Estimation et Optimisation de la Consommation lors de la conception globale de systèmes autonomes

Patricia Guitton

UNIVERSITE DE NICE-SOPHIA ANTIPOLIS - UFR Sciences
Ecole Doctorale de Sciences & Technologies de l'Information et de la Communication

T H E S E

pour obtenir le titre de
Docteur en Sciences
de l'UNIVERSITE de Nice-Sophia Antipolis

Discipline : (ou spécialité) Electronique

présentée et soutenue par
AUTEUR ***Patricia GUITTON***

Estimation et Optimisation de la Consommation lors de la conception globale des systèmes autonomes

Thèse dirigée par ***Michel AUGUIN***
soutenue le *jeudi 14 Octobre 2004*

Jury :

M. Jean-Marc Delosme	Professeur	rapporteur
Mme Nathalie Julien	Professeur	rapporteur
Mme Donatella Sciuto	Professeur	rapporteur
M. Michel Auguin	Directeur de recherche	examinateur
Mme Cécile Belleudy	Maître de Conférences	examinateur
M. Alain Anglade	invité (ADEME)	examinateur

Table des matières

Remerciements

Cette thèse s'est effectuée à l'Université de Nice Sophia-Antipolis et plus spécialement au sein du laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis (I3S) dans l'équipe MOSARTS (MOdélisation et Synthèse d'ARchitecture pour le Traitement du Signal) dont les études sont orientées vers les méthodes de co-conception logiciel/matériel d'applications de traitement du signal.

Je tiens particulièrement à remercier Mme Belleudy et M. Auguin qui ont co-encadré cette thèse, pour leur disponibilité, leurs conseils quotidiens, et leur bonne humeur quotidienne.

Tous mes remerciements à Mme Julien et Mme Sciuto et à M.Jean-Marc Delosme pour avoir rapporté sur mon mémoire de thèse. Je tiens à leur exprimer ma gratitude pour l'intérêt qu'ils ont porté à mes travaux.

Je remercie tous les membres de l'équipe pour les conseils et l'ambiance sympathique dans laquelle ce travail s'est déroulé.

Je remercie les directeurs successifs du Laboratoire I3S pour m'avoir accueilli, M. Bernhard en premier, puis, M. Fédou.

Une reconnaissance particulière pour Patrick Balestra, qui, en plus du café quotidien que nous prenions avec l'équipe, intervenait rapidement sur tous les problèmes administratifs.

Enfin des pensées pour leurs soutiens, aux cafés ou repas partagés qui vont droit vers Viviane, Micheline, Cécile, Corinne... et toutes les secrétaires.

Bien entendu, mes sentiments vont droit vers ma famille, et en premier lieu mon mari, qui a supporté mon travail pendant tant d'années, et m'a encouragé...

I. Introduction

I.1. Complexité accrue des applications pour des systèmes sur puce plus petits.

Les applications de télécommunication croissent en complexité de jour en jour. Il est demandé de plus en plus de calculs pour intégrer dans le même système un plus grand nombre de fonctionnalités (accès à internet, traitement multimédia...). Notons que les applications mobiles représentent aujourd'hui 32% du marché des ordinateurs individuels [NJ02].

Les nouvelles normes, telle que la «quatrième génération sans fil», exigent d'intégrer à la fois les services de téléphonie de la troisième génération, les fonctions sans fil entre périphériques d'un même système (comme par exemple Bluetooth ou Wi-Fi), des réseaux locaux «courtes distances», et les services associés à Internet! Toutes ces fonctionnalités vont être intégrées dans nos téléphones portables, sous forme d'un système sur puce (SoC, System on Chip). Cet objectif constitue un défi majeur compte tenu du faible accroissement de la productivité des outils et des méthodes de conception et du coût croissant des masques qui ne donnent pas ou peu le droit à l'erreur.

Le temps de réalisation de tels systèmes est lui-aussi en constante augmentation (il faut compter environ une année aujourd'hui).

Sur les circuits, il est classique de constater que :

- le nombre de transistors sur une puce double tous les 12 à 24 mois (loi de Moore),
- la fréquence des microprocesseurs double à chaque nouvelle génération (tous les 2 à 3 ans),
- la surface de silicium augmente de 25% par génération,
- la taille du logiciel double tous les ans.

Et bien que :

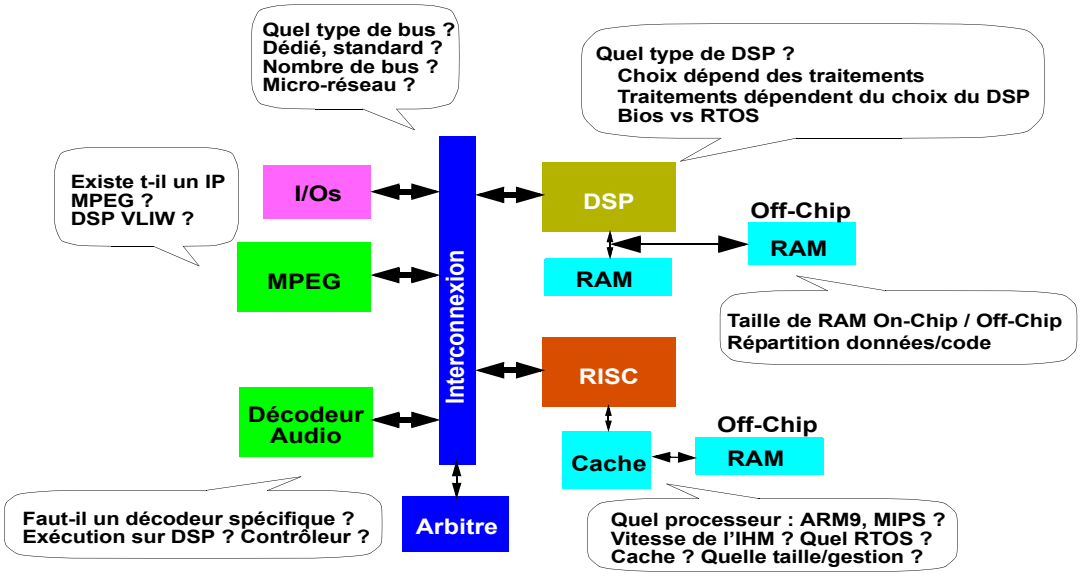
- la technologie (tailles des jonctions des transistors) diminue de 30% par génération,
- la tension d'alimentation diminue de 30% par génération, (jusqu'à 0.4V prévu pour 2016),

la consommation d'énergie et de puissance augmente.

La productivité des outils de conception systèmes croît seulement de 21% par an contre 58% pour la complexité des circuits. Pour remédier au fossé qui se creuse chaque année, il est important de développer des méthodes et des outils plus efficaces d'aide à la conception conjointe logiciel/matériel.

Dans le cas des systèmes embarqués actuels pour les télécommunications mobiles, on trouve classiquement un processeur de traitement du signal (*Digital Signal Processor*, DSP) et un processeur généraliste, tel qu'un RISC. Dans la majorité des cas, on ajoute à ces processeurs des «accélérateurs matériels», fonctions dédiées à des tâches particulières de l'application permettant de réaliser les traitements dans la durée d'exécution qu'ils doivent respecter. Par exemple, le signal de parole compressé et modulé dans la ligne de transmission de radio fréquence doit être restitué en un temps déterminé pour le confort des usagers de téléphones portables. Voici un exemple de système sur puce illustré sur la figure 1 .

FIGURE 1. Exemple de système sur puce



La détermination des composants présents dans ce système sur puce implique des techniques de conception conjointe logiciel/matériel. A partir d'un ensemble de modules disponibles (ASICs, FPGA, processeurs, mémoires, Network On Chip...), et à partir d'une description d'une application sous forme de tâches, la méthode de conception détermine les modules à affecter à l'exécution de chaque tâche de l'application.

Ainsi, non seulement, il est primordial de développer dans les premières étapes du flot de conception des méthodes de partitionnement logiciel/matériel pour compenser la croissance de la complexité des applications, mais il est aussi nécessaire d'intégrer la consommation comme paramètre lors de la conception globale de ces systèmes autonomes. En effet, la réduction de la consommation est importante, tant du point de vue écologique que de celui du poids des batteries qui est un critère majeur pour des systèmes portables. Particulièrement, réduire l'énergie permet d'augmenter la durée de

vie des batteries et donc l'autonomie du système, et la réduction de la puissance permet de réduire les écarts thermiques qui altèrent les circuits.

Nous avons situé la problématique, le rôle et la nécessité du développement de systèmes sur puce et des méthodes de conception associées prenant en compte les paramètres de durée d'exécution, de surface et de consommation. Le travail préalable au partitionnement consiste à caractériser en consommation tous les modules susceptibles de constituer l'architecture afin d'effectuer un partitionnement efficace. Nous nous intéressons à ces deux aspects dans la suite de ce mémoire en ciblant particulièrement les systèmes orientés traitement décrits par des graphes de flots de données.

Les motivations de cette thèse sont donc liées à la gestion de l'énergie, avec des retombées positives sur la pollution liées aux économies faites sur les batteries : durée de vie augmentée, meilleure utilisation de l'énergie disponible. Pour ces raisons, la thèse a été financée par l'Agence de l'Environnement et de la Maîtrise de l'Energie et le Conseil Régional de la région Provence Alpes Cote d'Azur.

I.2. Contenu de ce mémoire

Le mémoire comporte quatre parties:

Afin de minimiser la consommation lors de la conception globale de systèmes sur puce, il est nécessaire de disposer de modèles de consommation de chacun de ses composants. **Le chapitre deux** présente les techniques actuelles d'estimation de la consommation à différents niveaux, les diverses techniques de gestion de la puissance et de réduction utilisées dans les processeurs et dans les modules matériels. Nous constatons que peu de modèles sont disponibles dans le monde académique et dans le monde industriel. C'est pourquoi, nous nous sommes particulièrement intéressés à la consommation des processeurs et des unités dédiées à l'exécution de certaines parties de l'application.

Comme notre travail s'inscrit dans l'estimation et l'optimisation de la consommation lors de la conception globale de systèmes autonomes, nous nous intéressons aux méthodes de conception dans le **chapitre trois** qui présente donc les techniques de partitionnement les plus référencées dans la littérature. Les travaux dans le domaine sont étudiés en insistant sur ceux qui ont pour objectif de réduire la consommation. Les liens entre le partitionnement, l'ordonnancement et la basse consommation sont étudiés. L'outil de partitionnement développé au laboratoire, CODEF, est présenté dans ce chapitre.

Après avoir présenté des techniques classiques dans le domaine, le **chapitre quatre** présente nos modèles d'estimation de la consommation, aux différents niveaux requis, pour atteindre notre objectif. Ces modèles permettent de renseigner

les bibliothèques de l'outil de partitionnement développé au laboratoire, CODEF. Cela passe par les estimations des parties logicielles et des parties matérielles. Pour les premières, nous avons étudié particulièrement des processeurs de traitement de signal (DSP), car très présents dans les systèmes embarqués. Nous avons pu déduire des modèles de consommation unité fonctionnelle par unité fonctionnelle. Par ailleurs, une retombée de ces modèles est qu'il est possible de déduire des règles d'écriture de code optimisant la consommation. Cependant, les DSP, en raison de leur architecture hétérogène, ne disposent généralement pas de compilateurs efficaces. L'équipe a développé en collaboration avec Philips Semiconductors Sophia, un estimateur de performance que nous avons fait évoluer pour estimer en même temps la consommation. Cet estimateur et ses extensions sont décrits dans ce chapitre. Sur la base de la consommation individuelle de chaque entité, nous avons intégré dans CODEF un estimateur de consommation des systèmes obtenus par partitionnement.

Le dernier chapitre présente les optimisations en consommation effectuées pendant et après le partitionnement de CODEF. Dans ce chapitre, nous décrivons en détail la synthèse de niveau système. Nous analysons dans ce chapitre les étapes dans lesquelles il est intéressant d'intervenir pour optimiser la consommation.

Lors de la conception de ces solutions architecturales, une première optimisation consiste à favoriser le choix d'unités basse consommation lors de la phase d'allocation, tout en respectant les deux autres contraintes, à savoir le temps d'exécution et la surface totale du sillicium. Ceci est basé sur la notion d'urgence temporelle des différentes tâches. Ainsi, quand l'échéance d'une tâche n'est pas contraignante, on peut favoriser le choix d'une implémentation basse consommation plutôt qu'une implémentation «rapide».

Une deuxième optimisation consiste à raffiner l'étape d'ordonnancement en exploitant l'ajustement en tension et en fréquence. Cette technique permet de baisser à la fois la tension d'alimentation et la fréquence, ce qui ralentit l'exécution de la tâche et réduit sensiblement la consommation d'énergie.

Une troisième optimisation consiste à gérer les modes basse consommation, c'est-à-dire, mettre les unités dans les différents modes repos dès que possible et que cela est bénéfique.

Enfin, des résultats sur une application de détection de mouvement pour caméra embarquée, et sur l'encodeur et le décodeur de l'application MPEG4 sont présentés.

En fin du mémoire, nous présentons les améliorations et enrichissements à apporter à ce travail, ainsi que les perspectives ouvertes.

Chapitre II : Etat de l'art de la consommation

Introduction

L'objectif de ce chapitre est de présenter des travaux qui ont été réalisés dans les domaines de la réduction de la consommation en énergie et en puissance, en partant du niveau technologique, jusqu'au niveau système. Classiquement, les travaux s'intéressent au :

- niveau technologique
- niveau logique
- niveau processeur
- niveau système.

Afin de prendre en compte la consommation lors de la conception globale de systèmes autonomes, il est nécessaire d'étudier les différents modèles de consommation développés et d'analyser leur adéquation par rapport à un objectif d'optimisation au niveau architecture et système. De cette analyse, nous pourrons en déduire ceux effectivement utilisables et ceux qu'il s'agira d'étendre ou de développer. Ici, nous présentons donc des méthodes d'estimation et d'optimisation de la consommation issues de la littérature.

La consommation est devenue une métrique incontournable dans l'évaluation des systèmes électroniques numériques. Aussi, regardons dans ce chapitre la consommation aux différents niveaux cités précédemment en définissant les sources de consommation d'énergie et de puissance, et la façon de les estimer et de les maîtriser.

1 •Estimer et réduire la Consommation au niveau technologique

1.1.Estimer la consommation au niveau technologique.

La consommation d'un circuit intégré au niveau transistor peut se décomposer en deux termes : la «consommation statique» due principalement à des courants parasites, et la «consommation dynamique» conséquence de l'activité de commutation (ou «*switching activity*») des circuits.

Voyons d'où viennent ces facteurs.

1.1.a.Consommation statique.

Lorsqu'un circuit CMOS ne commute pas, sa consommation est faible (due à la résistance interne entre l'alimentation et la masse). Mais il reste quand même des courants parasites dont le plus important est le courant de fuite résultant du courant «sous-seuil» et du «courant-inverse».

- ***Le courant «sous-seuil»*** est la somme des courants engendrés par la diffusion de porteurs entre le drain et la source du transistor en régime de faible inversion c'est-à-dire le courant de fuite lorsque la tension entre la grille et la source est inférieure à la tension seuil. Ce courant est la plupart du temps négligé [BE95].
- ***Le courant inverse $I_{inverse}$*** est la somme des courants de fuite dans les diodes parasites formées entre le drain, la source et le substrat. C'est-à-dire, le courant minoritaire inverse qui se crée entre les jonctions. Son ordre de grandeur est le femto-ampère (fA : 10^{-15}). Ils sont négligeables tant que V_{dd} est supérieure à V_{seuil}, avec V_{dd}, tension d'alimentation, et V_{seuil}, la tension seuil caractéristique de la technologie.
- ***Conclusions :*** La puissance statique consommée est le produit de la tension d'alimentation par le courant de fuite:

$$P_{statique} = V_{dd} \times I_{fuite} = V_{dd} \times I_{inverse} \qquad (1)$$

Dans cette formule, I_{fuite} est la somme des courants parasites en régime statique.

Ces facteurs de consommation sont déterminés dès la conception par la technologie du composant. L'optimisation s'effectue au plus bas niveau d'abstraction où les paramètres technologiques de fabrication sont connus.

Pour autant, peut-on négliger ces contributions ?

Dans les faits, ces courants sont souvent négligeables pour les blocs logiques ayant beaucoup d'activité. La puissance statique pour les processeurs commercialisés actuellement sera négligée[NJ02]. Par contre, pour les mémoires de grandes

tailles ayant peu d'activité (relativement au nombre de transistors), ces contributions ne sont plus négligeables.

Par ailleurs, comme nous le verrons dans la suite, les technologies développées et prévues à court ou moyen terme, poussent à réduire la tension de seuil. Or, cette diminution augmente le courant de fuite.

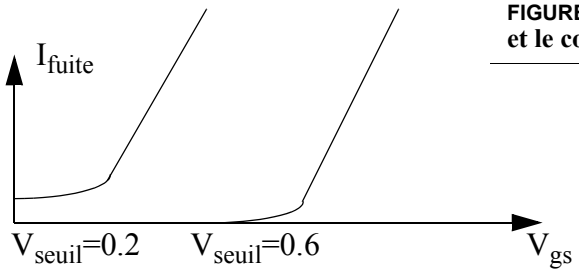


FIGURE 2. Lien entre la tension seuil et le courant de fuite.

Pour privilégier la consommation, il faudra donc préférer une tension de seuil élevée comme l'illustre la figure 2, Vgs étant ici la tension grille-source du transistor. Cependant, avec une tension V_{dd} faible, les portes logiques deviennent plus sensibles aux bruits. Etant donné que nous travaillons avec les technologies actuelles CMOS, nous négligerons la puissance statique dans toute notre étude. On peut difficilement chercher à optimiser ce type de contribution à la consommation avec des outils de conception système, car il s'agit avant tout d'un problème lié directement à la technologie utilisée. Nous revenons sur ce point un peu plus loin.

1.1.b.Consommation dynamique.

En régime dynamique deux facteurs sont prépondérants : d'une part le courant de court-circuit, d'autre part le courant de commutation. Ces contributions apparaissent lors de la commutation des transistors.

- ***Courant de court-circuit*** : dans le cas d'un inverseur CMOS (figure 2) une transition montante à son entrée entraîne la commutation simultanée des deux transistors (les transistors PMOS et le NMOS sont passants) sur un court intervalle de temps, ce qui crée un chemin direct entre l'alimentation et la masse.

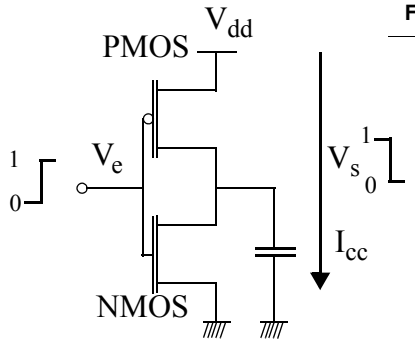


FIGURE 3. Courant de court-circuit

V_{dd}: tension d'alimentation
V_e : tension d'entrée
V_s : tension de sortie
I_{cc} : courant de court-circuit

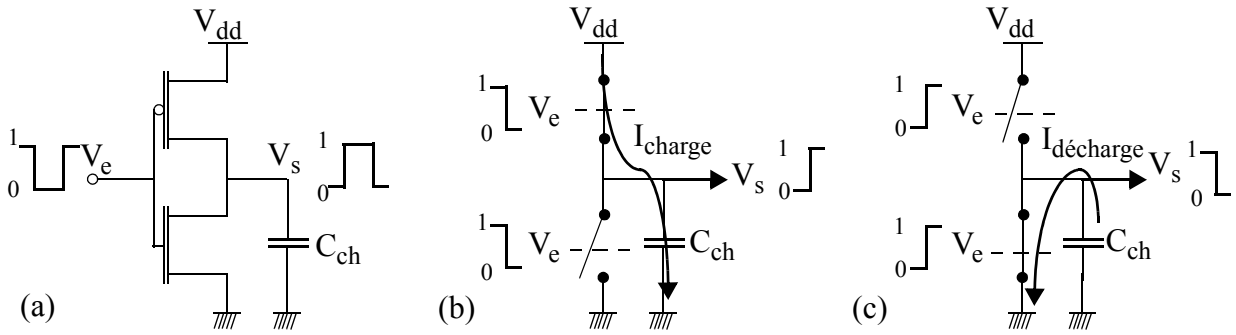
D'où l'expression : $P_{court\text{-}circuit} = V_{dd} \times I_{cc_{moy}}$ (2)

Ce courant est proportionnel au temps de montée, Tm (durée pour une transition 0-1 complète), et aux dimensions du transistor. Ce courant est de l'ordre du micro-ampère (µA : 10^{-6}), soit une grandeur très supérieure au courant inverse statique.

Pour des circuits conçus correctement, le courant de court-circuit représente 15% de la consommation dynamique [CG03].

- ***Courant de commutation*** : pour mettre en évidence la consommation due au courant de commutation on reprend l'exemple de l'inverseur avec une charge en sortie modélisée par une capacité C_{ch} (figure 4(a)).

FIGURE 4. Courant de commutation



On représente le comportement de l'inverseur CMOS avec en entrée une transition descendante (figure 4(b)) ou une transition montante (figure 4(c)). On modélise les portes NMOS et PMOS comme étant idéales par des interrupteurs.

Analysons la consommation en puissance de l'inverseur CMOS en commutation:

Transition descendante : le transistor PMOS est passant; le transistor NMOS est bloqué. Il y a donc charge de C_{ch}.

Un courant de charge (I_{charge}), fourni par l'alimentation, charge C_{ch}. Le calcul, sur un intervalle de temps [0,Tm] (nécessaire pour une transition de 1 à 0), de l'énergie $E_{1\text{-}0}$ fournie par l'alimentation, et de l'énergie $E_{stockée}$ accumulée par C_{ch} est :

$$P_{inst}(t) = V_{dd} \times I_{alim}(t) = V_{dd} \times C_{ch} \times \frac{d}{dt}(V_s)$$

$$\text{d'où : } E_{1\text{-}0}(T_m) = \int_0^{T_m} (P_{inst}(t) \cdot dt) = V_{dd} \times \int_0^{V_{dd}} (C_{ch} \cdot dV_s) = C_{ch} \times V_{dd}^2$$

et : $E_{stockée} = \int_0^{T_m} \left(P_{C_{chg}}(t) \cdot dt \right) = \int_0^{V_{alim}} (C_{ch} \cdot V_s \cdot dV_s) = \frac{1}{2} \times C_{ch} \times V_{dd}^2$

avec :

- I_{alim} : courant instantané fourni par l'alimentation;
- C_{ch} : capacité de charge;
- P_{inst} : puissance instantanée aux bornes du circuit;
- Pc_{chg} : puissance instantanée aux bornes de C_{ch};
- V_{dd} : tension d'alimentation;
- T_m : temps de montée des transistors;
- V_s : tension en sortie.

