-17 mai 2002 Kansai Japon.

- **2001 : High Level Energy Estimation for DSP Systems**

Johann Laurent, Eric Senn, Nathalie Julien, Eric Martin Conférence IEEE PATMOS pp3.1.1-3.1.10 26-28 septembre 2001 Yverdon les Bains Suisse.

- **2001 : High Level Power Estimation based on a Functional Analysis for Embedded DSP**

Johann Laurent, Nathalie Julien, Eric Senn, Eric Martin Conférence IEEE/ACM IWLS pp168-172 13-15 juin 2001 Lake Tahoe USA.

- **2000 : High Level Power Estimation for DSP**

Johann Laurent, Nathalie Julien, Eric Martin Conférence IEEE/ACM SAME pp112-117 25-26 octobre 2000 Sophia Antipolis France.

- **2000 : High Level Power Analysis for Embedded DSP Software**

Nathalie Julien, Johann Laurent, Eric Martin Conférence IEEE/ACM MEDEA worshop de la conférence PACT 19 octobre 2000 Philadelphie USA.

Publications en conférence nationale

2002 : Estimation de la Consommation d'un Algorithme C par Analyse Fonctionnelle

Johann Laurent, Nathalie Julien, Eric Senn, Eric Martin colloque GDR CAO pp165-168 15-17 mai 2002 Paris France.

Toutes ces publications sont accessibles sur le serveur web du laboratoire : http://lester.univ-ubs.fr :8080

Bibliographie

[Ben01] L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria, R. Zafalon
« *A Power Modeling and Estimation Framework for VLIW-Based Embedded Systems* »
In Proceeding of PATMOS 2001

[Bja93] E. Bjarnason, E. Haensler, M. Rupp
« Acoustic Echo Control : Advances in Algorithm Techniques »
In Proceedings of 3rd International Workshop of Acoustic Echo Control 1993

[Bona02] A. Bona, M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, R. Zafalon
« *Energy Estimation and Optimization of Embedded VLIW Processors based on Instruction Clustering* »
In Proceeding of DAC 2002.

[Brand00a] C. Brandolese, W. Fornacieri, F. Salice, D. Sciuto
« *An Instruction Level Functionnality-Based Energy Estimation Model for 32 bits Microprocessor* »
In Proceeding of DAC 2000 p346.

[Brand00b] C. Brandolese, W. Fornacieri, F. Salice, D. Sciuto
« *Energy Estimation for 32 bits Microprocessor* »
In Proceeding of CODES 2000 p24.

[Bur01] T. Burd
« *General Processor Infomation* »
Janvier 2001.

[Chak99] C. Chakrabarti & D. Gaitonde
« *Instruction Level Power Model of Microcontrollers* »
In Proceeding of ISCA 1999.

[DeMic00] G. DeMicheli & L. Benini
« System Level Power Optimization : Techniques and Tools »
Tutorial Date 2000.

[Far02] G. Fargeas
« *Mesures et Estimation de la Consommation sur DSP Texas Instruments TMS320C55* »
Rapport de stage de DUT GEII juin 2002.

[Gai98] S.Gailhard
« Conception d'architectures à faible consommation »
Mémoire de thèse décembre 1998.

[Gebotys98] C. Gebotys & R. Gebotys
« *An empirical Comparison of Algorithmic, Instruction and Architectural Power Prediction Models for High Performance Embedded DSP* »
In Proceeding of ISLPED 1998.

[Gebotys99] C. Gebotys & R. Gebotys
« *Designing for Low Power in Complex Embedded DSP Systems* »
In Proceeding of Hawaï International Conference on Systems Sciences 1999.

[Hen96] J. Hennessy & D. Patterson
« *Architecture des ordinateurs : une approche quantitative* »
Edition International Thomson Publishing France 1996.

[Klass98] B. Klass, D. Thomas, H. Schmit, D. Nagle
« *Modeling Inter Instruction Energy Effects in a Digital Signal Processor* »
In Proceeding of Power Driven Micro Architecture Workshop ISCA 1998.

[Lee97] M. Lee, V. Tiwari, S. Malik, M. Fujita
« *Power Analysis and Minimization Techniques for Embedded DSP Software* »
In IEEE Transaction on VLSI Systems, Vol 5 N°1 mars 1997.