On voit que l'énergie stockée par C_{ch} ne représente que la moitié de l'énergie fournie par l'alimentation. La seconde moitié est dissipée par effet joule dans le réseau de transistors PMOS.

De façon analogue, analysons la transition montante :

Transition montante : le transistor PMOS est bloqué; le transistor NMOS est passant, il y a donc décharge de C_{ch}.

Le courant de décharge $I_{décharge}$ fourni par la charge C_{ch} parcourt le réseau NMOS où se dissipe l'énergie : $E_{stockée} = \frac{1}{2} \times C_{ch} \times V_{dd}^2$.

Mais l'intervalle [0 ; Tm] représente le temps minimal nécessaire pour effectuer pleinement une transition en sortie de l'inverseur. Pour décrire ce comportement au niveau des processeurs il faut considérer que pour un cycle d'horloge, plusieurs milliers ou millions de transitions interviennent, d'où le modèle :

$$P_{commutation} = \alpha_{0\text{-}1} \times C_{ch} \times V_{dd}^2 \times f_{clock}$$

avec :

- $\alpha_{0\text{-}1}$: nombre moyen de commutations par cycle d'horloge pour une cellule logique du type inverseur CMOS;
- f_{clock} : fréquence de l'horloge.

D'où l'expression globale : $P_{commutation} = \sum_{i}^{n}\left(\alpha_{0\text{-}1_i} \cdot C_{ch_i} \cdot V_{dd}^{2} \cdot f_{clock}\right)$

avec :

- n : nombre de transistors;
- $\alpha_{0\text{-}1_i}$: nombre moyen de transitions de la cellule i;
- C_{ch_i} : capacité de charge de la cellule i.

Par conséquent, on déduit de l'analyse précédente l'expression de la puissance dynamique :

$$P_{dynamique} = P_{court\text{-}circuit} + P_{commutation}$$

$$P_{dynamique} = V_{dd} \times I_{cc_{moy}} + \sum_{i}^{n}\left(\alpha_{0\text{-}1_i} \cdot C_{ch_i} \cdot V_{dd}^{2} \cdot f_{clock}\right)$$

On se rend compte lors de cette étude que les paramètres qui génèrent ces pertes en puissance en régime dynamique sont liés aux caractéristiques matérielles du composant comme : la taille des transistors, la tension d'alimentation, la fréquence d'horloge et le nombre de transitions.

1.1.c.Conclusion.

L'étude précédente sur la consommation en puissance liée à l'aspect matériel des systèmes électroniques en technologie CMOS nous mène à l'expression : $P_{totale} = P_{dynamique} + P_{statique}$.

Les études bibliographiques issues de [CJ99] montrent que la consommation due à la puissance statique ne représente en moyenne que quelques nano-ampères (nA : 10^{-9}) de la consommation globale d'un circuit. De plus, il est mentionné que la puissance de court circuit représente 20% à 30% de la consommation dynamique et que la puissance de commutation représente 70% à 80% de la consommation globale d'un circuit dans de bonnes conditions.

Ces remarques justifient notre choix de négliger la puissance statique dans toute la suite de nos travaux, car les technologies sur lesquelles nous travaillons actuellement sont les mêmes. La puissance court-circuit représente 10% de la puissance globale pour une technologie adaptée, aussi elle est en général négligée.

Finalement, on approxime la puissance dynamique par :

$$P_{totale} \cong P_{dynamique} \cong P_{commutation} = \alpha \times f_{clock} \times C_{ch} \times V_{dd}^{2} \qquad (10)$$

Les techniques d'optimisation de ces facteurs de consommation comme la réduction des temps de montée des portes, la réduction de la tension d'alimentation, l'adaptation de la taille des condensateurs... sont utilisées dans les bas niveaux de conception comme nous le voyons dans la suite.

1.2.Réduire la consommation au niveau technologique.

Diverses optimisations sont envisagées. La puissance statique représente moins de 15% de la consommation de la puissance dynamique d'après [MP96], on la néglige donc.

Le problème de la maîtrise de la consommation est très complexe. De nombreux paramètres sont interdépendants. En effet, la puissance étant proportionnelle au carré de la tension, on peut penser que le plus efficace est de la baisser. Or, le temps de propagation du circuit est proportionnel à : $\frac{C_{ch} \times V_{dd}}{(V_{dd} - V_{seuil})^2}$ (11).

Il est donc nécessaire de trouver un compromis entre la vitesse et la consommation (opération de *voltage scaling)*, guidé par le critère énergie.

Par ailleurs, réduire V_{dd} conduit à baisser V_{seuil}. On constate une légère augmentation du courant de fuite et donc la puissance statique. Les technologies futures offrent bien sûr des avancées mais le courant de fuite est augmenté, il devient alors nécessaire de prendre en compte la puissance statique pour ces technologies. Cependant, les techniques d'estimation et d'optimisation au niveau système n'en seront pas bouleversées.

On peut aussi essayer de diminuer les capacités, c'est-à dire globalement la taille du circuit.

Une dernière possibilité est de tenter de réduire l'activité de commutation du circuit par la réduction de commutations parasites (*glitchs*) et par la mise en veille des portes du circuit qui ont une activité inutile.

Maintenant que nous connaissons les sources de consommation au plus bas niveau et comment tenter de les réduire, voyons leurs répercussions aux autres niveaux.

2 •Estimer et réduire la Consommation au niveau logique

2.1.Estimer la consommation au niveau logique.

Dans un circuit constitué de portes logiques, la puissance dissipée principale est donc la puissance dynamique. Elle dépend principalement de l'activité et donc des données. Il est en général assez difficile d'estimer pour un système complexe la puissance dynamique, car un très (trop) grand nombre de vecteurs de test est nécessaire pour balayer les diverses possibilités de valeurs des données.

Dans l'expression de la puissance dynamique, réduite à celle de la commutation (formule 10), le terme dépendant des données est principalement $\alpha \times C_{ch}$. Pour l'évaluer, les outils comme SPICE, POWERMIL... effectuent à partir de vecteurs de test une simulation de l'architecture. La trace obtenue permet à ces outils de déduire la consommation. Ce genre de techniques nécessite beaucoup de temps et de mémoire. On peut citer des améliorations apportées par des techniques consistant à éliminer les informations redondantes de la trace [WW99], ou bien encore à réduire l'information par le compactage par blocs des activités de commutation [BD97].

Signalons cependant d'autres approches utilisant des méthodes statistiques et probabilistes qui permettent d'estimer différents facteurs entrant dans la consommation.

2.1.a.Méthodes probabilistes [CM87].

Ce type de méthode se définit comme étant une mesure directe dans le circuit des probabilités d'états et/ou de transition.

Logique combinatoire : on calcule la densité de probabilité des transitions et des capacités internes du circuit à partir de simulations [FP98]. Citons les méthodes suivantes :

- Utilisation du *zero-delay model*[RD96] *:* le facteur d'activité à chaque noeud du circuit est considéré comme corrélé très fortement à la probabilité de chaque noeud d'être à un. Le plus difficile est d'estimer ces probabilités et il existe de nombreuses techniques et de théories qu'on peut mettre en oeuvre. Elles restent compliquées et longues à réaliser.

- Utilisation du *real-delay model* : il s'agit de prendre en compte les *glitchs (retards des signaux dans le circuit)*. [PS98]. Leur consommation est estimée à environ 20% de la consommation totale et peut parfois atteindre jusqu'à 70% [NF95].

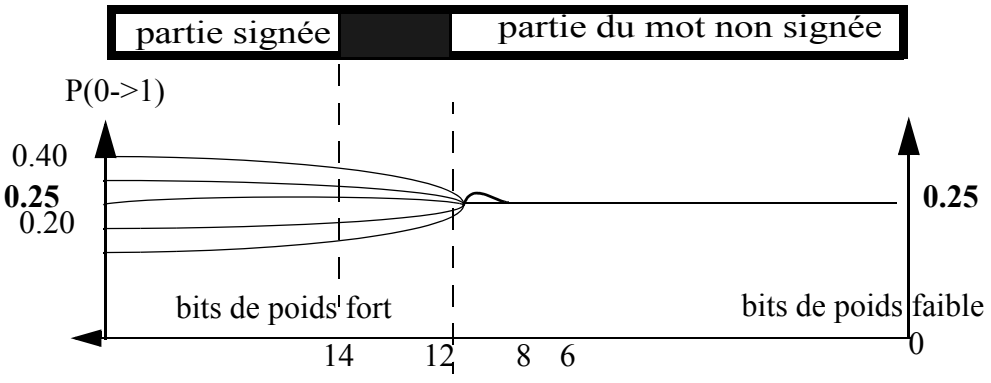
Logique séquentielle : on calcule les probabilités des états internes du circuit [BM99]. Voyons quelques démarches.

- Utilisation de la théorie des chaînes de Markov [CR95]. Les états suivants sont supposés ne pas dépendre des états précédents, seulement des états présents. A partir, soit des probabilités de transition, soit des probabilités d'état, on obtient respectivement les probabilités d'état et les probabilités de transition. On en déduit l'activité de commutation, ce qui permet d'estimer la consommation.

- Exploitation du modèle *Dual Bit Type* dont le but est de rendre compte de la consommation en fonction des données, donc en fonction de la valeur des bits. Les paramètres nécessaires comme la moyenne, la variance et la corrélation sont obtenus par simulation classique ou par mesure. Il faut bien entendu effectuer des simulations sur des échantillons conséquents. Ainsi, les auteurs de [LR94, LR95] constatent que tous les signaux d'entrées, qu'ils soient tirés d'une image ou d'une musique présentent les mêmes caractéristiques statistiques. Ils présentent une fonction de corrélation temporelle de forme Gaussienne. Celle-ci est supposée égale à la densité de probabilité de transition d'un bit de 0 à 1. Ils ont donc constaté que les statistiques des mots et des bits étaient très fortement liées.

Le schéma (figure 5) ci-dessous présente les caractéristiques de la courbe Gaussienne qui détermine les zones de comportement statistique au niveau du mot :

FIGURE 5. Courbe Gaussienne décrivant statistiquement les zones de comportement d'un mot



- Comme l'activité de commutation dépend des données, certains chercheurs se sont attachés à déterminer la probabilité de transition de chaque noeud. Ainsi, dans [XN94], il est proposé une méthode nommée «*Mean Estimator of Density*». A partir de la densité D(x) de transitions, pour chaque noeud, et de la fraction de temps où le circuit est au niveau haut, on obtient la probabilité P(x)

de transition. Par défaut la probabilité est 1/2. Par simulations répétées N fois, on obtient le nombre de transitions. Les noeuds sont alors classés en noeuds à densité régulière et à densité faible. Les auteurs différencient les montages synchrones (on prend alors en compte la période d'horloge) des montages asynchrones. On obtient donc des probabilités de transition de bit de 0 à 1 avec la formule : P(1/0)=TcD(x)/2P(x) qui permet d'estimer la consommation de ces changements de valeurs. Dans cette formule, Tc est le temps où le circuit est au niveau haut.

- Nous signalons encore les travaux relatés dans [PC95] qui déterminent au préalable des valeurs moyennes de capacité physique et de l'activité de commutation pour chaque bit d'entrée.

2.1.b. Méthodes statistiques.

Prenons par exemple les travaux de [BN92]. A partir d'une série d'entrées (vaste et représentative) présentées au simulateur logique, un ensemble de résultats caractéristiques (capacités, nombre de commutations...) est obtenu en sortie permettant de calculer la moyenne de la puissance dissipée.

Empruntant cette démarche, les travaux de [GN97] développent une technique basée sur des librairies de modèles (*profile-driven*).

On peut distinguer différentes approches, par exemple:

- celle utilisant la technique de Monte Carlo pour générer les vecteurs d'entrée au hasard qu'on injecte dans le simulateur et l'estimateur de puissance. Les puissances consommées relevées sont alors considérées comme des variables aléatoires. Le problème est alors de savoir combien d'échantillons il est nécessaire de prendre en compte pour une erreur donnée tolérée. Dans [BN92], ils proposent d'utiliser au minimum N échantillons tel que: $N > \langle \frac{(\alpha\sigma)}{\eta\varepsilon} \rangle^2$, où $\alpha, \sigma, \varepsilon, \eta$ sont respectivement «représentant» de la proportionnalité, de la déviation standard, de l'erreur tolérée et de la moyenne.

- ou encore, l'utilisation de paramètres dus à la technologie et aux tailles des transistors comme dans les travaux de [JP97]. Ils construisent un diagramme de transitions des états à chaque noeud du circuit, et y font figurer en annotation les signaux d'entrée et de sortie et les probabilités de chaque transition. A l'aide de cette approche, ils ont constitué une librairie des circuits les plus typiques et les plus répandus. L'énergie de chaque transition est calculée en utilisant HSPICE. Ils associent un facteur moyen d'activité à chaque circuit étudié. En le multipliant par la taille du mot, ils obtiennent une énergie pour chaque partie de circuit.

Limites :

Ces méthodes sont très limitées puisqu'il est nécessaire de disposer d'une description précise du circuit (niveau logique) sur laquelle est effectuée ensuite une simulation très coûteuse en temps. Elles deviennent de plus en plus compliquées à réaliser de par la complexité architecturale croissante des SOC, et prennent donc encore plus de temps.

Pour ces raisons, nous ne considérons pas ces approches, mal adaptées à nos objectifs qui sont d'intégrer des modèles de consommation dans les méthodes d'aide à la conception conjointe logicielle/matérielle.

2.2.Réduire la consommation au niveau logique.

Diverses techniques d'optimisation de la consommation ont été proposées au niveau logique, nous décrivons les principales.

2.2.a.Mise à zéro de la tension d'alimentation.

La mise à zéro de la tension d'alimentation ou le «*power supply shut down*» consiste à couper l'alimentation de certaines parties du circuit, momentanément non utilisées.

Pour réaliser cette fonctionnalité au niveau logique, il faut par exemple ajouter un interrupteur (switch) pour les parties de circuits visées. Malheureusement, chacun de ces interrupteurs ajoute alors une résistance et un délai. Aussi, on trouve plus couramment l'utilisation de transistors MOS. Afin de diminuer la résistance ajoutée par ce dernier, il faut augmenter sa largeur, ce qui accroît sa capacité. En plus de cette augmentation de capacité, il est nécessaire d'introduire un circuit buffer qui contrôle l'entrée de ce transistor pour que la stabilité soit satisfaisante. En effet, la «mise en mémoire» des informations permet d'avoir une plus grande stabilité. Bien évidemment, cette solution implique un délai de stabilisation qui limite l'intérêt de son application.

D'autres problèmes peuvent apparaître si le circuit contient des mémoires volatiles. En effet, avant la coupure de l'alimentation de cette partie, il faut sauver les données mémorisées. Il faut ensuite les restaurer au moment de la remise en route, ce qui augmente le coût en consommation.

Par ailleurs, l'emplacement physique de ce type de dispositif peut entraîner des capacités de couplage parasites. Il faut donc soigner ce choix.

En raison de tous ces problèmes, cette technique n'est pas utilisée actuellement dans la pratique.

2.2.b.Contrôle logique de l'horloge.

Cette technique, appelée en anglais *clock gating* est basée sur le découpage du circuit en zones contrôlées par des domaines d'horloges différents. Le signal d'horloge peut être bloqué par un signal logique de contrôle calculé pendant le fonctionnement normal. Ainsi, on peut interrompre l'horloge, et ceci pour chacune des régions définies indépendamment.

Contrairement à la méthode précédente, la puissance dissipée au repos est différente de 0 car le courant de fuite est indépendant de l'horloge. Plus particulièrement, le *clock gating* est adapté aux technologies CMOS statiques. En effet, en technologie CMOS dynamique, il faut rafraîchir périodiquement pour ne pas perdre toute l'information, donc maintenir une horloge active.

Cependant, en plus d'un surcoût énergétique, l'implémentation du *clock gating* a une sérieuse limite. En effet, si le signal de contrôle de l'horloge a un signal parasite (*glitch*), quand l'horloge est au niveau haut, le *glitch* sera propagé comme un front d'horloge, ce qui peut être désastreux. Il faut à tout prix éviter cela. On peut utiliser des bascules D «*latch*», qui ne réagissent qu'au niveau de l'horloge et donc seront insensibles à ces signaux parasites de front montant de signal de contrôle de l'horloge. Beaucoup d'études dans ce domaine sont menées pour remédier à ces problèmes, et à ce niveau, cette technique est très utilisée.

2.2.c.Bascules flip-flops de filtrage.

Quand le *clock gating* n'est pas disponible, une autre technique consiste à utiliser des bascules D *flip flop* en entrées, agissant en quelque sorte comme filtre. Un multiplexeur est relié à l'entrée de cette bascule. Lorsque le signal de contrôle du multiplexeur est à 1, il transmet une donnée, lorsqu'il est à 0, on ne change pas la donnée précédente, la donnée de la bascule D reste constante, limitant ainsi les activités de commutation.

L'avantage est le peu de circuits à ajouter, et une analyse aisée par les outils de niveau logique.

Malheureusement, dans le cas de préchargement à 1, son effet est dramatique, car la transmission de 0 consomme beaucoup, et la vitesse du circuit est réduite du fait que la transition descendante est plus lente.

En résumé, trois techniques de gestion de la puissance ont été présentées au niveau logique. La plus efficace et la plus répandue pour un circuit CMOS à faible courant de fuite est, comme nous l'avons vu, le *clock gating*.

Voyons maintenant ce que nous réserve l'avenir.

2.2.d.Ce que nous réserve l'avenir...

Les nouvelles technologies sont élaborées afin de réduire globalement la consommation, mais les méthodes d'estimation devront alors tenir compte de la puis-

sance statique qui n'est plus négligeable dans ce cas. En particulier, avec la diminution de la tension de seuil, le courant de fuite augmente. Or, les nanotechnologies entraînent une diminution de cette tension. La puissance statique arrive au tiers de la puissance globale pour une technologie de 0.09 μm et la moitié pour une technologie de 0.06 μm [NJ02]. Actuellement, une opération exécutée par un circuit fait monter la température jusqu'à 100°C au niveau de la jonction quand la température ambiante est de 45°C! Ces températures affectent l'intégrité des circuits et sont donc à maîtriser. Actuellement, on rafraîchit le système, ce qui entraîne un coût de 1$ par watt rafraîchi!

Les dernières évolutions des transistors réduisant les courants de passage ont pour conséquence d'augmenter les courants de fuite. Or, la technique du *clock gating* ne permet pas de réduire les contributions de ces courants. Or c'est la technique la plus répandue au niveau logique. Quelles sont donc les solutions futures à ce problème ?

Les auteurs de [SK01] présentent une solution intéressante permettant de réduire ce courant de fuite dans les dispositifs de plus en plus petits, où les transistors ont des dimensions de l'ordre de 50 ou 35 nanomètres. Leur idée est finalement simple (la réalisation l'est bien moins!) : il s'agit d'avoir deux tensions d'alimentation possibles pour une puce, donc deux régimes dont un «basse consommation». Le découpage de l'alimentation se fait au niveau logique. Les portes mises en régime basse consommation sont celles qui ne sont pas critiques en temps.

De même, ils arrivent à réduire la tension seuil de ces dispositifs. Les portes qui sont critiques en temps sont faites avec une tension seuil basse (pour aller vite), et les autres avec une tension seuil haute. Les résultats typiques obtenus grâce à ces implémentations montrent des réductions de 40 à 80% de la puissance de fuite avec des altérations en temps.

Voyons maintenant la consommation au niveau processeur.

3•Estimer et réduire la Consommation au niveau processeur

3.1.Estimer la Consommation au niveau processeur.

Les outils actuels de simulation comme PSPICE, POWERMIL permettent de calculer la consommation d'un processeur à partir de son modèle VHDL et des caractéristiques techniques de réalisation (*netlist*). Cependant, cette approche est peu applicable sur de réels codes applicatifs car la mise au point de vecteurs de test et la simulation de l'architecture au niveau logique pour quelques instructions assembleur nécessitent un temps de l'ordre d'une semaine.

Il apparaît donc indispensable d'utiliser des modèles plus abstrait.

Des études ont montré [TM96] qu'à partir de la consommation par type d'opérations on pouvait calculer la consommation du processeur et donc prévoir la consom-

mation d'un programme calculée par l'expression suivante:

$$E = \sum_{i}(R_i \times N_i) + \sum_{i,j}(O_{(i,j)} \times N_{(i,j)}) + \sum_{ki}(E_k)$$

où :

- $O_{(i,j)}$: coût dû aux changements de chemin de données et de contrôle entre deux instructions consécutives (overhead);
- R_i : coût de base d'une instruction en consommation;
- N_i : nombre de cycles nécessaires à l'exécution du programme.
- E_k, représente la consommation due aux cycles perdus, par exemple, lorsque les données ne sont pas dans le cache (*cache stall*) ou que le pipeline (*pipeline stall*) doit être vidé.
- $N_{(i,j),}$ le nombre de fois où les instructions i et j sont exécutées consécutivement;

Ces modélisations de la consommation de processeurs ont fait l'objet de divers travaux par plusieurs équipes de recherche. Regardons les modèles et outils qu'elles ont élaborées pour pouvoir rapidement estimer la consommation d'un processeur exécutant une application.

3.1.a.Outil d'estimation pour trois processeurs [Jo].

L'outil Jouletrack, disponible sur le web, développé dans le cadre du projet «µ *Adaptative Multi-domain Power aware Sensors*» par les *Microsystems Technology Laboratories*, permet d'estimer la consommation d'un code C exécuté par le Strong ARM. Cet outil est actuellement étendu pour les processeurs TMS 320C5x et Hitachi-SH4. Jouletrack fournit le code compilé en assembleur et une estimation en Joule de la consommation due à son exécution.

Par exemple, pour une transformée en cosinus discret, DCT, simulée pour le Strong ARM à 206 MHz, **Jouletrack** fournit le rapport suivant :

Tableau 1: rapport de Jouletrack pour l'exécution de la DCT

Operating frequency	206MHz
Operating voltage	1.5 Volts
Simulation level	0
Execution time	44347 secs
Average current	0.2374 Amps
Total energy	15792.0 Joules

3.1.b.Un modèle de la consommation du DSP TMS 320C6201 [NJ02].

Le Laboratoire LESTER, de l'Université de Bretagne Sud a défini un modèle de consommation du DSP TMS 320C6201/RISC par analyse fonctionnelle. C'est un DSP équipé d'un profond pipeline (11 étages), d'instructions VLIW, et dont le parallélisme peut atteindre 8 instructions par cycle.

Les mesures réelles au niveau assembleur ont pour but de déduire un modèle de consommation fonctionnel paramètré. Ces mesures ont pour but d'exciter chaque unité fonctionnelle du processeur. Les valeurs des données ne sont pas prises en compte dans ce modèle.

Ainsi, dans un premier temps, les courants dépendent de facteurs appartenant à deux catégories principales :

- paramètres de configuration : la fréquence en MHZ
- paramètres algorithmiques :
  - le taux de parallélisme, α
  - le taux d'unités de traitement actives, β
  - le taux de défaut de cache, γ
  - le taux de rupture de pipeline, PSR

Pour déterminer α et β [NJ02], on considère que le DSP charge 8 instructions en même temps. Elles forment un paquet de chargement (FP). Toutes les instructions exécutées en parallèles forment un EP. Chaque FP peut être constitué de plusieurs paquets d'exécution (EP). Les deux paramètres α et β sont alors calculés de la manière suivante :

$$\alpha = \frac{NFP}{NEP} \leq 1$$

$$\beta = \frac{1}{8} \cdot \frac{NPU}{NEP} \leq 1$$

Avec NFP et NEP, les valeurs moyennes de FP et EP respectivement. NPU représente le nombre moyen d'unités fonctionnelles utilisées (toute opération sauf l'instruction NOP).

Estimer la consommation, consiste à déterminer les valeurs des paramètres algorithmiques ci-dessus.

PSR et γ sont déterminés grâce à une trace du programme.

α et β sont déterminés directement à partir du code source C, en prédisant le nombre de paquets de chargement, d'exécution et le nombre d'unités de traitement utilisés par l'application.

Les auteurs appliquent ensuite ce modèle sur un programme écrit en C. Il est nécessaire que l'algorithme soit constitué de boucles de tailles suffisantes pour pouvoir négliger le code non inclus dans la ou les boucles. Ceci est vérifié dans la plupart des algorithmes de traitement du signal et de l'image.

Les résultats obtenus donnent des estimations proches des mesures réelles. Toutefois, cette approche ne semble pas être adaptée pour des codes irréguliers.

3.1.c.Un estimateur pour un processeur pipeliné.

Cet estimateur [KK02] est figé pour un processeur pipeliné à 5 étages, avec un cache interne à un seul niveau. Il s'agit d'une extension de *SimplePower* [IK01].

Le modèle énergétique est décomposé en contributions dues :

- aux chemins qui déterminent en fait le nombre d'activations de chaque type de composants,
- au cache, c'est-à-dire prenant en compte le nombre de défauts de cache, et les nombres d'écriture et de lecture dans le cache,
- à la mémoire principale, c'est-à-dire le nombre d'accès et les intervalles de temps entre chacun de ces accès,
- aux bus, c'est-à-dire le nombre de transactions,
- à l'horloge, c'est-à-dire le nombre de cycles d'exécution.

Le coût en énergie de chaque instruction est estimé un grand nombre de fois (environ 1000) avec des valeurs de données très différentes, afin d'obtenir des valeurs moyennes.

Les auteurs établissent une bibliothèque de valeurs de chaque contribution moyenne obtenue par ces mesures, comme illustré par le tableau 2.

Tableau 2: paramètres extraits des mesures, d'après [KK02].

Paramètre	Valeur
Tension d'alimentation	3.3 V
Configuration du Cache de données	4 KB, 2-way, 32 bytes
Temps d'accès à la ligne du cache de données (*Hit Latency*)	1 cycle
Accès à la mémoire (*Memory Access Latency*)	100 cycles
Energie par accès lecture au Cache de données	0.20 nJ
Energie par accès écriture au Cache de données	0.21 nJ
Energie par accès mémoire	4.95 nJ
Energie due aux Transactions des bus interne	0.04 nJ
Energie due aux Transactions des bus externes	3.48 nJ
Energie par Cycle d'horloge	0.18 nJ
Technologie	0.35 micron

Ces contributions permettent de déduire les coûts des instructions comme le montre le tableau 3:

Tableau 3: Exemples de coûts par instruction

Type d'instruction	Coût (énergie en nJ)
bne	200.04
mult	366.01

D'après la liste des contributions de consommation établie ci-dessus, l'estimation du coût d'une application nécessite la connaissance du nombre d'instructions exécutées par type. Ces informations sont obtenues après analyse du code compilé.

Pour estimer le nombre global de cycles d'exécution, il est considéré que chaque instruction nécessite un cycle d'initialisation (sans tenir compte des cycles d'attente).

Il faut ensuite ajouter le nombre estimé de cycles d'attente (*stall cycles*) (qui est pris fixe, estimé pour chaque défaut de cache déduit). Enfin, le nombre de transitions sur les bus est considéré comme proportionnel au nombre d'accès au cache et à la mémoire. On peut donc avoir une estimation du coût d'une application exécutée par ce processeur.

3.1.d.Modèle d'un processeur VLIW.

Ces processeurs ont intéressé nombre de chercheurs, dont [MD00]. Dans leurs travaux, un modèle de consommation de ce type de cœur de processeur est déduit après avoir réalisé des simulations et des synthèses. Ceci permet de détailler la contribution de chaque étage du pipeline pour tenir compte :

-du coût en consommation de l'exécution de l'instruction en cours dans l'étage,

-du coût dû aux connexions entre les étages du pipeline et de l'énergie consommée par les caches d'instruction ou de données.

Les modèles que nous avons présenté ont été, pour certains, intégrés dans les bibliothèques de notre outil de conception conjointe logicielle/matérielle, comme le modèle du processeur StrongARM. Les autres étaient trop dédiés pour être adaptés à nos objectifs.

Sans être exhaustif, nous avons répertorié ici quelques modèles de consommation de processeurs. Voyons maintenant quelques stratégies pour réduire leur consommation.

3.2.Réduire la consommation au niveau processeur.

3.2.a.Conception d'arbre d'horloge.

Le problème est complexe [BD98]. Il concerne la conception d'un arbre d'horloge pour les unités fonctionnelles. Comment choisir les domaines d'horloge de manière optimisée ?

L'opportunité de gagner en puissance est importante. En effet, d'un côté la multiplication des branches dans l'arbre d'horloges complique la gestion, mais de l'autre elle est l'occasion de minimiser les activités de commutation dans le circuit.

Pour cette réduction, une solution proposée est de regrouper les branches qui ont le même type d'activité. L'activité du nouvel arbre d'horloge en sera réduite.

3.2.b.Agencement des registres (*floorplan* des registres).

La même idée est proposée dans la gestion des registres, accessibles rapidement depuis les unités fonctionnelles, et peu coûteux énergétiquement. Ainsi, en rattachant les registres ayant le même type d'activité sur la même branche de l'arbre d'horloge, on peut gagner en puissance.

3.2.c.Ajustement en fréquence.

Une technique particulièrement efficace à ce niveau est «l'ajustement de la fréquence». En effet, l'ajout de multiplieurs et de diviseurs de fréquence permet de modifier la fréquence voir de l'annuler dans certaines unités fonctionnelles. Cette stratégie permet de gagner jusqu'à 50% en énergie [BD97]!

C'est grâce à cette gestion, que les unités fonctionnelles peuvent être mises au repos. Malheureusement, il faut être attentif à ne pas perdre le bénéfice en introduisant un coût trop élevé lors de l'augmentation de la fréquence («au réveil» du circuit) ou en utilisant de nombreuses stratégies de prédiction. De nombreux travaux sont en cours dans cette optique. Citons ceux de [CB99].

Par ailleurs, on peut signaler une amélioration mentionnée dans [BD98]. Quand le résultat d'une unité n'est pas utilisé, on force l'unité à exécuter une instruction fonctionnellement fausse mais moins coûteuse en consommation.

Au niveau d'une unité, la technique la plus étudiée est l'ajustement conjoint en tension et en fréquence. Nous détaillons cette technique dans le paragraphe suivant et décrivons quelques implémentations industrielles.

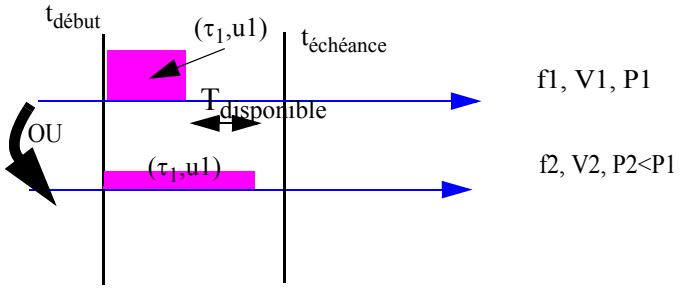
3.2.d.Ajustement conjoint en tension et en fréquence (DVS).

La majorité des processeurs, et quelques accélérateurs matériels, sont dotés de différents modes de fonctionnement. Ces modes sont dits mode normal et «modes basse consommation». En effet, la baisse de la fréquence d'horloge du processeur permet de réduire la tension d'alimentation. Faisons un petit point sur cette caractéristique.

Pour définir cette technique, nous nous basons sur la formule suivante [LJ00] : $T_{min}=kV_{dd}/(V_{dd}-V_{seuil})^2$ (voir figure 6), qui montre le lien entre la période minimum T_{min} de l'horloge et la tension d'alimentation V_{dd} (cf. équation 11). Ainsi, si nous baissons la tension d'alimentation, on obtient la valeur minimale de la période à laquelle le processeur peut fonctionner à cette tension.

La figure 6 illustre le gain en énergie et en puissance qu'on peut espérer par cette technique de DVS (Dynamic Voltage Scaling) appliquée à une unité u_1. La date d'exécution au plus tôt est $t_{début}$ et l'échéance, $t_{échéance}$ indique la date à laquelle la tâche doit impérativement être terminée. $T_{disponible}$ indique le temps disponible qui permet de diminuer la fréquence d'horloge du processeur, et enfin, V_i et f_i sont les différentes tensions d'alimentation et fréquences auxquelles sont exécutées la tache τ_1 sur l'unité u1. P_i est la puissance dissipée obtenue par l'exécution de cette tâche.

FIGURE 6. Ajustement conjoint en tension et en fréquence (Dynamic Voltage Scaling)



Ainsi, au niveau système, l'exploitation du DVS permet de gagner de manière efficace en consommation.

On peut donc profiter du temps disponible pour chaque tâche pour décider de baisser la fréquence du processeur tout en s'assurant que la tâche dure un temps inférieur ou égal au temps qui lui est imparti. En diminuant la fréquence, on peut baisser la tension, ce qui permet de gagner de manière significative en énergie et en puissance.

3.2.e.Gestion des modes basse consommation d'un processeur.

Les processeurs actuels possèdent des modes basse consommation. Explorons-les.

3.2.e.i.Gestion du *Power Management* dans le processeur Intel.

Dans le document [In02] expliquant la gestion du Power Management, pour la famille des processeurs Pentium embarqués, il est défini 5 modes de fonctionnement différents :

- *Normal state* : mode d'opération normal du processeur;

- *Stop Grant State :* il fournit un réveil rapide qui peut être commandé par un signal sur une des entrées du processeur. Le courant fourni est alors diminué de 15% par rapport au courant quand le processeur est dans l'état normal.

- *Auto Halt Powerdown State* : le processeur est mis dans l'état *Auto Halt Powerdown* par l'exécution de l'instruction assembleur HLT. Le courant fourni est alors diminué de 15% par rapport au courant de l'état normal. Dans cet état, le processeur maintiendra seulement les entrées nécessaires (pins) pour gérer la cohérence du cache.

- *Stop Clock Snoop State* : le processeur passe dans ce mode depuis l'un des deux autres modes précédents, grâce à des instructions dédiées qui invalident la gestion de la cohérence du cache. Il gèle alors l'horloge interne du processeur puis maintient seulement les entrées nécessaires. Il n'est pas indiqué le gain en courant obtenu.

- *Stop Clock State* : le processeur entre dans cet état uniquement depuis l'état Stop Grant State. Le courant fourni est alors diminué de 1% par rapport au courant de l'état normal. Dans cet état, l'horloge est arrêtée, on garde toutes les données dans le cache pour ne pas les perdre. C'est durant le temps où le processeur est dans cet état que la fréquence peut être changée, ce qui prend un peu de temps jusqu'à stabilisation vérifiée grâce à une PLL.

Ainsi, ce qui apparaît au niveau système comme étant deux ou trois modes basse consommation, est plus compliqué au niveau du processeur. En effet, celui-ci passe par des états intermédiaires, par exemple, il passe dans un état où l'horloge est arrêtée pour modifier la fréquence. Ceci explique les pénalités en temps et en consommation (dus à la remise sous tension) lors des changements de fréquences.

Comment se fait cette gestion ?

Toujours chez Intel, afin de gérer ces différents modes présentés ci-dessus, une interface a été implémenté :

L'ACPI ou «*Advanced Configuration and Power Interface*» gère la puissance matériellement. Les sociétés Intel, Mitsubishi et Toshiba ont adopté ce standard.

Nous décrivons rapidement l'ACPI ici et nous la décrirons plus en détail dans le paragraphe concernant la partie système. Il nous a semblé important de voir rapidement à ce niveau la réalisation de la gestion de puissance, ce qui nous amène à mesurer dores et déjà l'importance que prennent les systèmes d'exploitation dans cette gestion.

Dans cette interface, on distingue :

- Une interface utilisateur pour gérer la puissance et les configurations.
- Un composant pour la gestion de puissance et la politique de gestion de puissance (*power management and power policy component* (OSPM)).
- Les *drivers* relatifs à l'ACPI (par exemple, drivers pour des contrôleurs embarqués, SMBus, Smart Battery, et Control Method Battery).
- ACPI *Core Subsystem component* qui fournit les services fondamentaux de l'ACPI (comme *l'AML* : interpréteur responsable de l'analyse et de l'exécution du code fourni par le système d'exploitation de l'ordinateur.)

L'objectif est de déplacer les fonctionnalités de contrôle de la puissance du système d'exploitation (*Operating System, OS*) vers cette ACPI pour être indépendant de tout OS.

Voyons d'autres propositions de modes basse consommation de quelques processeurs.

3.2.e.ii.Modes basse consommation du ARM9 [Arm].

Quatre modes sont proposés:

- *Mode normal* : l'horloge du coeur et les horloges des périphériques sont actives et dépendent des exigences de l'application.
- *Mode idle* : l'horloge du cœur est inhibée et attend la prochaine instruction de réveil (une instruction d'interruption ou un *reset* général). Les horloges des périphériques sont actives et dépendent des exigences de l'application exécutée.
- *Mode Slow clock* : cet état est similaire à celui de l'état normal, excepté l'oscillateur principal et la PLL qui sont éteints pour économiser de la puissance.
- *Mode Stand by* : est une combinaison du mode *slow clock* et du mode *idle*. Il permet au processeur de se réveiller très vite au moindre évènement en économisant la puissance.

3.2.e.iii.Processeur à large choix de tensions : le Xscale.

Ce processeur [In00] est réalisé en collaboration entre Intel et Arm. Il est compatible avec le jeu d'instructions ARM Version 5TE ISA (mis à part le jeu d'instructions sur flottant).

Il possède un pipeline de 7-étages pour le traitement des entiers et de 8-étages pour la mémoire. Outre, ses modes repos et endormissement, et sa capacité à effectuer un réveil rapide, ce processeur permet de fonctionner suivant 4 tensions d'alimentation et 11 fréquences.

3.2.e.iv.Le processeur CRUSOE de TRANSMETTA.

Ce processeur est déjà implémenté dans plusieurs *power books* commercialisés.

Les processeurs de cette famille [Tr00] sont compatibles avec le jeu d'instructions des x86. L'architecture est basée sur celle d'un VLIW (*Very Long Instruction Word*). Les dépendances entre instructions ne sont pas gérées par le matériel mais par le logiciel, lors de la compilation. De ce fait, une première économie d'énergie est réalisée. Un processeur CRUSOE est capable d'exécuter jusqu'à 4 opérations à chaque cycle d'horloge. Il dispose de deux unités de traitements en nombres entiers, d'une unité de traitements en nombres flottants, d'une unité de gestion de la mémoire (l*oad/store*) et d'une unité de branchement.

Le caractère basse consommation de ce processeur repose sur deux techniques : il utilise un «*code morphing*» et un «*long run power management»*. Regardons rapidement en quoi consiste chacune de ces techniques.

- **Le *code morphing***

Chaque instruction a une longueur de 64 ou 128 bits, appelée molécule, et contient des instructions type RISC appelées atomes.

Les atomes d'une molécule sont exécutés en parallèle et le format de la molécule détermine la façon dont les atomes sont répartis sur les unités. On peut donc avoir au départ un code superscalaire comme celui des Pentium II et III.

En fait, c'est un système de transformation dynamique (le *translator*) qui agence les instructions d'un jeu (ici type x86) dans un autre jeu (ici VLIW).

Ainsi, le code *morphing* effectue une première passe, où il décode les instructions x86 et les traduit en une séquence d'atomes. Dans une seconde passe, un outil nommé *optimizer* applique les règles d'optimisations de compilation au code, comme l'élimination de sous-expressions communes, de boucles invariantes. Dans la passe finale, le *scheduler* réordonne les atomes et les groupe dans des molécules individuelles. Ce processus est similaire à ce que font les processeurs de manière matérielle. Mais ici, l'utilisation d'algorithmes d'ordonnancement devient possible car l'étape est réalisée de manière logicielle.

Plus le *scheduler* a de liberté, meilleur devrait être le code. Or, la limite principale vient du fait qu'il peut y avoir des dépendances entre opérations impliquant les mémoires.

Pour y remédier, quand le *translator* déplace un *load* devant un *store*, il convertit le *load* en *load-and protect* (qui ajoute au *load* une opération d'enregistrement des adresses et de la taille des données, ce qui évite de les écraser).

Le *translator* repère les parties de code coûtant le plus cher en temps et les optimise. Pour cela, le *translator* ajoute du code qui a pour but de collecter les informations telles que les fréquences d'exécution des blocs et l'historique des branchements (*branch history*). Ces données sont utilisées pour effectuer des optimisations. Par exemple, s'il y a un branchement conditionnel x86 revenant fréquemment, le système peut privilégier des optimisations pour favoriser le chemin le plus fréquemment utilisé. Alternativement, il peut choisir d'exécuter un code de manière "spéculative" sur deux chemins et choisir ultérieurement.

- ***Gestion de puissance : Long Run Power Management [MF01]***

Le gestionnaire de puissance de ce processeur réduit la puissance dissipée en jouant sur des changements de tension et de fréquence.

Les fréquences peuvent changer par pas de 33 MHz et ceci entre 500 et 700 MHz. Les tensions peuvent changer par pas de 25 mV et ceci de 1.2 à 1.6V. Plus la fréquence est basse, plus il est possible de baisser la tension.

Plus de 200 changements de fréquences et de tension peuvent être réalisés par seconde.

Le réglage de ces deux facteurs est effectué en fonction des besoins de performances. Cependant, il n'est pas précisé [MF01] comment cet ajustement de tension est implémenté. Le gestionnaire de puissance détermine les besoins de performance par échantillonnage des périodes de repos (temps où le processeur est dans divers états de *basse consommation*). Il peut détecter les différents scénarios basés sur les performances et les exploiter pour adapter sa puissance en fonction de ces observations. La méthode utilisée est donc une méthode de type prédiction des moments de repos. La puissance de repos «profond» est de 100 mW.

Parmi ces techniques présentées, nous nous sommes particulièrement intéressés à celles d'ajustement conjoint en tension et en fréquence, et à la gestion des modes basse consommation.

Une fois exposées les techniques de réduction d'énergie implémentées au niveau des unités, voyons comment ces fonctionnalités sont exploitées au niveau système.

4•Estimer et réduire la consommation au niveau système

Un processeur est un composant d'un système, aussi, il est nécessaire d'estimer la consommation des autres unités comme les mémoires, les bus, les circuits périphériques.
En effet, l'étude de [PB99], indiquent que les parts de consommation d'énergie dans un système sont de l'ordre de 58% pour le coeur de processeur, 33% pour la mémoire cache, 7% pour les bus du processeur, et 2% pour la SRAM interne. Dans ce système, ne sont pas compris les circuits périphériques. Ces chiffres nous permettent de mesurer l'importance que représente la consommation des mémoires (SRAM et cache). Ces chiffres sont valables pour un ARM8 et ne prennent pas en compte l'ensemble des composants présents dans un système embarqué autonome (exemple du système de la figure 1).

4.1.Estimer la consommation au niveau système.

On décompose la puissance suivant les puissances des unités fonctionnelles mises en jeu, c'est-à-dire les contributions de chacune des n unités de traitement, du

système mémoire $P_{mémoire}$ et de l'interconnexion P_{bus} :

$$P_{total} = \sum_{i = 1, n} P_{unité\ i} + P_{mémoire} + P_{bus}$$

Voyons maintenant comment estimer chacun de ces termes.

Pour l'estimation des bus, certains proposent d'utiliser directement le modèle des capacités comme dans [MR94]. Ce modèle permet, grâce à des règles d'estimer les valeurs des différentes capacités de charge et de décharge des parties du circuit. Les auteurs utilisent un Control Data Flow Graph (représentation graphique de l'application) en modélisant chaque opération par un noeud. Il est possible de déduire des informations comme information le nombre d'accès, le nombre de transferts sur le bus.