[LeeC97] C. Lee, M. Potkonjak, W. Mangione-Smith
« MediaBench : A Tool for Evaluating and Synthesizing Multimedia and Communication Systems »
In Proceedings of Micro 30 1997. (http://www.cs.ucla.edu/~leec/mediabench)

[Len01] F. Le Ninan
« *Mesure de la Consommation sur Processeurs de Traitement du Signal* »
Rapport de stage de DUT GEII 2001.

[Mar96] E.Martin & J.L. Philippe
« *Ingénierie des Systèmes Microprocesseurs : Application au Traitement du Signal et de l'Image* »
Edition Masson, CENT, ENST 1996.

[Macii98] E. Macii, M. Pedram, F. Somenzi
« *High Level Power Modeling, Estimation and Optimization* »
In IEEE Transaction on CAD, Vol 17 N°11 pp1061-1079 1998.

[Pig01] S. Pignolo
« Optimisations de la consommation des applications temps-réel embarquées »
Mémoire de thèse janvier 2001

[Pur96] D. Pursley
« *A Gate Level Simulator for Power Consumption Analysis* »
M.S Thesis Carnegie Mellon University 1996.

[Rab96] J.M. Rabaey & M. Pedram
« *Low Power design Methodologies* »
Edition Kluwer Academic Publishers 1996.

[Roy96] K. Roy & M.C. Johnson
« *Software Design for Low Power* »
In NATO Advanced Study Institute on Low Power Design in Deep Submicron Electronics August 1996 NATO ASI Series.

[Sami00a] M. Sami, D. Sciuto, C. Silvano, Z. Zaccaria, R. Zafalon
« *Power Exploration for Embedded VLIW Architecture* »
In Proceeding of ICCAD 2000 p498.

[Sami00b] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria
« *Instruction-Level Power Estimation for Embedded VLIW Cores* »
In Proceeding of CODES 2000 p34.

[SIA01] Semiconductor Industry Association
« *The National Technology Roadmap for Semiconductor* »
Janvier 2001.

[Sinha01] A. Sinha & A. Chandrakasan
« *JouleTrack- A Web Based Tool for Software Energy Profiling* »
In Proceeding of DAC 2001 p220.

[Steinke01] S. Steinke, M. Knauer, L. Wehmeyer, P. Marwedel
« *An Accurate and Fine Grain Instruction Level Energy Model Supporting Software Optimizations* »
In Proceeding of PATMOS 2001.

[Tex98] Kile Castille
« *Consumption Summary* »
Juin 19998.

[Tex99a] Texas Instruments
« *TMS320C6000 CPU and Instruction Set Reference Guide* »
Literature Number : SPRU189D mars 1999.

[Tex99b] Texas Instruments
« *Technical Brief* »
Literature Number : SPRU197D février 1999.

[Tex00a] Texas Instruments
« *TMS320C55 Technical Overview* »
Literature Number : SPRU393 février 2000.

[Tex00b] Texas Instruments
« *TMS320C55 Functional Overview* »
Literature Number : SPRU312 juin 2000.

[Tex01] Texas Instruments
« *Code Composer Studio Getting Started Guide* »
Literature Number : SPRU509C novembre 2001.

[Tex01b] Texas Instruments
« *TMS320C55 CPU Reference Guide* »
Literature Number : SPRU317D juin 2001.

[Tex02] Texas Instruments
« *Instruction Cache Reference Guide* »
Literature Number : SPRU576a février 2002.

[Tiw94] V. Tiwari, S. Malik, A. Wolfe
« *Power Analysis of Embedded Software : A First Step Towards Software Power Minimization* »
In IEEE Tansaction on VLSI Systems décembre 1994.

[Tiw96] V. Tiwari, S. Malik, A. Wolfe
« *Instruction Level Power Analysis and Optimization of Software* »
In Journal of VLSI Signal Processing 1996 Kluwer Academic Publishers.

[Tiw96b] V. Tiwari
« *Logic and System Design for Low Power Consumption* »
Ph D Thesis 1996.

[Val01] M. Valluri & L. John
« *Is Compiling for Performance == Compiling for Low Power ?* »
In proceeding of INTERACT-5 2001.

[Valentin96] T. Valentin
« *Estimation de la consommation des circuits VLSI* »
Rapport de stage de DEA 1996.