Pour extraire les valeurs de capacité des différentes parties du circuit, citons l'approche de Fastcap [FC99], dont l'objectif est de calculer les inductances et les capacités mutuelles d'une structure conductrice décrite en trois dimensions, structure plongée dans un diélectrique. Il est nécessaire de spécifier la discrétisation des surfaces du conducteur et des discontinuités. Les résultats sont fournis sous forme de matrices de capacitance de Maxwell.

Pour le système mémoire, CACTI [SJ01] permet d'extraire les valeurs de capacité de parties de circuit et d'estimer la consommation.

Voyons maintenant diverses techniques qui ont été mises en œuvre pour estimer en particulier les consommations des mémoires et des bus.

4.1.a.Estimation de la consommation des mémoires.

La consommation des mémoires se décompose en consommation dynamique et en consommation statique. La puissance statique dépend de la taille mémoire, car elle est due principalement au courant de fuite, qui est fonction du nombre de transistors. Ici, contrairement au cas des processeurs, elle ne peut pas être négligée.

On peut trouver dans la littérature quelques travaux sur l'estimation de la consommation mémoire. Ce problème étant lié à la technologie, peu de règles générales ont été tirées, nous assistons plutôt à des études de cas dont nous tentons de faire une synthèse dans ce paragraphe.

Considérons les travaux de [KG97a,KG97b,KG98] qui modélisent la consommation d'une mémoire cache (figure 7) de la manière suivante :

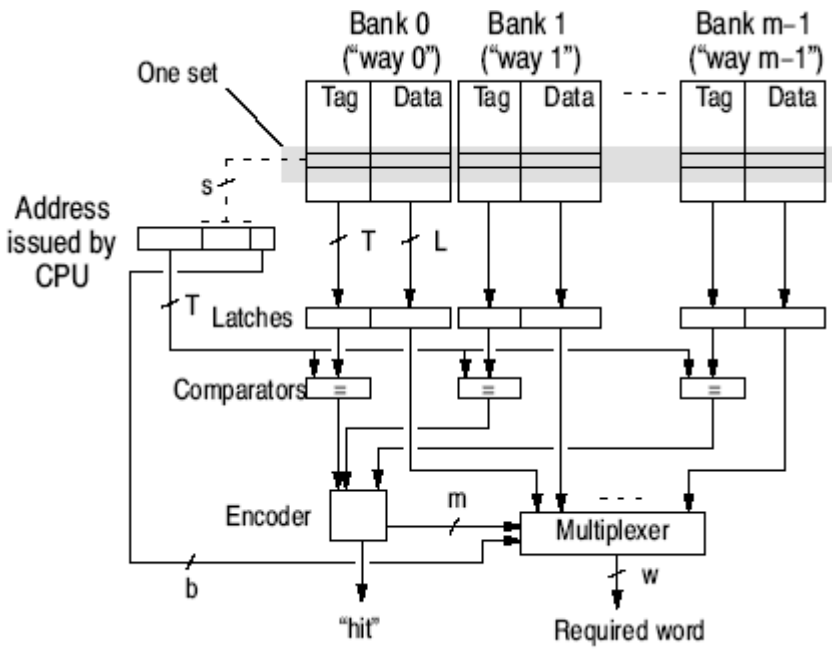


FIGURE 7. cache d'associativité m

Sur la figure 7, L est une ligne de 2^b mots, w est la taille d'un mot, T est le nombre de bits dans un drapeau pour une ligne de cache.

$$E_{cache} = E_{bit} + E_{word} + E_{output} + E_{ainput} + E_{compr} + E_{latch} + E_{sense\_amp}$$

avec :

- E_{bit} : énergie dissipée due aux précharges durant la préparation d'un accès, et pendant la lecture ou l'écriture,
- E_{word} : énergie due à la sélection d'un mot par le driver afin d'exécuter une lecture ou une écriture,
- E_{output} : énergie dissipée au moment de la conduite des données dans les interconnexions extérieures au cache vers le CPU ou les mémoires,
- E_{ainput} : énergie dissipée dans le décodeur de ligne,
- E_{compr} : énergie dissipée dans le comparateur,
- E_{latch} : énergie dissipée dans la bascule latch,
- E_{sense_amp} : énergie dissipée dans l'amplificateur.

Détaillons l'un de ces termes, par exemple E_{word} :

$$E_{word} = V_{dd}^2 . N_{array\_acces} . C_{wordline}$$

avec :

$$C_{wordline} = m.C_{wordline,\ tag\ /\ data} + C_{wordline,\ tag\ /\ status}$$

où $C_{wordline,\ tag\ /\ data}$ est la charge capacitive due à l'accès à la ligne contenant le mot dans le banc de données, $C_{wordline,\ tag\ /\ status}$ est celle due aux accès dans le banc des bits d'état, et $N_{array\_acces} = N_{hits} + N_{misses}$, nombre d'accès au cache, et enfin, m est le degré d'associativité.

Cet exemple illustre la complexité de l'estimation de la consommation d'une mémoire cache, due au nombre de paramètres à prendre en compte. Pour approfondir ces études, notons les travaux de [SD95] et l'étude bibliographique de [BH03] auxquels peut se reporter le lecteur. *Pour estimer la consommation des mémoires, l'étude doit être plus fine. Toutefois, il est envisageable de déduire, des diverses études, des règles de sélection de l'architecture mémoire à sélectionner pour une architecture donnée, dédiée à un type d'applications.*

Intéressons-nous maintenant à la consommation des bus.

4.1.b.Estimation de la consommation des bus.

Généralement, le modèle de consommation d'énergie utilisé pour les bus est du type $E = \frac{1}{2} \times C_{ch} \times V_{dd}^2$, avec C_{ch} la capacité totale de charge du réseau de connexions.

D'après [TJ01], cette formule peut entraîner des erreurs importantes. En effet, elle ne prend pas en compte le fait que certaines lignes seulement du réseau s'activent à un instant. Aussi proposent-ils un modèle basé sur les capacités de couplage entre les lignes parallèles.

Nous avons aussi la possibilité d'exploiter le modèle utilisé par [HL01] qui est proche de celui décrit précédemment. Ce modèle prend en compte les capacités de charge entre le métal et la ligne, appelée capacité de base et la capacité de charge entre deux lignes parallèles et voisines. On fait la somme de ces différentes capacités de charge à chaque fois qu'il y a un événement sur chaque ligne. Ce calcul est donc à réitérer à chaque fois qu'un nouvel événement intervient.

L'énergie est alors $E = \frac{1}{2} \times \sum_{i=0}^{N-1} C_i L_{bus} \times V_{dd}^2$

avec

- C_i la capacité totale par unité de longueur, définie précédemment,
- L_{bus}, la taille totale de chacun des bus mis en jeu (unité de longueur),
- N, le nombre de bus.

Une technique de codage est développée dans [HL01] qui inverse ou non les mots afin de limiter l'activité de commutation.

Par ailleurs, les travaux décrits dans [SB02] ont cherché à extraire un modèle de consommation des interconnexions entre des ensembles de circuits.

Une approche plus pragmatique consiste à obtenir une *netlist* (code VHDL ou Verilog) des interconnexions qui est ensuite exploitée par un estimateur de consommation. Par exemple SONICS propose l'outil SMART Interconnect IP[1], réalisant une interconnexion de type réseau sur puce. Il génère un code VHDL des interconnexions synthétisables, à partir des déclarations des débits, des tailles de données entre les IP (Intellectual Properties). L'outil fournit une *netlist* du bus synthétisable, au niveau porte. En utilisant ensuite cette *netlist* dans un outil d'estimation de la consommation, on peut obtenir une évaluation de consommation des interconnexions par approche statistique ou par simulation.

Cette approche permet de valider un cas d'étude mais ne peut être considérée dans une approche méthodologique de conception système basse consommation.

Voyons maintenant quelles optimisations en consommation il est possible d'effectuer.

4.2.Réduire la consommation au niveau système.

Les systèmes embarqués comme les ordinateurs portables ou les téléphones cellulaires contiennent des dizaines d'entités interconnectées. Il est difficile d'effectuer une optimisation globale en consommation, cependant, c'est bien au niveau système qu'il faut intervenir pour minimiser efficacement la consommation car c'est à ce niveau que les gains les plus importants peuvent être obtenus. Comme vu précédemment, une technique efficace est la gestion dynamique de puissance alliant les mises en veille et/ou en repos des unités et les ajustements de tension et de fréquence. Malheureusement, peu de concepteurs prennent actuellement en compte les gestions des modes basse consommation au niveau système et c'est encore moins vrai pour les outils actuels d'aide à la conception conjointe logicielle/matérielle. Par ailleurs, les industriels ont développé en commun une stratégie de gestion de la puissance appelée *OnNow* comportant la gestion de la puissance par le système d'exploitation et l'ACPI [In02] («*Advanced Configuration and Power Interface*»). Ces aspects sont détaillés ci-dessous.

1. http://www.sonicsinc.com/sonics/products/

4.2.a.Techniques de gestion de puissance au niveau système.

A ce niveau, la gestion de puissance se réalise par des mises en veille de certaines parties du circuit et parfois, effectue des changements de couples <tension, fréquence>. Pour cette première catégorie, la question qui se pose est de déterminer les instants où il est opportun d'effectuer ces mises en veille.

Mesurons ici toute la difficulté de telles mises en œuvre. Plusieurs manières d'opérer ces coupures sont possibles [LX03] :

- «shutdown» appelé aussi «time-out after idleness» qui après un temps seuil déterminé de non utilisation d'une unité la met en mode veille;
- adaptatif : on peut adapter continuellement ce seuil d'après l'historique des utilisations (en estimant la prochaine utilisation par comparaison des temps de veille précédents) ou de manière mathématique (chaînes de Markov exploitant les probabilités d'état et de transitions).

Les problèmes posés par la gestion de puissance sont alors principalement dus :

- aux retards induits par les changements de mode,
- à la puissance dissipée supplémentaire pour changer le mode.

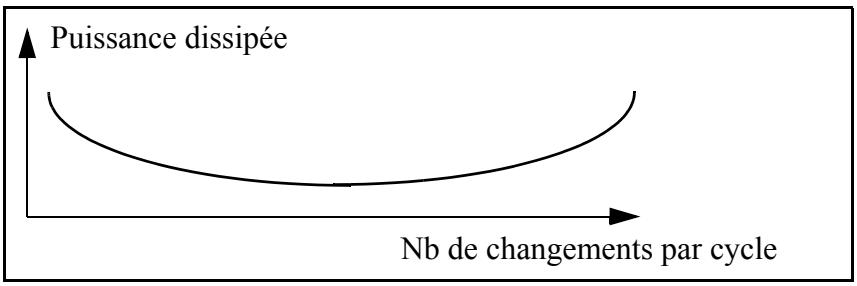
En effet, la mise en mode veille ou la réduction d'alimentation peut coûter plus cher en puissance en raison du coût dû aux changements de régime d'exécution.

Considérons maintenant l'ajustement conjoint en tension et en fréquence. Des problèmes de même nature que ci-dessus se retrouvent pour les changements de fréquence/tension. En effet, le temps dû au changement de vitesse à différentes fréquences a deux sources : le temps nécessaire au calcul de la nouvelle vitesse et le temps nécessaire à sa mise en place (car utilisation d'un *DC-DC switching regulator*).

Certains travaux proposent de repérer si un changement de vitesse est envisageable pour l'exécution de certaines parties de code. Une technique de cette catégorie est basée sur la notion de *Power Management Points* [AM01, MA00]. Dans cette technique, les *Power Management Points* sont des portions de code introduites pour gérer l'exécution de segments de programme afin de décider d'un changement de la vitesse du CPU. Leur étude montre qu'il existe un nombre optimal de ces points. En effet, on se rend compte qu'avec peu de points, il n'est pas possible de profiter efficacement des réductions d'énergie potentielles et avec un trop grand nombre de points, les change-

ments de mode trop fréquents entraînent un surcoût prohibitif, comme le montre la figure (figure 8) ci-dessous :

FIGURE 8. Lien entre le nombre de changement de fréquences et le gain en puissance.



Le problème se pose de savoir comment déterminer le nombre optimal de points. Retenons seulement que les auteurs annoncent 90% de gain par rapport à un système sans gestion de puissance et 60% de gain pour un système qui modifie de manière dynamique les vitesses par rapport à un système à vitesse unique et mode veille.

Des travaux récents [ZC03] perfectionnent cette approche en introduisant une technique adaptative de «points de vérification» dans la gestion de puissance par ajustement conjoint de tension et de fréquence. Leur but est de respecter les dates impératives de fin des tâches en minimisant l'énergie. Ils se placent dans la situation où ils peuvent ajuster la tension et la fréquence au cours de l'exécution d'une tâche. La technique adaptative de «points de vérification» est basée sur une étude statistique du taux de défaut de fin des dates des tâches : ils supposent que chaque tâche a une probabilité (loi de Poisson) de ne pas être réalisée en temps voulu. Ils déduisent un nombre seuil de points de vérification. Un algorithme a été développé qui détermine dynamiquement des intervalles de temps où la possibilité de changer la fréquence et la tension du processeur est vérifiée (même au milieu d'une exécution de tâche). Leurs résultats sont encourageants, et cette approche très fine peut être adaptée à la gestion de puissance au niveau système.

4.2.b.Gestion de la puissance par le système d'exploitation.

On considère ici que le système d'exploitation est le gestionnaire, le coordinateur de toutes les entités de l'architecture, y compris les différents modes de fonctionnement du processeur.

Un exemple de mécanisme adossé à un système d'exploitation est la proposition commerciale OnNow [Mi97] qui exige que :

- Le PC soit prêt dès que possible quand l'utilisateur le met en route.

- Le PC apparaisse comme éteint quand il n'est pas utilisé, mais il doit être capable de se réveiller au moindre évènement, qu'il soit dû à l'utilisateur, ou à tout autre périphérique.
- Le système d'exploitation et les applications doivent «coopérer» pour rendre efficace la gestion de puissance. Par exemple, les applications ne doivent pas être conçues en considérant que les parties matérielles seront toujours alimentées et pleinement fonctionnelles.
- Si une application n'est pas active, elle doit informer le système d'exploitation de son état pour lui permettre de contrôler les ressources qui sont inutilisées.
- Toutes les unités matérielles contribuent à la réduction de la puissance en étant capable de répondre aux commandes de gestion de puissance d'une manière efficace.

Ainsi, le système d'exploitation possède un rôle primordial dans la gestion de la puissance. Il prend en compte à la fois les exigences des applications en terme de ressources et de leur disponibilité. C'est lui qui assure l'optimisation de la puissance dans les ordinateurs actuels. En fait, pour communiquer avec le système d'exploitation, les applications passent par l'intermédiaire d'une API.

L'API de l'OnNow fournit :

- une interface qui permet d'interroger le système d'exploitation sur les états et les évènements liés à la puissance. Quand le système d'exploitation prépare un état de mise en veille, un message est envoyé aux applications pour qu'elles réagissent de manière coordonnée. Plusieurs actions doivent être décidées comme par exemple sauver des informations ou informer le système d'exploitation que cette mise en veille est impossible.

- un mécanisme pour informer le système d'exploitation sur les exigences spécifiques qui peuvent influer sur la gestion de puissance. Les fonctions de base requises par ce type d'interaction sont la commande prévenant le système d'exploitation qu'une ressource peut être mise en repos et une commande pour forcer le système en mode veille.

En résumé, la figure ci-dessous présente les interactions entre les différents modules :

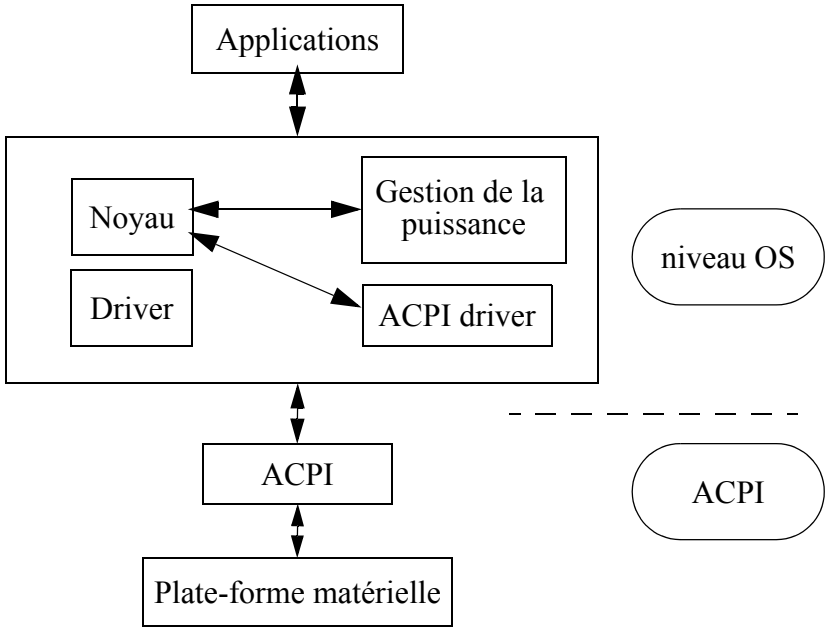


FIGURE 9. Gestion de la puissance et systèmes d'exploitation [BD98].

Voyons maintenant d'autres formes d'économie de consommation.

4.2.c.Gestion des batteries.

La gestion des batteries est devenue cruciale pour les systèmes portables. Nous ne discutons pas des technologies de batteries, signalons seulement que le matériel communique avec le système d'exploitation pour le prévenir par exemple que les batteries sont pratiquement déchargées. C'est à lui ensuite d'envoyer tous les signaux nécessaires pour sauvegarder les informations avant...extinction... Outre cette gestion simplifiée, il serait possible de tenir compte dynamiquement de l'énergie disponible dans la batterie pour maximiser la QoS (Quality of Service) et optimiser la durée de fonctionnement du système. Par exemple**,** les travaux présentés dans [LJ01] portent sur la gestion d'énergie de batterie en gérant d'une part, les moments de charge et de décharge, afin que le temps de décharge corresponde à la durée d'un cycle d'exécution et, d'autre part, l'optimisation de cette durée par le DVS. Les auteurs ont développé un algorithme cherchant à tout moment à augmenter la vitesse des tâches, afin d'être en adéquation avec la durée de la batterie. De même, cet algorithme vise à repérer les instants où les besoins en énergie ou en puissance sont importants.

4.2.d.Optimisation de la consommation des mémoires.

Rappelons que la consommation des mémoires se décompose en consommation dynamique et statique et que cette dernière dépend de la taille mémoire, car elle est due principalement au courant de fuite, qui est fonction du nombre de transistors. Dans le cas des mémoires, la consommation statique n'est pas négligeable. En effet, d'après les études développées dans [FC98], la mémoire représente de 20 à 80% de la consommation globale, dont la moitié dissipée par le décodage et les buffers d'adresses. Ainsi, les auteurs [FC98] ont développé une méthode d'optimisation de la consommation de la mémoire, DTSE, *Data Transfert Storage Exploration*. Cependant, le temps nécessaire à l'obtention d'une solution optimisée est relativement important car, de nombreuses étapes de la méthode ne sont pas automatisées.

D'autres travaux ont été entrepris cherchant à intégrer les contraintes liées à la mémorisation tout au long du flot de conception, afin de rechercher un compromis entre le temps de conception et la qualité de la solution. Ainsi, les auteurs [GN02] ont élaboré une stratégie qui consiste à identifier les paramètres mis en jeu et à définir les phases du flot de conception durant lesquelles ces paramètres ont le plus d'influence sur la consommation. Ainsi, ces paramètres sont les transferts de et vers la mémoire, le nombre de données "vivantes" à un instant donné et le nombre de transitions sur les bus d'adresses. La stratégie concerne ensuite les choix des mémoires. Elle se compose de deux phases principales :

- **Phase d'agencement de la hiérarchie mémoire,** à savoir la sélection du nombre de niveaux de la hiérarchie en fonction de la taille de chaque niveau de la hiérarchie et des coûts en temps et en consommation :

  - choix de la taille de chaque niveau,
  - choix du nombre de ports,
  - distribution des données dans les niveaux.

- **Phase de conception d'un niveau de la hiérarchie mémoire :**

  - gestion de la cohérence des données,
  - sélection du type de mémoire le mieux adapté (temps, surface, consommation),
  - distribution des données,
  - placement et génération des adresses (affectation d'une adresse précise à chaque donnée).

5• Conclusion

Ce chapitre nous a permis de rappeler les techniques actuelles d'estimation et d'optimisation de la consommation du niveau technologique au niveau système.

Certaines de ces techniques ont été considérées dans la suite de nos travaux.

Malgré l'importante littérature sur le sujet de la réduction de la consommation d'énergie, il apparaît que de nombreuses avancées sont encore nécessaires pour faire face aux défits liés aux évolutions de la technologie. Ces avancées sont rendues difficiles du fait de la complexité des problèmes soulevés à tous les niveaux de conception d'un système.

Après cette analyse des techniques d'estimation et de réduction de la consommation, voyons dans le chapitre suivant, les techniques actuelles d'aide à la conception conjointe logicielle/matériel.

Chapitre III : Synthèse d'un système autonome basse consommation

Introduction

Notre travail concerne la conception globale de systèmes sur puce basse consommation. Dans cet objectif, ce chapitre présente des méthodes d'aide à la conception conjointe logiciel/matériel. Ces méthodes sont en général basées sur des techniques d'allocation et d'ordonnancement. Nous indiquons les liens réalisés entre ces dernières techniques et l'optimisation en consommation, que ce soit au niveau du partitionnement ou au niveau de l'ordonnancement.

Enfin, nous détaillons plus précisément l'outil d'exploration d'architecture CODEF qui a servi de support à nos travaux.

1•Le partitionnement dans la conception conjointe logiciel/matériel

Donnons un aperçu des techniques de conception conjointe logiciel/matériel en situant le rôle du partitionnement.

1.1.Conception conjointe logiciel/matériel.

Cette approche cible la conception de systèmes à base d'architectures logicielles (processeurs...) et matérielles (ASIC, ASIP, FPGA). Encore aujourd'hui, la conception de chacune de ces entités se fait séparément pour être ensuite assemblées. Parmi les systèmes qui pourraient bénéficier d'une approche de conception conjointe, on peut citer les applications de télécommunication qui sont de plus en plus complexes et où le choix des unités matérielles et logicielles est de plus en plus important.

Des techniques permettant la conception de tels systèmes ont été développées dans la littérature afin d'aider les architectes système dans leur choix d'implémentation des différentes fonctionnalités de l'application sur les diverses unités dont ils disposent. Par exemple, le but des techniques d'exploration d'architectures est de proposer au concepteur plusieurs solutions possibles d'architectures, selon des critères précis. Le concepteur peut être aussi amené à faire des choix en fonction des contraintes imposées par l'application.

Nous nous intéressons dans la suite aux principes développés dans les approches classiques. A chaque fois se posent les questions de la modélisation, de la spécification des applications, et, du niveau de granularité le mieux adapté.

1.2.Partitionnement logiciel/matériel.

En général, les facteurs pris en compte de manière prioritaire pour le partitionnement des tâches d'une application sont les contraintes temporelles, la surface de silicium et maintenant, de manière incontournable pour tout système embarqué, la consommation. Les contraintes temporelles concernent les dates d'occurrence et/ou les périodes des évènements de réveil des tâches, et les échéances sur les fins d'exécution des tâches. La contrainte de surface a pour objectif de limiter le coût du produit final. On cherche en général à minimiser ce facteur. D'autres contraintes s'ajoutent à ces contraintes physiques : les tâches possèdent en général des dépendances ce qui implique des communications et des synchronisations avec les partages des ressources matérielles associées. Du fait de tous ces paramètres et des objectifs d'optimisation, le problème du partitionnement est un problème NP-complet. Nous présentons ci-dessous des méthodes classiques, ne prenant pas en compte

la consommation. Dans la suite, nous indiquons celles qui peuvent s'adapter à cette contrainte.

Les méthodes classiques de partitionnement sont constituées de deux étapes corrélées : une étape d'allocation et une étape d'ordonnancement.

Parmi les techniques de partitionnement les plus répandues, nous pouvons citer celles basées sur des heuristiques, des algorithmes génétiques, la programmation linéaire en nombres entiers, les algorithmes à recuit simulé ou encore le partitionnement assisté par des techniques d'évaluation de performances.

Avant d'entrer dans le vif du sujet, rappelons rapidement quelques modèles de spécification des applications utilisés.

Le choix du modèle est souvent guidé par le type de l'application ou de la classe d'applications ciblée. Pour les applications orientées contrôle, les modèles de type Esterel [BG88], Grafcet [IE88], Statecharts [HD87], sont bien adaptés.

Les applications orientées traitement sur des flots réguliers d'informations se modélisent efficacement par des réseaux de processus de tâches [KG74], ou le langage Lustre [HC91]. D'autres modèles peuvent également être considérés comme les réseaux de Petri [MT89] ou même directement un langage de programmation de type C ou VHDL peut être utilisé [EH93].

Pour les outils développés au sein de l'équipe (et particulièrement l'outil CODEF), le choix a été fait d'utiliser les graphes de flots de données (*Data Flow Graph* : DFG), car [BL99] :

- il spécifie efficacement des opérations sur des flots de données,
- il est possible de spécifier le système à plusieurs niveaux,
- il est possible de vérifier les propriétés du système,
- l'utilisation de ce formalisme se prête bien à la co-conception logicielle/matérielle,
- sa nature descriptive est graphique.

Ainsi, à partir de telles représentations des applications, les méthodes de partitionnement déterminent des répartitions des fonctionnalités de l'application sur les entités de l'architecture, sélectionnées ou non par la méthode. Dans la suite nous présentons quelques unes de ces méthodes.

1.2.a.Partitionnement logiciel/matériel guidé par le chemin critique.

Un exemple de ce type d'approche est proposé dans [BM97]. Les principes reposent sur le calcul du chemin critique d'un graphe de type DFG. L'architecture visée est hétérogène et connue à l'avance. Elle est constituée d'éléments tels que des ASIC, des processeurs et des bus. Le but de l'algorithme est de déterminer une allocation et un ordonnancement des tâches qui minimise le temps d'exécution de l'application, en

tenant compte des interconnexions de type bus entre les unités de traitement et en respectant les dépendances de données. Ainsi, le chemin critique d'une tâche exprime le plus long des chemins (exprimée en temps), partant de cette tâche jusqu'à un nœud terminal. Cette approche a été étendue dans CODEF qui est détaillé plus loin.

1.2.b.Partitionnement logiciel/matériel par algorithmes génétiques.

Les algorithmes génétiques se basent sur une procédure itérative durant laquelle un ensemble de générations (ou populations) est créé, une par itération. L'entière population évolue simultanément de façon à ce que la probabilité de convergence vers un minimum local soit réduite. Après avoir défini un codage des individus de la population afin de représenter les solutions potentielles, la procédure d'évolution est constituée de quatre étapes :

- une étape d'évaluation,
- une étape de sélection,
- une étape de génération,
- une étape de renouvellement.

Les étapes de sélection, de génération, de renouvellement sont assez génériques vis-à-vis des différents problèmes à optimiser. L'étape d'évaluation est plus spécifique, étant donné qu'elle doit définir une mesure de la qualité de chaque individu. Dans le cas de la conception de SOC, cette qualité peut être évaluée comme une combinaison de plusieurs critères : contraintes de temps et de consommation en énergie dans [DJ98, PG02] ou d'un seul critère : temps d'exécution minimum [BK02]. La technique utilisée pour mesurer cette qualité repose pour ces trois approches sur un algorithme d'ordonnancement.

Un aspect intéressant de l'approche par algorithme génétique est leur faculté «naturelle» à effectuer une exploration de l'espace de conception qui permet en particulier d'obtenir un ensemble de points de Pareto correspondants à des solutions optimisées suivant plusieurs critères [PG02]. Leurs inconvénients reposent d'une part sur les temps de convergence qui peuvent être longs et d'autre part sur la difficulté à «rejouer» un scénario pour évaluer par exemple l'apport d'une évolution de l'architecture ou d'une modification de la spécification.

1.2.c.Partitionnement logiciel/matériel par programmation linéaire en nombres entiers.

Le but est là encore de trouver des solutions architecturales optimales. Citons les travaux dans ce domaine de [NM96]. Les auteurs ont développé une stratégie en deux étapes :

- une étape d'allocation des nœuds en logiciel ou matériel en fonction de l'urgence temporelle,
- une étape précisant l'ordonnancement grâce aux informations de partitionnement de la première étape.

L'architecture peut être composée de plusieurs processeurs et d'un accélérateur matériel (ASIC). La spécification du système est fournie à l'algorithme sous une forme VHDL. Celle-ci est transformée en une représentation graphique interne. Un outil de compilation est utilisé pour déterminer les coûts logiciels et un outil de synthèse pour déterminer les coûts matériels. La prise en compte de ces estimations de coûts et des contraintes de conception permet de générer un graphe, contenant toutes les informations nécessaires au partitionnement. Les coûts utilisés par l'approche ILP sont calculés de façon indépendante pour chaque nœud. Aussi, le regroupement des nœuds, migrant du matériel vers le logiciel, nécessite de recalculer ces coûts. L'algorithme réitère jusqu'à ne plus trouver de solution, la dernière partition représente alors la solution. L'approche ILP, pour l'ordonnancement des nœuds, peut conduire à des temps de calcul très longs. Aussi, l'ordonnancement est dans un premier temps effectué avec une heuristique. Une fois les choix du partitionnement connus, l'approche ILP recalcule avec précision les dates d'ordonnancement des tâches. S'il y a violation de la contrainte temporelle, la seule possibilité est alors d'augmenter cette dernière si l'application le permet. Cette technique itérative est très gourmande en temps de calcul.

1.2.d.Partitionnement avec la méthode MCSE [CJ90,MF00].

Réaliser du partitionnement avec MCSE (Méthodologie de Conception des Systèmes Electroniques) nécessite de disposer d'une composante structurelle (décrite par le modèle fonctionnel), d'une composante comportementale (qui décrit temporellement les constituants du modèle fonctionnel), d'une composante exécutive (qui exprime les constituants matériels et les interconnexions nécessaires pour le fonctionnement du système), et la dernière composante décrit les moyens informatique et électronique à mettre en oeuvre pour atteindre l'objectif.

De nombreux choix de conception sont laissés au concepteur. Cependant, une fois ces choix déterminés, la structure d'exécution est définie, avec les tâches matérielles et logicielles et leur support. La vérification se fait en utilisant un modèle de performance qui permet d'obtenir, par simulation, les propriétés de performance du système logiciel/matériel issu du partitionnement. Toutefois, cette méthode ne permet pas de réaliser une exploration automatique d'un espace de solutions.

1.2.e.Partitionnement logiciel/matériel par algorithmes à recuit simulé

Ce type d'algorithme permet d'obtenir une solution à un problème d'optimisation combinatoire si la qualité de la solution cherchée peut être évaluée par une fonc-

tion de coût. Ceci est le cas du partitionnement logiciel/matériel comme cela a été montré par [EH93, EP97, LL03, MB03]. Pour appliquer une technique d'optimisation par recuit simulé, il est nécessaire de définir la fonction de coût dans le but d'évaluer la qualité d'une solution et de guider l'algorithme. Par ailleurs, il est nécessaire d'exprimer les contraintes de conception qui définissent l'espace de conception dans lequel la recherche de la solution s'effectue. Le principal avantage du recuit simulé réside dans sa généralité à traiter ce type de problème d'optimisation. Son inconvénient majeur est le temps de calcul [LL03].

1.2.f.Partitionnement logiciel/matériel par algorithmes de partitionnement à granularité dynamique.

Les résultats du partitionnement logiciel/matériel sont très sensibles à la granularité utilisée pour décrire l'application. Schématiquement, si l'application est décrite avec une granularité fine, on peut espérer une meilleure optimisation du fait du plus grand nombre de possibilités offertes, mais la complexité qui en résulte empêche l'utilisation d'algorithmes efficaces. A contrario, une forte granularité limite les choix, permettant la recherche de solutions optimales pour ce niveau de granularité. Une approche à granularité variable [HE96] essaie d'évaluer pendant le partitionnement le meilleur niveau de granularité exploitable pour effectuer le partitionnement. L'application est partitionnée une première fois, puis à chaque itération, on cherche à regrouper des tâches en évaluant un facteur de qualité (selon les critères à privilégier). Leur algorithme opère sur un code de type C, le partitionnement est une permutation des blocs de l'application qui passent d'une allocation matérielle à une allocation logicielle et vice-versa. Les auteurs donnent une méthode de regroupement de blocs avant l'étape de calcul de destination précise.

Regardons maintenant les approches développées dans la phase d'ordonnancement du partitionnement et les algorithmes associés les plus classiques.

2•L'ordonnancement

L'ordonnancement est en général incontournable dans le partitionnement car il est nécessaire d'une part de définir un ordre d'exécution des fonctionnalités de l'application sur chaque entité de l'architecture et, d'autre part, d'évaluer la qualité d'une solution. L'ordonnancement est un problème qui a fait l'objet de très nombreux travaux. Il a pour but de déterminer une stratégie d'exécution d'un ensemble de tâches. Dans un premier temps, présentons les caractéristiques des tâches généralement considérées.

Les contraintes temporelles des tâches sont classiquement de deux sortes :

- *Hard*, la tâche est dite à contrainte «dure» si un dépassement de son échéance, engendre des conséquences catastrophiques sur le système. Dans ce cas, il faut que l'implémentation de la tâche puisse garantir sa réalisation temps réelle dans le pire cas.
- *Soft*, la tâche est dite à contrainte «souple» si un dépassement d'échéance n'est pas fatal vis-à-vis du système.

Ces caractéristiques orientent les algorithmes d'ordonnancement qui doivent à tout prix respecter les échéances dures, en profitant cependant des libertés temporelles possibles pour optimiser l'architecture du système.

Les algorithmes d'ordonnancement sont généralement caractérisés par les propriétés suivantes [GC02] :

- **préemptif** : une tâche en cours d'exécution peut être interrompue pour permettre au processeur d'exécuter une autre tâche active plus prioritaire.
- **non-préemptif** : une tâche non préemptive, une fois démarrée, est exécutée par le processeur jusqu'à la fin de son exécution. Dans ce cas, toutes les décisions d'ordonnancement sont prises à la fin de l'exécution de la tâche en cours.
- **statique** : ces algorithmes basent leurs décisions d'ordonnancement sur des paramètres fixes, attribués à chaque tâche avant leur activation.
- **dynamique** : ces algorithmes basent leurs décisions d'ordonnancement sur des paramètres dynamiques changeant pendant l'évolution du système.
- **hors-ligne** : il est dit d'un algorithme qu'il est hors-ligne quand l'ordonnancement de l'ensemble des tâches est réalisé, fixé avant leur activation. L'ordonnancement généré est placé dans une table qui est utilisée à l'exécution pour activer les tâches dans l'ordre qui a été calculé.
- **en ligne** : il est dit d'un algorithme qu'il est en ligne quand les décisions d'ordonnancement sont prises à chaque fois qu'une nouvelle tâche est susceptible d'être exécutée ou quand une tâche est terminée.
- **optimal** : il est dit d'un algorithme qu'il est optimal s'il minimise une fonction définie pour l'ensemble des tâches. Quand aucune fonction coût est définie et qu'est recherché seulement un ordonnancement faisable, un algorithme est dit optimal si aucun autre algorithme basé sur les mêmes hypothèses ne trouve de meilleurs résultats.

De nombreuses applications sont constituées d'un ensemble de tâches à exécuter sur des requêtes périodiques.

Dans le cas de tâches périodiques et indépendantes, deux techniques classiques d'ordonnancement en ligne sont le *Rate Monotonic* (RM) et l'*Earliest Deadline First* (EDF). Décrivons-les rapidement.

- le RM (*Rate Monotonic*) est à priorité fixe, il donne la priorité de la tâche à exécuter en fonction de sa période. Il attribue la plus grande priorité à la tâche qui possède la plus petite période.
- l'EDF (*Earliest-Deadline-First*) est à priorité dynamique, il, donne la priorité la plus élevée à la tâche ayant sa date d'échéance la plus proche dans le temps.

Le RM est optimal pour un algorithme d'ordonnancement attribuant des priorités de manière fixe, et l'EDF est optimal pour un algorithme d'ordonnancement à priorité dynamique [GC02].

De nombreux travaux sur l'analyse d'ordonnançabilité existent pour ces algorithmes et leurs dérivés [GC02]. Ils sont en général limités au cas monoprocesseur. Le passage au cas multiprocesseurs pose des difficultés importantes. Par ailleurs, l'analyse d'ordonnançabilité a été étendu par exemple au cas des tâches dépendantes (pour s'exécuter, une tâche i attend des données d'une tâche j), ou des tâches partageant des ressources communes. Les méthodes d'analyse ont été utilisées dans le partitionnement pour des architectures distribuées, par exemple [TW95].

L'ordonnancement hors-ligne a été très largement utilisé dans des méthodes de partitionnement. On peut citer l'adaptation de l'ordonnancement par liste dans [KL94], l'ordonnancement suivant le chemin critique [BM97] ou l'ordonnancement dirigé par les forces [PK89] qui est une extension de l'ordonnancement par liste. L'avantage de l'approche hors-ligne par rapport aux approches en ligne est de faciliter l'analyse et l'évaluation des solutions construites. Elle permet également de minimiser le coût lié aux mécanismes à intégrer dans le système pour prendre les décisions en ligne mais possède une adaptabilité réduite vis-à-vis de leur environnement.

Regardons maintenant ce qu'implique d'ajouter le critère de la puissance consommée à des approches de conception conjointe logiciel/matériel.

3•Le Partitionnement basse consommation

Nous présentons ci-dessous quelques méthodes et outils de partitionnement prenant en compte la consommation.

Rappelons les facteurs importants qui interviennent dans la consommation : $P = \alpha \times f_{clock} \times C_{ch} \times V_{dd}^{2}$, avec α, l'activité de commutation, f_{clock}, la fréquence d'horloge, C_{ch}, la capacité de charge et V_{dd}, la tension d'alimentation.

3.1.L'approche MOCSYN [DJ99] .

Cette méthode de partitionnement est basée sur un algorithme génétique, qui opère sur un graphe de tâches périodiques. Ce graphe est déroulé sur une hyperpériode correspondant au plus petit commun multiple des périodes des tâches. L'algorithme est structuré en sept étapes :

- 1.Détermination de la fréquence d'horloge de chaque unité assurant un compromis entre temps d'exécution et puissance. Chaque fréquence est obtenue par la division d'une seule fréquence globale.
- 2. Initialisation : chaque tâche doit avoir au moins un type d'unité capable de l'exécuter (processeur, coprocesseur...).
- 3. Allocation des tâches à chaque cœur en minimisant les conflits sur les bus.
- 4. Définition des types de bus nécessaires (volume des données et date limite de transmission).
- 5. Estimation d'un placement des cœurs afin d'évaluer les délais, la puissance et la surface de silicium.
- 6. Assignation de chaque évènement de communication à un bus.
- 7. Ordonnancement des tâches sur les cœurs et les évènements de communications sur les liens de communication.

Les étapes 3 à 7 se répètent lors de l'optimisation : croisements, mutations... réordonnancements et estimations jusqu'à trouver des solutions respectant les contraintes. Cette optimisation est réalisée en prenant en compte les délais, la puissance et la surface de silicium calculés à l'étape 5.

La puissance est prise en compte en ajoutant l'énergie de l'exécution de chaque tâche, l'énergie globale de la distribution de l'horloge et l'énergie du réseau de communication. Ces deux dernières valeurs sont évaluées en tenant compte des longueurs de connexion, des nombres de transitions pendant une hyperpériode et d'un facteur d'énergie de l'horloge. Cet algorithme de partitionnement ne permet pas de faire varier les fréquences et tensions d'alimentation dans les unités, facteur primordial pour réduire la consommation d'énergie au niveau système.

3.2.L'approche COSYN [BL97] .

Cette approche est basée sur une méthode de *clustering* et opère sur un graphe de tâches. Un *cluster* de tâches est associé à une unité matérielle ou logicielle. Les tâches sont groupées en *clusters* selon leur priorité et leurs dépendances.

Les priorités sont calculées selon les contraintes temporelles (échéance de chaque tâche), les temps d'exécution des tâches et les temps de communications entre les tâches.

Le calcul se fait en partant des tâches feuilles du graphe et en remontant vers les tâches racines. La procédure de construction des *clusters* opère itérativement à partir de la tâche la plus prioritaire non affectée à un *cluster*. Cette tâche est associée au *cluster* compatible tel que les tâches concurrentes sont placées de façon privilégiée dans le même *cluster* et la taille du *cluster* est inférieure à une valeur limite correspondant au taux d'utilisation de la ressource.

L'algorithme ordonnance ensuite les *clusters* de tâches sur les processeurs. L'ordonnancement peut être avec ou sans préemption. Que l'ordonnancement donne une solution valide ou pas, l'algorithme explore d'autres allocations de *clusters*, ce qui induit des modifications sur les priorités des tâches dans les *clusters*. Une fois toutes les allocations de *clusters* explorées, la solution la moins coûteuse est sélectionnée.

L'outil COSYN a ensuite été modifié (COSYN-LP) pour prendre en compte la consommation de manière dynamique lors du partitionnement. Ainsi, sont modifiées les priorités par rapport à la consommation. La puissance est prise en compte par des valeurs moyennes de puissance dissipée par les tâches et leurs communications. Les tâches les plus coûteuses sont regroupées prioritairement.

COSYN-LP permet un gain en puissance dissipée de 33% en moyenne sur divers tests par rapport à la version précédente.

L'intérêt de COSYN-LP est de prendre en compte les puissances de dissipation sur les différentes unités disponibles, et d'explorer différentes solutions. Cependant cette approche ne permet pas de faire varier les fréquences et les tensions d'alimentation des unités.

Voyons un autre exemple de partitionnement basse consommation.

3.3.L'approche IMPACCT [LC01,CB94] .

Cette approche a pour objectif d'optimiser la consommation au niveau système. A partir d'une représentation sous forme d'un graphe de tâches sur lequel sont annotées les contraintes de temps et de consommation, la méthode proposée procède en trois étapes dans l'ordonnancement et le partitionnement, nommées *timing scheduling*, *max power*, et *min power*.

La principale étape de «*timing scheduling»* recherche une allocation et un ordonnancement en respectant les contraintes de temps. La deuxième étape d'ordonnancement, *max-power*, se focalise sur les contraintes de puissance maximum. Si ces contraintes ne sont pas vérifiées, un nouvel ordonnancement met en série certaines tâches, afin de réduire les pics de puissance en minimisant le recouvrement d'exécution des tâches. Si la contrainte de puissance n'est toujours pas respectée, alors on

analyse quelles sont les tâches qui peuvent être ralenties. Si on ne respecte pas les contraintes, on refait toutes les étapes afin de trouver une autre allocation.

Quand l'outil est assuré d'avoir une solution, il réarrange, dans l'étape d'ordonnancement *min-power*, les tâches pour obtenir la puissance minimale. Cette approche d'ordonnancement basse consommation prend en compte les modes actif et veille. Voyons un dernier exemple de partitionnement basse consommation.

3.4.Une autre méthode exploitant les modes basse consommation.

Les modes basse consommation, par exemple les modes *sleep* et *idle*, permettent de gagner en puissance et en énergie. Il est donc important de les exploiter. C'est l'idée qui est développée dans les travaux relatés dans [DA03]. A partir de premiers résultats d'allocation/ordonnancement, l'objectif est d'essayer de regrouper au maximum les exécutions de tâches sur une même unité. Le but est de libérer au maximum des unités pendant une durée permettant de mettre l'unité dans un des états de repos. En effet, comme nous l'avons vu précédemment, le gain en énergie et en puissance est significatif quand la durée est suffisante pour mettre une unité en état de repos, sans que le coût du réveil ne réduise le bénéfice obtenu par le repos. Ces mises hors tension permettent en particulier de réduire la puissance statique, ce qui est d'autant plus intéressant qu'on cible des technologies sub-microniques.

Ainsi, l'allocation et l'ordonnancement sont modifiés. L'optimisation est réalisée grâce à un algorithme génétique. Sur une application téléphonique, un gain de 67% est obtenu.

Voyons maintenant des techniques qui ont été particulièrement développées pour réaliser des ordonnancements basse consommation.

4•L'ordonnancement basse consommation

Dans cette partie, nous décrivons diverses techniques d'ordonnancement prenant en compte la consommation. Nous analysons les liens éventuels entre les techniques d'ajustement de tension et de fréquence, ou entre la gestion des modes basse consommation et l'ordonnancement.

4.1.Travaux relatifs à un ordonnancement dans le cas monoprocesseur.

4.1.1.Politique dynamique.

- **L'ordonnancement stochastique.**

L'approche décrite dans [GN03, GNC03] concerne l'ordonnancement stochastique en politique dynamique. Son principe est de faire démarrer l'exécution d'une

tâche à la vitesse du processeur et de l'augmenter, si nécessaire, graduellement au cours de l'exécution de la tâche de façon à respecter son échéance.

- **L'ordonnancement avec instrumentation du code .**

Cette technique [GN03, DK01, MA00], a déjà été mentionnée au chapitre deux (*Power Management Points*), sur les optimisations en consommation possibles au niveau système. L'idée est d'évaluer en ligne le pire temps d'exécution restant et de modifier en conséquence la vitesse pour la tâche analysée ou même éventuellement la vitesse d'autres tâches en attente d'exécution. Pour ce faire, le code d'une tâche est divisé en sections pour lesquelles les WCET (*worst case execution time)* sont connus. La vitesse du processeur est recalculée en-ligne après chacune de ces sections en fonction de la différence entre le temps d'exécution effectif et le WCET. Plus l'exécution du programme progresse et plus la connaissance du temps d'exécution restant est fine et donc plus la vitesse choisie sera proche de l'optimal. Une des difficultés majeures est de choisir la bonne granularité d'instrumentation car le temps de commutation de vitesse et les instructions supplémentaires rajoutées lors de la phase d'instrumentation du code pour la mesure des temps d'exécution et le calcul des vitesses peuvent induire une surcharge supérieure aux gains potentiels. Pour pallier ce problème, des techniques basées sur la notion de *Power Management Points* [AM01, MA00] peuvent être appliquées.

- **Autres travaux en ordonnancement basse consommation.**

L'objet du travail de [PS01] est de développer des algorithmes intégrés dans l'ordonnanceur des systèmes d'exploitation temps réel pour fournir un gain notable en énergie en respectant les contraintes de temps. Dans le cas dynamique, deux techniques sont présentées et appliquées à l'EDF (*Earliest-Deadline-First*) et au RM (*Rate Monotonic)*. La première consiste à exploiter le fait que les tâches ne s'exécutent pas toujours en un temps correspondant à leur pire cas. Dans ce cas, on dispose d'un budget en cycles (temps) jusqu'à la prochaine invocation de la tâche. Ce budget peut être reporté sur les autres tâches afin d'en profiter pour baisser la fréquence et la tension.

La deuxième technique consiste à délayer le plus possible la charge de calcul de la tâche en faisant l'hypothèse que le temps d'exécution effectif de la tâche sera inférieur à celui de son pire cas. Si cette hypothèse n'est pas vérifiée, il est nécessaire d'augmenter la fréquence de manière à ce que la tâche respecte son échéance. On débute ainsi l'exécution d'une tâche à une fréquence réduite par opposition à la première technique.

Dans les deux cas, des gains en énergie de 20 à 40% sont obtenus dans des exemples.

4•1•2•Politique hors-ligne.

Dans ce type de politique, les calculs d'ordonnancement sont exécutés avant l'exécution, et à fréquence individualisée pour chaque tâche.

Politique hors-ligne à vitesse unique et non préemptive.

Cette politique est également connue dans la littérature sous l'appellation MRS (*Maximum Required Speed* - MRS). L'objectif est de déterminer une vitesse du processeur valable pendant toute l'exécution de l'application. Cette vitesse doit être calculée au plus juste pour garantir le respect des échéances, ce qui présuppose une connaissance fine des besoins de l'application.

- **Politique hors-ligne à vitesse unique sous ordonnancement FPP (*Fixed Priority Preemptive*) .**

Dans le cas de tâches périodiques à échéances sur requêtes, on peut par exemple utiliser le test de faisabilité de [LL73] pour déterminer une vitesse minimale unique. Cette condition, basée sur la charge du processeur n'étant qu'une condition suffisante, et non une condition nécessaire, la vitesse trouvée pourrait encore être diminuée dans certains cas en respectant toutes les contraintes.

- **Politique hors-ligne à vitesse unique sous ordonnancement EDF.**

Brièvement, pour les tâches périodiques avec échéance égales aux périodes, l'adaptation du critère de faisabilité [GN03], $\sum_{\tau_i} \frac{w_i}{T_i} \leq 1$, avec w_i le temps d'exécution de la tâche i, et T_i, la période de la tâche i, permet d'obtenir aisément une vitesse minimale qui garantit un taux d'utilisation du processeur le plus proche de 100%.

- **Politique hors-ligne à vitesses multiples sous ordonnancement FPP.**

Citons les auteurs des travaux [QX02] qui proposent un algorithme optimal du point de vue énergétique dont le point de départ consiste à remarquer que certains ensembles de tâches peuvent être ordonnancés sous FPP à la vitesse optimale calculée pour EDF. Les auteurs présentent ensuite une technique qui, en modifiant certaines échéances, se ramène à de tels ensembles de tâches. L'inconvénient de cette approche est qu'elle est difficilement applicable en pratique.

- **Politique hors-ligne à vitesses multiples sous ordonnancement EDF.**

Les travaux relatés dans [GN03], appliqués à un ensemble de tâches non-récurrentes à échéances, ont pour point de départ la recherche de l'intervalle critique où la vitesse du processeur requise est maximale. La vitesse minimale admissible sur cet intervalle est assignée aux tâches qui appartiennent à cet intervalle. Une fois cet intervalle étudié, il est éliminé et on analyse le prochain intervalle critique. La vitesse du processeur est déterminée à chaque instant dans le but de respecter les échéances tout en minimisant l'énergie consommée.

4.3.Travaux relatifs à une architecture hétérogène.

Tous les travaux ci-dessus concernent l'ordonnancement monoprocesseur. Le cas multiprocesseur est complexe à traiter, toutefois quelques travaux existent. Citons ceux de [MH00]. L'algorithme s'appuie sur une heuristique gloutonne qui sélectionne itérativement la tâche qui peut être étendue au maximum afin de réduire la tension et la fréquence et ce, en tenant compte des communications et des dépendances entre les tâches de l'application soumise à des contraintes temporelles.

Cette heuristique est ensuite utilisée comme fonction d'évaluation dans un algorithme génétique qui optimise l'allocation des tâches sur les différentes unités de l'architecture. Cet algorithme permet une optimisation d'environ 20% sur les exemples testés. Cependant, les temps d'exécution de la méthode semblent importants sur des exemples relativement simples.

4.4.Contrôle de la puissance de fuite par une technique d'ordonnancement [LR03].

De nombreux travaux portent sur la gestion de puissance lors de la phase d'ordonnancement. Ainsi, dans ces travaux, le but cherché est de réduire le terme statique de la consommation, c'est-à-dire la puissance de fuite. En effet, en plaçant les unités en mode *sleep* et *idle*, la puissance de fuite est réduite. Les auteurs de [LR03] ont expérimenté cette vue avec les deux techniques d'ordonnancement : *Earliest-Deadline-First* et *Rate Monotonic* pour des systèmes temps réels durs. Ils calculent la possibilité de retarder une période d'utilisation de l'unité. Leurs travaux en perspective consistent à ajouter la gestion du DVS, donc de chercher à changer conjointement la tension et la fréquence.

4.5.Conclusions.

Cette analyse des travaux sur le partitionnement et l'ordonnancement basse consommation illustre la diversité des approches proposées qui résultent de nombreuses possibilités d'intervention sur la consommation mais aussi de nombreuses variantes d'architectures et les différents modèles d'applications considérées. En particulier, du fait de l'absence d'une approche générale d'optimisation de la consommation au niveau système, on est amené à considérer un type d'application et son modèle associé à partir duquel on cherche à optimiser au mieux la consommation.

Dans notre étude, nous considérons des applications de télécommunication orientées traitement du signal pour lesquelles une modélisation par graphes de flots de données est bien adaptée. Dans l'équipe, une méthode et un outil d'exploration d'architectures de SOC (CODEF) basés sur ce modèle a été développé [BL99].

L'objectif de notre travail est d'adapter les modèles et les algorithmes développés dans CODEF pour explorer des architectures de SOC basse consommation. Par

conséquent, nous présentons dans la suite l'environnement CODEF avant de détailler son extension à la basse consommation.

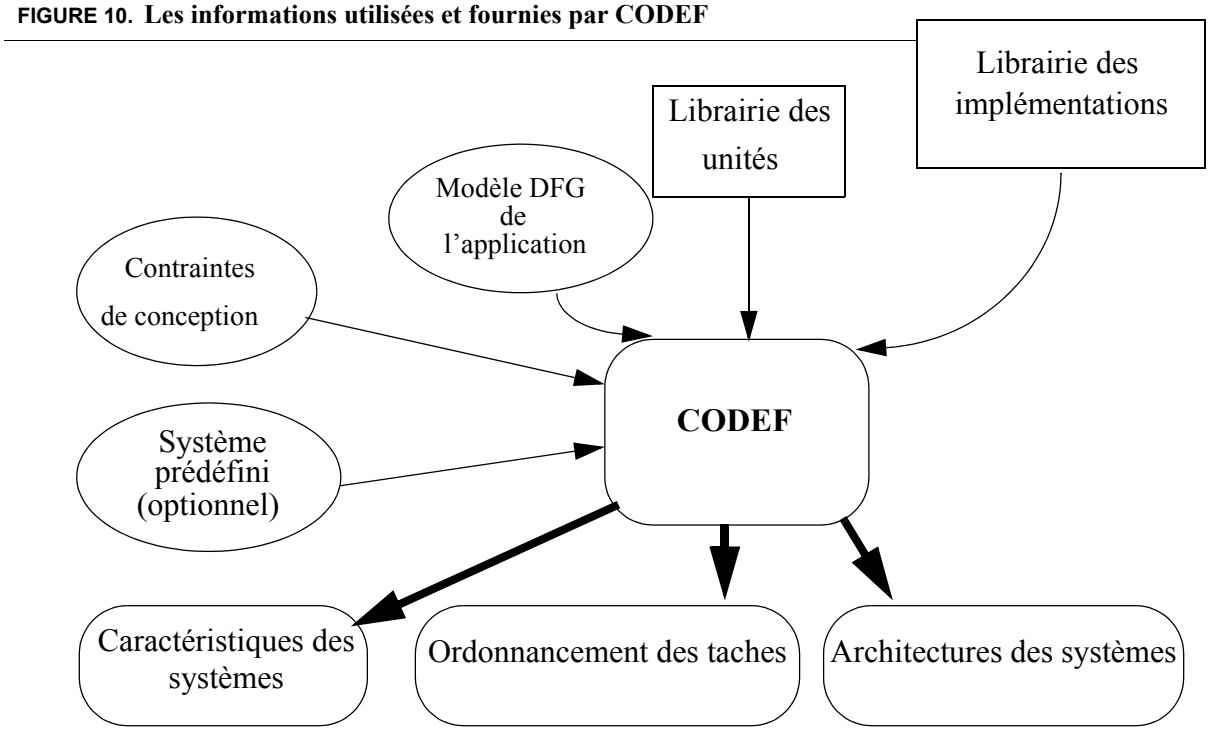
5•Description de l'environnement CODEF

5.1.Présentation générale de CODEF, *CODEsign of Framework* [BA98, CA00, CF00].

- **Objectif:**

Cet environnement a été étudié et développé dans le cadre d'un projet avec Phillips Semiconductors. L'objectif est de rechercher des architectures de SOC optimisées en surface de silicium capable d'exécuter en temps réel des applications en télécommunication et multimédia. La nature de ces applications a conduit à retenir un modèle de type graphe de flots de données (DFG, *Data Flow Graph*) pour les représenter. Ce graphe décrit de manière fonctionnelle l'application à un niveau de granularité élevé (niveau fonction). A partir d'une telle description d'une application, de contraintes temporelles et de réalisation (surface de silicium), CODEF sélectionne dans des bibliothèques, les unités logicielles et matérielles nécessaires à l'exécution de l'application et produit des architectures systèmes spécialisées, optimisées sur lesquelles les fonctions de l'application sont allouées et ordonnancées (figure 10).

FIGURE 10. Les informations utilisées et fournies par CODEF



Les bibliothèques mentionnées dans la figure et décrites dans la suite, correspondent d'une part, à la librairie des unités où sont décrites les caractéristiques telles que la surface et la fréquence d'horloge, et, d'autre part, à la librairie d'implémentations qui contient les caractéristiques d'exécution de chaque tâche pour chaque unité. La méthode est basée sur un algorithme de partitionnement/allocation/ordonnancement qui détermine une répartition temporelle des tâches de l'application sur les unités sélectionnées depuis la librairie ou contenue dans une architecture prédéfinie.

Précisons maintenant la représentation graphique utilisée pour représenter l'application en entrée de CODEF.

Le modèle de graphe de type flots de données (DFG) est une représentation graphique très répandue pour spécifier les applications orientées traitement dans le domaine du partitionnement. Cette spécification décrit le comportement du système par rapport aux flots de données le traversant. Cette représentation graphique, illustrée par l'exemple de la figure 11 est constituée de nœuds et d'arcs.

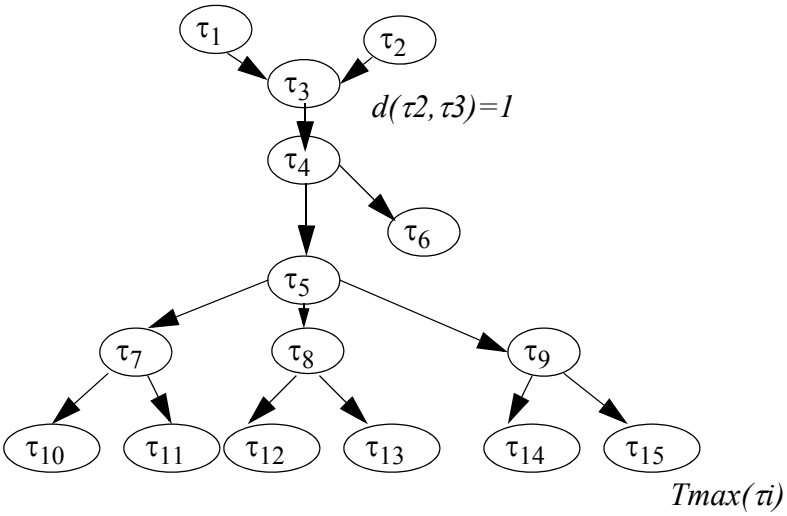


FIGURE 11. Exemple de DFG.

Les nœuds τ_i du graphe représentent les traitements ou tâches, et les arcs entre les nœuds représentent les relations de précédences entre les tâches. Chaque tâche consomme un jeton sur chaque arc entrant et après exécution produit un jeton sur chaque arc sortant. Sur les arcs, est noté le volume en octets d(τi,τj) des données échangées entre les tâches τ_i et τ_j. Sur les tâches terminales, sont spécifiées des contraintes temporelles maximum Tmax(τi) de fin d'exécution de ces tâches. Si l'exécution a lieu au-delà de cette date, il y a violation de la contrainte. Les valeurs de ces contraintes sont propagées vers les tâches prédécesseurs. Lorsqu'une tâche a plusieurs successeurs, la contrainte la plus sévère est sélectionnée. Ces contraintes vont permettre de guider les choix d'allocation et d'ordonnancement comme indiqué au paragraphe 5.3.

5.2.Bibliothèques de CODEF.

Comme indiqué sur la figure 10, CODEF utilise deux types de bibliothèques, la première décrivant les unités, et la deuxième décrivant les implémentations. Comme la classe d'applications principalement visée concerne les systèmes embarqués de traitement du signal pour les télécommunications, les architectures susceptibles d'être particulièrement candidates sont composées en général de plusieurs cœurs de processeur, souvent de différents types (RISC, DSP...), associés à des coprocesseurs ou à des accélérateurs matériels. Un coprocesseur n'est accessible que par le seul processeur programmable auquel il est directement connecté. Un accélérateur est une unité non programmable qui possède son propre contrôleur. Il peut être partagé, en exclusion mutuelle, entre plusieurs unités (processeurs et accélérateurs) et peut ainsi exécuter une tâche en parallèle avec les autres unités. La structure de la hiérarchie mémoire dépend du type de processeur. Une mémoire cache (un seul niveau) peut être placée dans la structure mémoire d'un processeur. CODEF étend avec de la mémoire externe l'espace mémoire des unités qui ne disposent pas de suffisamment de mémoire interne.

Détaillons les bibliothèques.

- *Bibliothèque des unités :*

CODEF nécessite une description des unités potentiellement utilisables. Cette description comporte les critères technologiques, comme la taille totale de l'unité en mm^2, les tailles des mémoires ROM et RAM si l'unité est programmable, le nom de la mémoire externe (externe à l'unité, cette déclaration de mémoire est utilisée dans le cas où CODEF alloue une tâche à l'unité alors que ses mémoires internes ne peuvent contenir entièrement le code et les données des tâches), la fréquence d'horloge, les ports d'entrées/sorties.

Nous produisons un exemple d'une bibliothèque composée d'un DSP et d'un accélérateur matériel (figure 12).

FIGURE 12. Extrait de la bibliothèque d'unités

```
processor: OAK
  {total area: 3.4 mm2; //.35
       ROM size: 5 Kbyte;
       RAM size: 12 Kbyte;
       memory: external_ram_Oak;

       clock cycle: 12.5 ns; // 80 Mhz typ.

        I/O port: portO
       { throughput: 2.0 byte/cycle;
              width: 16 bit;
              transfer mode: cpu;} }
       accelerator : HW_seuillage
          {total area : 0.007 mm2;
       memory : IO_port ;
       clock cycle : 10.0 ns;
       I/O port : port_data
         {        throughput : 2.0 byte/cycle;
              width : 16 bit;
              transfer mode : cpu;    }
```

Ici, la première unité décrite est un DSP, le processeur OAK. Il a une surface de 3,4 mm^2 et des tailles de ROM et de RAM de 5 et 12 kbytes respectivement. La mémoire externe qui lui est rattachée, est nommée external_ram_Oak. Cette unité a une période de 12,5 ns. Un port d'entrée/sorties est déclaré avec une capacité de 2bytes/cycle, pour des mots de largeur de 16 bits. Le mode de transfert se fait via l'unité centrale directement, noté dans le champ transfert-mode (l'autre possibilité est par *DMA, Direct Acces Memory*). De même, la deuxième unité est un accélérateur matériel avec le même type de champs décrits.

Nous verrons dans le chapitre suivant que nous sommes intervenus à ce niveau afin d'introduire les informations nécessaires pour prendre en compte la consommation. Voyons maintenant les caractéristiques des implémentations des tâches sur les unités disponibles.

- *Bibliothèque d'implémentations :*

Les tâches décrites dans le DFG de l'application doivent avoir chacune au moins une réalisation possible décrite dans la bibliothèque d'implémentations. Cette dernière comporte, pour une tâche donnée toutes les unités capables de l'exécuter, et pour chaque possibilité, les caractéristiques associées. Un exemple de description de ces informations est donné sur la figure 13.

FIGURE 13. Extrait de la bibliothèque d'implémentations

```
DecodMant_OAK: DecodMant {
   support: OAK;
   runtime: 18940 cycle;
   memory access ratio: 62 % ;
   scratch ram: 0 bytes;
   rom size: 274 bytes;
   ram size: 150 bytes; }

DecodMant_ARM: DecodMant {
   support: ARM7TDMI;
   runtime: 21310 cycle;
   runtime with cache: Arm_cache
            {          cache miss: 5%;          }
   memory access ratio: 55 % ;
   scratch ram: 0 bytes;
   rom size: 392 bytes;
   ram size: 150 bytes;  }
```

Nous avons décrit ici une tâche, nommée DecodMant, qui peut être exécutée par le processeur OAK ou le processeur ARM7TDMI. Ainsi, la convention qui a été adoptée dans CODEF est de donner en premier le nom de l'implémentation suivi du nom de la tâche à laquelle cette implémentation correspond.

La ligne *runtime* correspond au nombre de cycles nécessaires à l'exécution de cette tâche par l'unité. Le temps correspond au nombre de cycles réels de l'unité pour exécuter la tâche sans tenir compte d'éventuelles pénalités dues à des défauts de cache ou des accès à une mémoire externe. Dans le cas où l'unité possède une mémoire cache (dont les caractéristiques sont listées dans la librairie des unités), le taux de défaut mentionné est utilisé par CODEF pour tenir compte des cycles d'attente dans le temps d'exécution. De même, le taux d'accès mémoire par rapport au nombre de cycles d'exécution de la tâche, est utilisé par CODEF pour tenir compte des cycles d'attente qui interviennent si la tâche est placée dans la mémoire externe de l'unité. Ces cycles sont déduits du temps d'accès mémoire renseigné dans la bibliothèque des unités. Les tailles mémoire indiquées correspondent à l'occupation induite par le placement de la tâche sur l'unité (code et données). Certaines implémentations de tâches peuvent nécessiter l'utilisation d'un co-processeur spécifique. Dans ce cas, le nom d'un coprocesseur est déclaré dans la liste des caractéristiques de l'implémentation. Les caractéristiques du coprocesseur lui-même (surface, port d'entrée/sortie sur lequel il est connecté au processeur) sont déclarées dans les bibliothèques des unités.

De même que pour la bibliothèque d'unités, nous avons effectué ici quelques modifications liées à la consommation due aux exécutions des tâches. Nous verrons en détail ces modifications dans le chapitre 4.

Etudions maintenant le fonctionnement de CODEF.

5.3. CODEF étape par étape.

A partir de la description de l'application, des bibliothèques et des contraintes de conception, CODEF construit un ensemble de solutions possibles (si au moins une solution est trouvée) qui vérifient les contraintes temporelles de l'application et les contraintes de conception. Bien entendu, d'autres solutions sont possibles. Pour chaque solution, CODEF fournit sa description architecturale, un ordonnancement des tâches de l'application sur cette architecture et des caractéristiques globales comme, par exemple les taux d'utilisation de chaque unité. Le concepteur peut alors réinjecter dans CODEF une des solutions trouvées pour l'affiner ou réaliser des optimisations supplémentaires. On peut par exemple jouer sur différents facteurs comme les tailles de mémoire ou les caractéristiques d'une mémoire cache et regarder l'impact de ces modifications sur les performances d'exécution de l'application.

- Méthode de partitionnement :

Elle est basée sur un algorithme d'ordonnancement/allocation par listes [BA98]. Elle opère en deux phases.

Dans la première phase, on calcule les bornes minimum ($pl_{min}(\tau_i,u_k)$) et maximum ($pl_{max}(\tau_i,u_k)$) des longueurs temporelles des chemins issus de chaque tâche vers les tâches terminales. Ces calculs sont réalisés récursivement à partir des tâches terminales, et prennent en compte les temps d'exécution $t_{exe}(\tau_i,u_k)$ des différentes réalisations u_k (logicielles ou matérielles) possibles de chaque tâche et les éventuelles communications inter-tâches $tc_{uk,ul}(\tau_i,\tau_j)$, si u_k est différent de u_l. Plus l'écart entre $pl_{min}(\tau_i,u_k)$ et $pl_{max}(\tau_i,u_k)$ est réduit, moins le choix du placement des successeurs n'aura d'influence sur les performances temporelles, si τ_i est placée sur u_k.

Dans la deuxième phase, on effectue la sélection des unités, l'allocation et l'ordonnancement des tâches en analysant le DFG des racines vers les feuilles.

Pour chaque tâche ordonnançable, on calcule un temps disponible $T_d(\tau_i,u_k)$ qui correspond à la différence entre la contrainte temporelle $T_{max}(\tau_i)$ de la tâche τ_i et le maximum des dates de fin d'exécution des tâches prédecesseurs de τ_i augmenté du temps de communication éventuel. En comparant la valeur de la longueur temporelle minimum du chemin issu d'une tâche τ_i avec la valeur du temps disponible de cette tâche, on obtient une mesure de l'urgence $\gamma(\tau_i,u_k)$ à ordonnancer τ_i sur u_k.

En effet, plus cette différence est faible, moins on dispose de degré de liberté dans l'ordonnancement de la tâche et de ses successeurs. Cette mesure d'urgence

temporelle permet à l'algorithme d'ordonnancement de construire des listes dynamiques pour chaque couple $<\tau_i,u_k>$.

- Utilisation de listes dynamiques :

L'intérêt des listes construites de façon dynamique est de permettre d'ajuster les choix en fonction des contraintes de temps et de surface.

A chaque étape, il y a ordonnancement d'une tâche ordonnançable, et vérification qu'il existe une unité pour laquelle le chemin minimum issu de la tâche vers la tâche terminale la plus contraignante est inférieur à T_d. Pour effectuer ce choix, CODEF construit des listes d'implémentations de couples $<\tau_i,u_k>$ pour l'ensemble des tâches ordonnançables.

Définissons ces listes:

Liste	Caractéristiques des implémentations $<\tau_i,u_k>$	Condition à vérifier
L^1_{reuse}	tâche déjà allouée sur cette unité présente dans l'architecture	$pl_{max}(\tau_i,u_k) \le T_d$
L^2_{reuse}	tâche déjà allouée sur cette unité présente dans l'architecture	$\gamma \ge T_{reuse}$
L^1_{new}	unité déjà présente, mais tâche non assignée à cette unité	$\gamma \ge T_{new}$
L^2_{new}	unité non présente et capable de réaliser la tâche	$\gamma \ge T_{new}$
L_{other}	autres cas	$pl_{min}(\tau_i,u_k) < T_d$

- T_{reuse} : paramètre compris entre 0 et 1 dont la valeur initiale est fixée par l'utilisateur. Ce paramètre permet de fixer un seuil sur la performance minimum que doit offrir une unité déjà présente dans l'architecture pour exécuter une tâche.
- T_{new} : paramètre de même type que T_{reuse} mais dédié aux unités à instancier dans l'architecture depuis la bibliothèque.

Les listes L^1_{new}, L^2_{new} et éventuellement L_{other} impliquent l'ajout d'une nouvelle unité dans l'architecture avec pour conséquence un accroissement de la surface de silicium. Afin d'optimiser la surface de silicium, on introduit le paramètre $\Omega(\tau_i,u_k)$ qui est une combinaison de l'urgence temporelle et d'une surface pondérée de u_k. La surface est pondérée par la possibilité de réutiliser l'unité pour les tâches restant à ordonnancer dans le temps disponible. Cette surface pondérée tient compte de la surface propre de l'unité mais aussi des extensions éventuelles de mémoire pour contenir le code et les données des tâches.

On a donc des listes dont la signification est :

- L^1_{reuse} : liste des couples $<\tau_i,u_k>$ où u_k représente une unité déjà instanciée et τ_i une tâche déjà réalisée par u_k. Ces couples vérifient que $pl_{max}(\tau_i,u_k) \leq T_d$ et sont, de ce fait, les moins contraignants pour la suite du partitionnement. La réutilisation des unités u_k de cette liste n'entraîne pas d'augmentation de surface. Les couples sont triés par ordre décroissant de la différence entre $T_d(\tau_i,u_k)$ et $pl_{max}(\tau_i,u_k)$.
- L^2_{reuse} : liste des couples $<\tau_i,u_k>$, où u_k correspond à une unité déjà présente dans l'architecture et τ_i une tâche déjà réalisée par u_k. Ces couples ont une valeur de $\gamma(\tau_i,u_k)$ supérieure à la valeur du seuil de réutilisation T_{reuse} (l'unité est jugée suffisamment rapide par rapport à l'urgence temporelle). Ces choix sont considérés comme non critiques. Comme les unités présentes dans cette liste existent déjà dans l'architecture, leur réutilisation n'entraîne pas d'augmentation de surface. Une zone mémoire est déjà réservée, pour le code et les données de la tâche sur l'unité. Les couples sont triés par ordre décroissants des valeurs $\gamma(\tau_i,u_k)$.
- L^1_{new} : liste des couples $<\tau_i,u_k>$, tels que l'unité u_k est déjà présente dans l'architecture, mais τ_i est une tâche non assignée à cette unité. De plus, ces couples vérifient que la valeur de $\gamma(\tau_i,u_k)$ est supérieure à la valeur du seuil d'instantiation T_{new} (l'unité est jugée suffisamment rapide par rapport à l'urgence temporelle). Ces couples sont donc considérés comme non critiques. Cependant, choisir un couple dans cette liste peut entraîner une augmentation de surface, liée à l'augmentation de la mémoire de l'unité. Il faut en effet, dans ce cas, vérifier que l'espace mémoire disponible pour l'unité (interne et externe), peut contenir le code et les données persistantes de la tâche.
- L^2_{new} : liste des couples $<\tau_i,u_k>$, tels que l'unité u_k, non présente dans l'architecture, est capable de réaliser la tâche ordonnançable τ_i. De plus, la valeur de $\gamma(\tau_i,u_k)$ est supérieure à la valeur du seuil d'instanciation Tnew. Mais choisir une unité dans cette liste entraîne une augmentation de surface. Cette augmentation de surface est due à la surface de l'unité elle-même et à la quantité de mémoire nécessaire pour l'implémentation de la tâche sur l'unité.
- L_{others} : cette liste correspond aux couples $<\tau_i,u_k>$, considérés comme critiques et donc, devant être choisis en dernier recours. Ils correspondent aux couples qui n'ont pas pu se placer dans les listes précédentes. Cette liste contient les plus mauvais choix d'unités u_k réalisant les tâches τ_i et offrant le moins de garanties sur le respect des contraintes temporelles.

Une fois ces listes constituées avec l'ensemble de tâches ordonnançables, l'algorithme détermine parmi l'ensemble de ces tâches, celle qui est la plus critique en temps, c'est-à-dire $T_d(pl_{min}(\tau_i,u_k))$ est minimum $\forall\tau_i$ et $\forall u_k$. Pour cette tâche, l'algorithme parcourt les listes dans l'ordre L^1_{reuse}, L^2_{reuse}, L^1_{new}, L^2_{new}, L_{others} et dans chaque liste dans l'ordre où elles ont été triées. La première unité rencontrée capable de réaliser la tâche la plus critique en temps est sélectionnée.

Si pour une tâche ordonnançable τ_i aucun couple $<\tau_i,u_k>$ n'a été placé dans la liste, le partitionnement échoue car il existe au moins une contrainte temporelle qui n'est jamais satisfaite quelque soit l'unité considérée.

Les seuils d'urgence (seuils de réutilisation T_{reuse} et d'instanciation T_{new}) ont des valeurs comprises entre 0 et 1. Ils permettent d'orienter l'algorithme de la manière suivante :

- Lorsque T_{reuse} est proche de 0, l'algorithme ne différencie pas, d'un point de vue temporel, les couples $<\tau_i,u_k>$. Ainsi, on favorise a priori la réutilisation et la minimisation de la surface de silicium par utilisation du paramètre $\Omega(\tau_i,u_k)$. Pour des valeurs de T_{reuse} proches de 1, l'algorithme choisit les unités, parmi celles existantes dans l'architecture, les unités les plus rapides. L'algorithme a donc la possibilité de favoriser ou de restreindre la réutilisation des ressources et d'orienter les choix en fonction de la rapidité des unités.

- Lorsque T_{new} est proche de 1, les unités les plus rapides, non présentes dans l'architecture, vont être choisies, en priorité, pour enrichir l'architecture. Ceci peut conduire à une augmentation de la surface de l'architecture finale, mais peut assurer le respect des contraintes temporelles. Proche de 0, ce seuil peut entraîner le choix d'unités ayant un coût moindre en surface et donc généralement plus lentes. Ceci peut conduire à l'échec de l'algorithme à cause de la violation des contraintes temporelles.

L'algorithme effectue automatiquement une exploration de l'espace de conception en faisant varier les paramètres T_{new} et T_{reuse}. A partir de ces deux valeurs définies pour ces paramètres, par exemple les valeurs initiales, les incréments sur ces paramètres sont évalués dynamiquement en calculant la valeur minimum d'incrément qui conduit à déplacer de liste au moins un couple $<\tau_i,u_k>$ sélectionné pendant la partition du DFG. Tant que les seuils sont inférieurs à 1, l'algorithme de partitionnement est réappliqué en ajoutant aux seuils les valeurs d'incréments obtenus. On obtient ainsi un ensemble de solutions (si au moins une solution existe) qui vérifient les contraintes temporelles et pour lesquelles l'algorithme cherche à minimiser la surface de silicium.

- Résumé :

Dans une première phase, un calcul de la longueur de chemins minimum et maximum est effectué.

Dans une deuxième phase, l'algorithme construit les listes avec les tâches ordonnançables du DFG. A chaque itération, l'algorithme de partitionnement opère en deux étapes :

- construction des listes avec l'ensemble des couples $<\tau_i,u_k>$ correspondant aux tâches ordonnançables.
- sélection de la "meilleure" implémentation pour la tâche la plus critique en temps.

Cette deuxième phase est répétée en faisant varier les seuils de réutilisation et d'instantiation afin d'explorer plusieurs solutions dans l'espace de conception.

6•Conclusion

Comme nous l'avons vu, l'ordonnancement est un point clé dans le partitionnement, il apparaît donc important de définir des techniques d'ordonnancement efficaces en gestion de la consommation.

De plus, le choix des unités est aussi un point crucial dans le processus d'optimisation, en particulier, si ces unités offrent différents modes de fonctionnement, et la possibilité d'ajuster en tension et en fréquence.

Notre objectif étant de concevoir des systèmes autonomes sur puce basse consommation pour les télécommunications, nous avons considéré l'outil d'aide à la conception, CODEF, développé au sein de l'équipe. Afin de privilégier le critère de consommation dès les premières phases de la conception de ces systèmes autonomes, nous avons choisi d'implémenter diverses stratégies d'économie d'énergie dans cet outil. Cette étape permettra de les tester et de retenir celles qui permettent les meilleures optimisations. A ces fins, après avoir décrit les travaux précédents dans le domaine de la conception conjointe logicielle/matérielle avec prise en compte ou non de la consommation, nous avons décrit en détail le fonctionnement de l'outil CODEF.

Cependant, pour définir des techniques de conception basse consommation prenant en compte ces aspects, il faut préalablement se doter de méthodes et d'outils pour évaluer la consommation de ces unités. Dans le chapitre suivant, nous présentons les travaux réalisés sur cet aspect, l'estimation de la consommation.

Chapitre IV : Estimation de la consommation d'un système autonome

Introduction.

L'objectif de ce chapitre est d'introduire le travail amont qui a été mis en oeuvre pour prendre en compte la consommation lors de la conception globale de systèmes autonomes basse consommation. Ainsi, nous décrivons ici comment ont été renseignées les bibliothèques de CODEF de façon à estimer et optimiser l'énergie d'un système.

Les travaux réalisés ciblent principalement l'estimation de la consommation de la partie logicielle (les processeurs) mais nous indiquons aussi les thématiques utilisées pour l'estimation de la consommation de la partie matérielle (les accélérateurs matériels).

Les applications de traitement du signal pour les télécommunication sont souvent décrits en langage C afin de les implémenter sur des processeurs DSP. Estimer manuellement la consommation d'un code C exécuté par la partie logicielle de l'architecture, en se ramenant au niveau assembleur peut être très gourmand en temps. Nous avons donc réalisé une automatisation de cette estimation. En effet, l'équipe, en collaboration avec Philips Semiconductors, a développé un estimateur de performances pour DSP (VESTIM). La motivation principale de ce travail émane de l'inefficacité des compilateurs traditionnels en raison de l'hétérogénéité des architectures, par exemple liée à la diversité des modes d'adressages et à la spécialisation des registres dans le chemin de données du processeur [PA99].

Le travail a donc porté sur l'étude d'un modèle de consommation de DSP qui a été introduit ensuite dans VESTIM afin de caractériser à la

fois en performance et en consommation, les traitements (fonctions de l'application écrites en C). Il est à noter que cet estimateur accepte toute construction autorisée par le langage C.

Naturellement, des processeurs à usage plus général sont souvent utilisés dans les architectures systèmes. Nous avons donc cherché des modèles de consommation pour ce type de processeur. Des travaux ont été effectués sur ce problème et des outil libres sont disponibles, comme nous le signalons dans la suite.

Une fois connus tous les comportements en performance et en consommation des unités d'un système, il s'agit ensuite de caractériser en consommation une architecture complète dédiée à une application. Nous détaillons dans la deuxième partie de ce chapitre l'approche développée pour estimer la consommation au niveau système d'une application ordonnancée sur une architecture. Cette estimation a été intégrée dans CODEF.

1•Estimation de la consommation de la partie logicielle

Cette partie est consacrée à l'estimation de la consommation pour la partie logicielle. Nous nous intéressons en particulier au cas des DSP, et, en ce qui concerne les processeurs à usage général (RISC) nous rappelons des résultats applicables à ce problème d'estimation.

1.1.Les processeurs à usage général, type RISC.

Afin de disposer d'une unité processeur à usage général, nous avons exploité l'estimateur de performance/consommation disponible en logiciel libre, Jouletrack [Jo]. Cet estimateur cible l'architecture du StrongARM qui est composé d'un cache de 16 kbyte d'instructions, et 8 kbyte de données, d'unités fonctionnelles classiques, plus une pour gérer la puissance. La fréquence d'horloge de ce processeur peut varier de 59 à 206 MHz et la tension d'alimentation de 0.8 V à 1.5 V.

Toutefois, cet outil ne supporte que des applications de petites capacités, puisque le fichier d'entrée contenant le code C à estimer est limité à 50 koctets. De même, il n'est pas envisageable de mettre plusieurs fichiers en entrée. En découpant une application de vidéo embarquée (présentée dans la partie des résultats), nous avons pu obtenir des renseignements suffisants pour enrichir les bibliothèques de CODEF.

Si pour un processeur RISC, il est possible de procéder à une simulation du pipeline à partir d'un code compilé pour estimer la consommation, il n'en est pas de même pour les DSP qui ont une architecture plus hétérogène.

1.2.Processeurs de traitement du signal (DSP).

Dans cette partie, nous nous intéressons à la consommation des processeurs de traitement du signal et des processeurs à usage plus général. De manière générale, les compilateurs pour DSP sont peu efficaces, aussi les programmes en assembleur sont obtenus manuellement pour obtenir des codes efficaces.

Les DSP ont été étudiés pour exécuter efficacement des fonctions de traitement du signal qui opèrent en général sur des flots réguliers de données. Ils exploitent un parallélisme de données important et intègrent des opérateurs spécifiques pour accélérer les traitements qui chaînent des opérations suivant un motif répétitif. Les modes d'adressage et l'hétérogénéité qui découlent de leur architecture rendent délicat la génération et l'optimisation de code dans les compilateurs. Cependant, il est difficile de connaître a priori la qualité d'un code complet d'une application si on ne connaît pas la performance que devrait fournir le processeur pour cette application.

Pour pallier ce problème, l'équipe, en collaboration avec Philips Semiconductors Sophia, a développé un logiciel, VESTIM [PA99] qui permet d'obtenir rapidement, et de manière automatique une estimation du temps d'exécution d'un code C par

un DSP. L'objectif est ici de faire évoluer cet outil en un estimateur de performances et de consommation.

1.3.Présentation de VESTIM [PA99].

Ce logiciel estime les performances d'un code généré à partir du code C d'une application. Il correspond à une estimation d'un code assembleur optimisé. En particulier, l'accent est mis sur la recherche d'une mesure de qualité du code, et non pas sur une recherche d'un temps d'exécution dans le pire cas.

L'estimation des performances d'un code nécessite de connaître deux types d'information :

- le nombre de fois qu'est exécutée chaque instruction (x_i)
- le nombre de cycles nécessaire à l'exécution de chaque instruction (c_i)

La figure 14 présente les entrées et sorties de VESTIM.

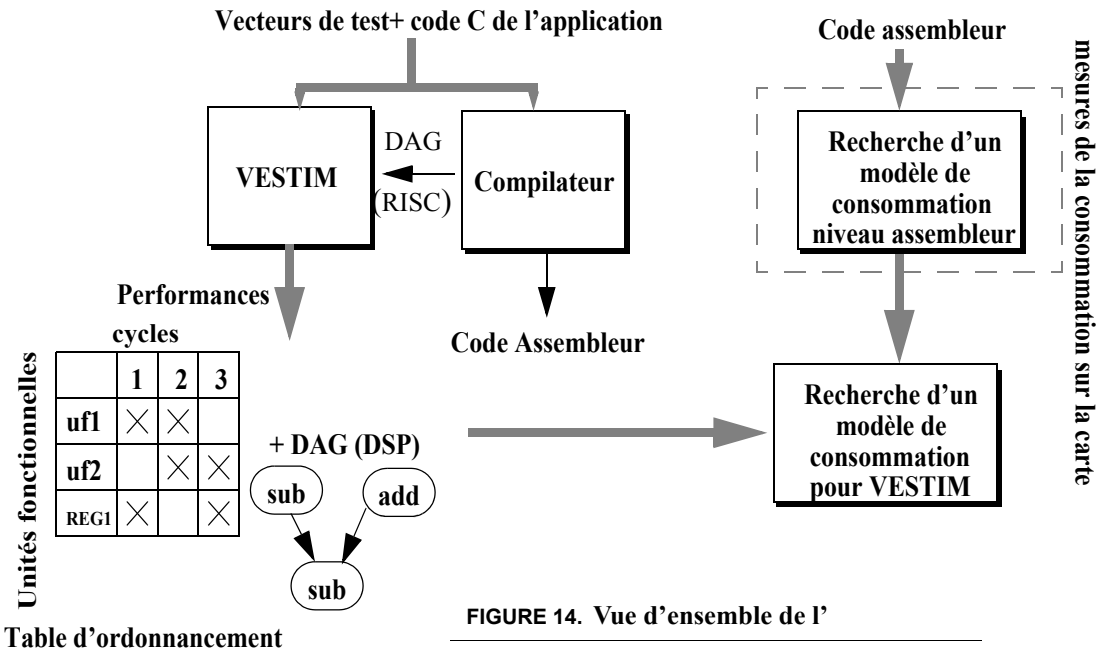


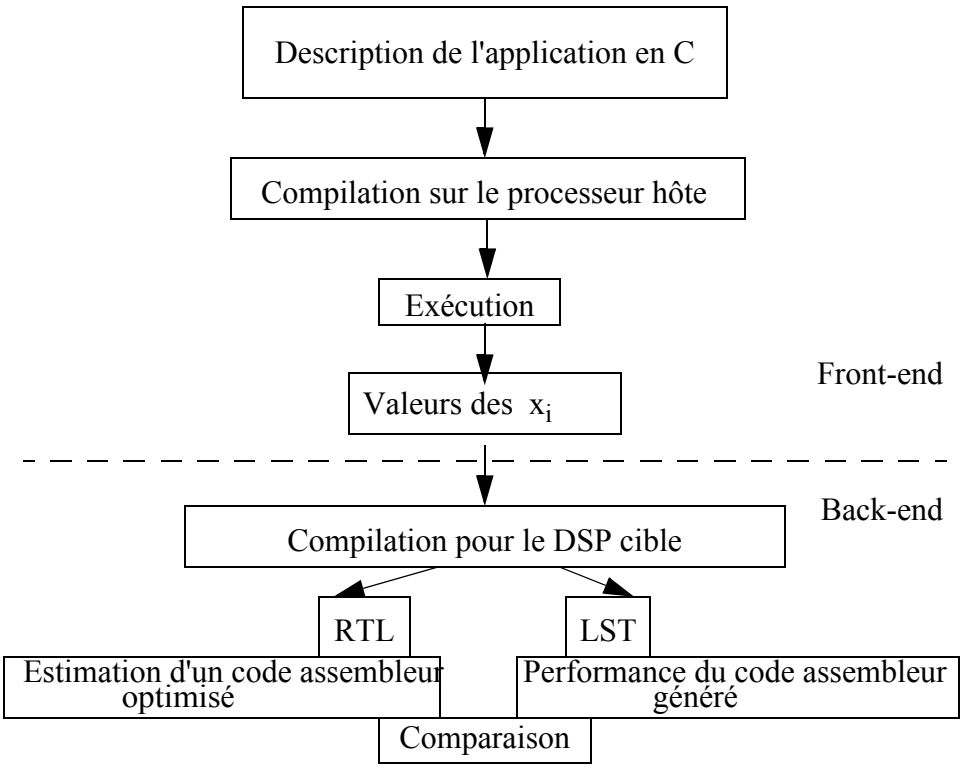
FIGURE 14. Vue d'ensemble de l'

A partir d'un code C de l'application, VESTIM fournit un *DAG* (Direct Acyclic Graph) adapté à une architecture DSP et une table d'ordonnancement (partie gauche de la figure 14). Cette table d'ordonnancement permet de déterminer, à chaque cycle les opérations exécutées et donc, les unités activées. Grâce à ces éléments, VESTIM estime les performances équivalentes à un programme assembleur optimisé. Notre objectif est alors d'estimer dans le même temps la consommation (partie droite de la figure14). A cette fin, nous avons dans un premier temps élaboré des modèles de consommation par instruction assembleur (détaillés en annexe). Toutefois, ces modèles ne sont pas directement exploitables par VESTIM, qui opère à par-

tir de la table d'ordonnancement construite. La consommation est alors déduite en calculant la contribution de chaque unité de l'architecture impliquée dans l'exécution des instructions du programme. Regardons plus en détails VESTIM dans la suite.

On se place ici dans le cas d'une architecture à comportement fortement déterministe comme c'est le cas de la majorité des DSP. Le nombre de cycles pour exécuter l'instruction (c_i) implique une estimation précise qui doit tenir compte des spécificités de l'architecture. La figure 15 présente une vue générale de VESTIM basé sur ces principes.

FIGURE 15. Vue générale de VESTIM



Le *front-end* décrit l'ensemble des phases dépendantes du langage source mais indépendantes du processeur DSP cible. Ici, on exécute le programme sur le processeur hôte avec une séquence de test afin d'obtenir une évaluation des valeurs des x_i. Le choix de la séquence de test est important pour obtenir une estimation réaliste.

Le *back-end* décrit l'ensemble des phases dépendantes du processeur cible mais indépendantes du langage source. Ici, on calcule les performances du code généré et on estime celles d'un code assembleur optimisé.

Définissons quelques termes présents dans la figure 15 ou utilisés par la suite.

RTL : *Register Transfer Language.* (niveau transfert registre) C'est le code intermédiaire du compilateur GNU utilisé par VESTIM. C'est à partir de ce code modifié qu'est effectué l'estimation d'un code assembleur optimisé.

LST : C'est le code assembleur généré par le compilateur GNU C du DSP. VESTIM calcule le nombre de cycles c_i pour chaque instruction par simple sommation des nombres de cycles des instructions assembleur associées.

Bloc de base : Segment de code dont le seul point d'entrée est la première instruction et le seul point de sortie est la dernière instruction.

Détaillons maintenant chaque étape de VESTIM.

1.3.a.Vestim étape par étape.

Le programme est décomposé en blocs de base. Le nombre d'exécutions de chaque bloc de base x_i est obtenu en exécutant le code C sur un processeur hôte.

Il est ensuite calculé le coût c_i en nombre de cycles machine de chaque bloc de base du code assembleur produit par le compilateur. Ce coût est calculé par sommation des coûts de chaque instruction dans le bloc de base. Cette méthode est valable ici car la classe des DSP ne possède pas de cache de données, sinon il faudrait modéliser leur comportement pour affiner le calcul [TM96]. Le temps d'exécution total du code C complet compilé pour le DSP est $T_{exe}= \sum_{i=0}^{N} x_i \times c_i$ composé de N blocs de base.

Pour l'estimation du code assembleur optimisé, on opère à partir d'une description du programme dans un langage intermédiaire (RTL). Ce langage intermédiaire est obtenu à partir de GNU-CC. Dans ce compilateur, différentes passes sont utilisées pour raffiner de proche en proche la description du programme jusqu'à arriver au code cible. Dans l'outil VESTIM, l'allocation des registres n'a pas été faite car cela évite de considérer les nombreuses sauvegardes temporaires de registres (*spilling*) qui sont générées ensuite dans le code pour le DSP. De façon générale, un code optimisé pour DSP contient très peu de *spilling*. Ce code intermédiaire est ensuite modifié pour tenir compte des spécificités des DSP et en particulier les modes d'adressage qui n'utilisent en général pas les registres de calcul mais des registres dans les unités d'adressage. Sur ce code intermédiaire orienté DSP (DIR : DSP *Intermediate Register*), il est effectué une pseudo-génération de code par ordonnancement dans le but de construire une table d'ordonnancement par bloc de base (DAG).

Cette pseudo-génération de code tient compte de la description de l'architecture du processeur.

- Ordonnancement des opérations :

On cherche à effectuer un ordonnancement au niveau opération de la DIR afin de construire des pseudo instructions du DSP: les colonnes de la table de réservation.

- La modélisation du processeur :

Le processeur est modélisé suivant ses unités fonctionnelles et en particulier celles de son chemin de données. En particulier, le chemin de données fait apparaître l'utilisation parallèle possible d'unités fonctionnelles qu'il s'agit d'exploiter dans la pseudo génération de code. Sa représentation est simplifiée par deux critères. Le premier concerne les classes de mémoires et de registres. Toutes les opérations qui conduisent à l'utilisation commune de ressources impliquent que ces dernières appartiennent à la même classe. Le deuxième consiste à s'abstraire de toutes les caractéristiques architecturales du processeur qui ont des effets limités sur les performances effectives (le principe du pipeline est de faire apparaître une exécution d'une instruction par cycle, par conséquent à toute instruction on associe un cycle d'exécution par défaut). Après ordonnancement, on obtient pour chaque bloc de base une estimation des valeurs des c_i ce qui permet d'évaluer le temps d'exécution globale de l'application.

Cette méthode a été développée dans l'outil VESTIM et a été validée vis à vis des deux DSP : le OAK et le Palm. Par conséquent, notre objectif a été d'étendre la méthode et l'outil pour obtenir une estimation de la consommation, en s'appuyant sur ces deux DSP.

Comme les processeurs sont modélisés par leurs unités fonctionnelles et leurs chemins de données dans VESTIM, nous avons orienté nos modèles de consommation afin qu'ils soient en concordance avec ce niveau de description de l'architecture.

Nous présentons en annexe les architectures de ces DSP, la méthode utilisée pour les caractériser en consommation et enfin, nous présentons brièvement les modèles de consommation obtenus pour le OAK et le PALM.

1.3.b.Implémentations de la consommation dans VESTIM.

L'outil VESTIM effectue une allocation et un ordonnancement des opérations décrites dans la représentation interne de chaque bloc de base et construit une table d'ordonnancement donnant cycle par cycle l'occupation des unités fonctionnelles du DSP. Cette table d'ordonnancement permet de déduire l'activation des unités fonctionnelles nécessaires à l'exécution des opérations décrites dans chaque bloc de base. De plus, elle exprime naturellement le parallélisme entre les unités fonctionnelles.

Par conséquent, l'estimation de la consommation à un cycle donné peut être obtenue par sommation des contributions de chaque unité activée. Pour toutes les unités fonctionnelles du processeur, nous associons une valeur moyenne de consommation à chaque unité fonctionnelle définie dans la description d'architecture du DSP. Cette valeur est déduite des modèles de consommation (décrits dans l'annexe).

Prenons l'exemple du *DAG, Data Acyclic Graph* de la figure 16:

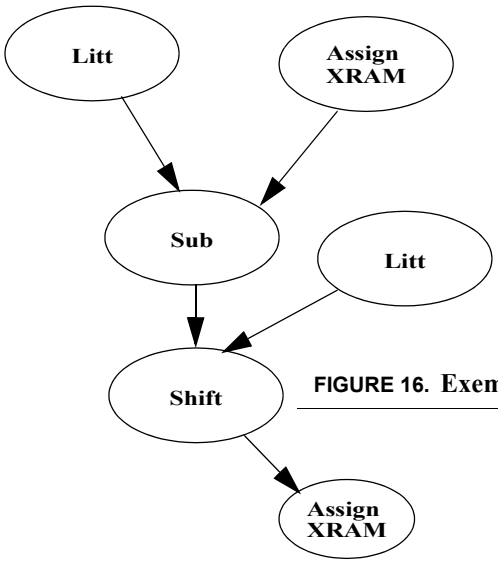


FIGURE 16. Exemple de DAG associé à un bloc de base

VESTIM fournit la table d'ordonnancement décrite dans le tableau 4 correspondant à ce DAG:

Tableau 4: Table d'ordonnancement associée au DAG

u_i	cycle 1	cycle 2	cycle 3	cycle 4
ALU		Sub		
Barrel Shifter			Shift	
MEM		Ltd		X
CG (constant Generator)	X		X	
ACCU	X	X	X	
RI		X		
SV			X	X

Ces informations permettent de connaître à chaque cycle les unités activées et les types d'adressage utilisés. Par exemple, durant le cycle 2, l'unité *ALU* est activée pour exécuter un *sub*, la mémoire et les unités de registres *RI* et *ACCU* sont aussi activées : chacun participera à la consommation. A chaque cycle, on somme les différentes contributions et entre chaque cycle, lorsque des unités différentes sont activées, il est nécessaire d'ajouter un surcoût, appelé *overhead,* noté ici $E_{ov(i,i+1)}$.

Ainsi, dans notre exemple du tableau 4, le bilan en consommation pour le cycle 2 est: $E_{cycle\ 2} = E_{sub} + E_{mémoire} + E_{accu} + E_{RI}$.

Sur l'ensemble des cycles, le bilan énergétique E_{bb} du bloc de base sera donc:

$$E_{bb} = E_{cycle\ 1} + E_{ov(1,2)} + E_{cycle\ 2} + E_{ov(2,3)} + E_{cycle\ 3} + E_{ov(3,4)} + E_{cycle\ 4}$$

On obtient l'énergie associée à l'exécution du programme complet, par sommation des énergies de chaque bloc de base multipliée par le nombre de fois où celui-ci est exécuté dans le programme (x_j) : $E = \sum_j E_{bb\ j} \times x_j$.

Nous devons donc prendre en compte les coûts dus à l'activation des unités u_n E_{i,u_n} à chaque cycle i, et ceux dus aux changements de chemins de données aux cycles i et i+1, lorsque deux unités sont activées successivement. L'énergie se déduit de la puissance évaluée sur la durée du cycle. Formulons ce principe :

Pour chaque cycle i, on a : $E_{cycle\ i} = \sum_{unités\ n} E_{i,u_n} = \left(\sum_n P_{i,u_n}\right) \times \frac{1}{f}$

L'énergie d'overhead (due aux changements de chemins de données lorsque deux unités sont activées successivement) est :

$E_{ov(i,i+1)} = \beta \times E_{moyen}$, avec β=1 si il y a un changement d'unité activée entre les cycles i et i+1. On a considéré ici un coût moyen pour ce type d'overhead, car sur des tests, tel qu'un FIR ou une FFT, l'estimation présentait une erreur de 5% dans ces conditions, ce qui est acceptable.

Finalement, le coût du bloc de base j est : $E_{bb\ j} = \sum_i E_{cycle\ i} + \sum_i E_{ov(i,\,i+1)}$.

Du fait de la décomposition du programme effectuée par VESTIM en fonctions et en blocs de base, on obtient une analyse énergétique du programme suivant ces entités facilitant ainsi une optimisation en énergie du code.

1.3.c.Résultats de VESTIM.

Avec VESTIM nous avons estimé les performances et la consommation d'un code C d'une FFT sur un DSP de type OAK. Nous avons ensuite mesuré la consommation de l'exécution par le OAK d'un code assembleur optimisé. Les erreurs sur l'estimation de consommation et de performance par rapport aux mesures effectuées sont d'environ 5%, ce qui est acceptable.

Regardons maintenant des résultats sur une application plus importante.

- **Description de l'application cible**

Dans le cadre du projet RNTL EPICURE[1] une application test proposée par le CEA a été étudiée. Il s'agit d'un algorithme de détection de mouvement dans une caméra intelligente. Les domaines d'utilisation possibles sont tant la vidéo-surveillance pour le contrôle qualité dans l'industrie, que la sécurité des personnes et des biens, par exemple.

Cette application permet de réaliser une détection de mouvement avec un fond d'image fixe. La détection des mouvements considère une image référence, appelée image de fond, composée des objets fixes. Par comparaison de cette image de fond avec les nouvelles images seuillées puis filtrées, il est possible de détecter les objets en mouvement.

Voyons un peu plus schématiquement quelles sont les principales étapes de cet algorithme (figure 17).

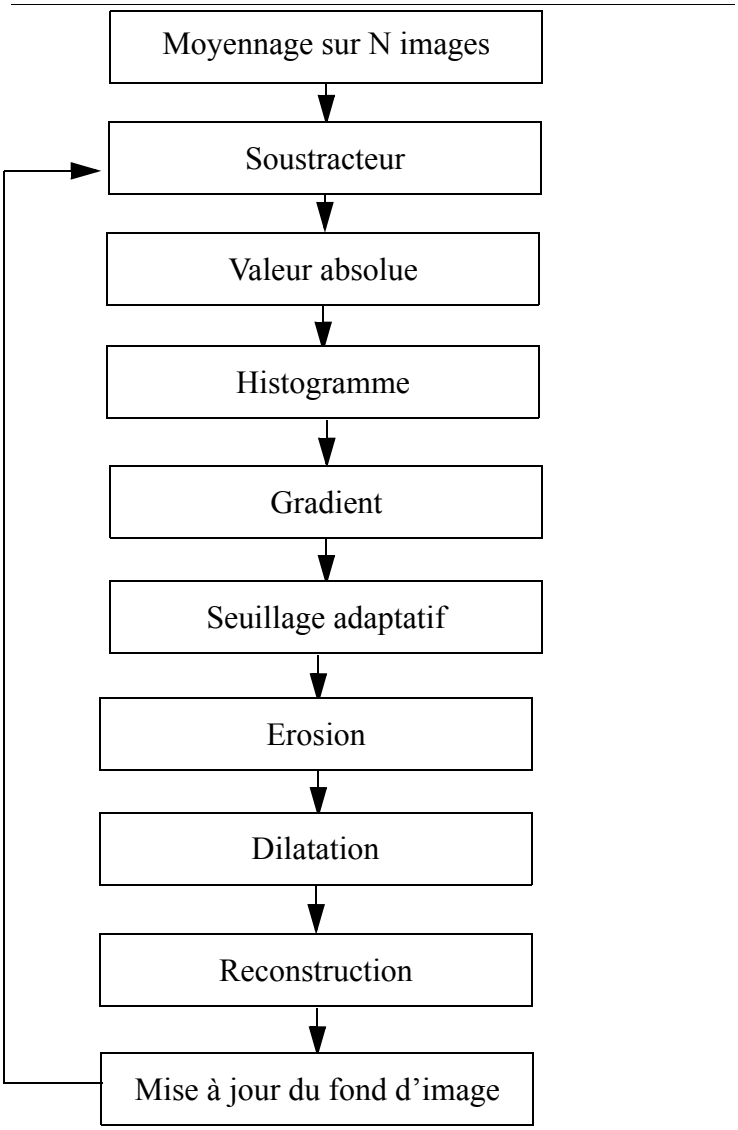
- Un moyennage paramétrable sur quelques images brutes a pour objectif de limiter l'impact du bruit d'une image à l'autre,
- Une soustraction avec l'image de fond permet de distinguer les zones en mouvement des zones fixes. En fait, l'algorithme effectue la différence pixel à pixel entre les deux images. Les zones fixes apparaîtront en noir et les autres en différents niveaux de gris,
- Une opération de seuillage binarise l'image afin de faciliter les traitements sur les objets,
- Un traitement dit "morphologique" filtre l'image pour faire disparaître tous les points isolés. L'algorithme convertit un pixel blanc en noir si au moins un pixel voisin est noir. Ce traitement élimine le bruit sur l'image. Pour ce faire, il est réalisé en trois sous-étapes qui sont l'érosion, la dilatation et la reconstruction. La première sous-étape élimine le bruit mais aussi des détails des objets en mouvement, on effectue alors une dilatation et une reconstruction afin de limiter la perte d'informations.

Ce traitement d'images doit prendre au maximum 40 ms par images, pour respecter un débit de 25 images par seconde.

1. www.telecom.gouv.fr/rntl/FichesA/Epicure.htm

La suite des traitements de l'application est représentée dans le graphe de la figure 17 ci-dessous.

FIGURE 17. Détection de mouvement sur fond d'image fixe



A partir du code C de l'application, nous avons estimé avec VESTIM la consommation de ces traitements ou tâches sur un processeur OAK. Nous présentons ces résultats dans le tableau 5.

Tableau 5: Résultats d'estimation de consommation et performances de l'application cible sur un OAK

Tâche	Energie (mJ)	Nombre de cycles
Moyenne, soustraction, valeur absolue	0,9	520328
Seuillage adaptatif	2,8	1951058
Reconstruction	17,5	12228489

Chaque tâche est constituée de fonctions C estimées de manière séparée. Prenons par exemple la première tâche du tableau 5 «Moyenne, soustraction, valeur absolue». Le détail des contributions de chacune des fonctions est présenté au tableau 6:

Tableau 6: Coûts en consommation et en cycle des fonctions les plus représentatives contribuant à Moyenne, soustraction, valeur absolue

fonction C	Energie (mJ)	Nombre d'exécutions	Nombre de cycles
ic_add	$2{,}2.10^{-5}$	1	14
ic_div	$2{,}98.10^{-1}$	1	195129
ic_substract	8.10^{-6}	1	5
ic_absolute	$5{,}9.10^{-1}$	1	325180

Un avantage de cette approche par rapport à une méthode plus abstraite qui opère à partir de caractéristiques générales du code [JN04] est sa faculté à analyser avec précision les parties de code orientées «contrôle», c'est-à-dire où le flot de contrôle est dépendant des données. Cependant, il est nécessaire de considérer des séquences de test mettant en évidence les flots de contrôle sur lesquels on souhaite obtenir des estimations. A partir de ces estimations automatisées des codes C exécutés par les DSP, il est possible de renseigner les bibliothèques de CODEF afin d'estimer la consommation au niveau système d'une architecture de SOC.

2•Estimation de la consommation de la partie matérielle

Afin de disposer aussi de valeurs de consommation d'accélérateurs matériels dans les bibliothèques de CODEF, nous avons procédé de façon plus pragmatique en codant les fonctions associées en VHDL. Après synthèse, ces codes sont estimés en consommation grâce à un outil commercial, WattWatcher. Nous disposons alors de la surface en mm^2, de la puissance moyenne (mW) et du temps d'exécution (en ms).

Cet outil a la possibilité d'estimer la consommation de trois manières différentes sur des changements des valeurs des données. Premièrement, le mode probabiliste exploite uniquement des probabilités de changements des valeurs des données en fonction de la fréquence de variation des entrées; deuxièmement, le mode semi-probabiliste exploite ces probabilités ainsi que des résultats de simulation; et enfin, le dernier mode exploite uniquement des résultats de simulation.

Après avoir synthétisé le code, il associe à chaque type de porte un comportement de consommation, et le mode (probabiliste ou non, ou mixte...) fixe une activité de commutation pour toutes les parties du circuit.

De même que dans le paragraphe précédent, présentons les accélérateurs développés pour l'application de détection de mouvement de la caméra embarquée (tableau 7). Les estimations ont été faites en mode probabiliste. Nous avions comparé ce mode avec le mode simulation sur des tests et l'écart entre les deux estimations était faible, aussi nous n'avons considéré que le mode probabiliste.

Tableau 7: Résultats de synthèse et d'estimation des accélérateurs matériels

accélérateur dédié à la tâche	surface (mm^2)	Nombre de cycles	puissance (mW)
Moyennage soustraction et Valeur absolue	0,8	65025	2,63
Erosion, dilatation	0,01	65025 65025	0,19 pour Erosion, 0,19 pour Dilatation
Histogramme	0,02	65025	1,98
Gradient	0,02	65025	13,8
seuillage	0,001	65025	0,03

Passons maintenant à l'estimation d'une architecture complète, c'est-à-dire composée à la fois d'unités type processeur et accélérateur matériel.

3•Estimateur de la consommation au niveau système

Afin de prendre en compte la consommation lors de la conception globale d'une architecture pour une application donnée, nous avons étendu CODEF afin de calculer la consommation de chaque architecture fournie par l'outil.

A la fin du partitionnement effectué par CODEF, les solutions précisent l'allocation de chaque tâche sur les unités de l'architecture et leur ordonnancement.

Nous avons donc diverses solutions d'architectures systèmes décrites par un graphe de flots de données d'une application qui respectent des échéances temporelles des tâches. Lors de la constitution de ces solutions, CODEF a pour objectif de réduire la surface de silicium.

L'objectif de l'estimateur de consommation niveau système est de déterminer, pour chaque solution, l'énergie consommée de manière globale, ainsi que les pics de puissance. Nous détaillons ces deux estimations dans la suite.

3.1.Préludes à l'estimation: les bibliothèques.

Chaque solution issue du partitionnement se présente comme une liste d'allocations de chaque tâche sur les unités de l'architecture avec leurs dates de début et de fin d'exécution.

Pour effectuer l'estimation globale de la consommation, il est nécessaire de disposer d'une description des consommations des implémentations de chaque tâche sur les unités. Ces renseignements ont été introduits dans les bibliothèques de CODEF.

-bibliothèque d'unités : Nous avons ajouté des informations concernant les puissances de repos, P_{repos} (en mW) des unités lorsqu'elles sont inactives, les puissances en mode basse consommation pour les processeurs qui en disposent: P_{idle} et P_{sleep} (en mW).

Pour chaque unité, est indiqué aussi si l'unité est capable de changer de tension dynamiquement, dans un champ nommé *DVS*.

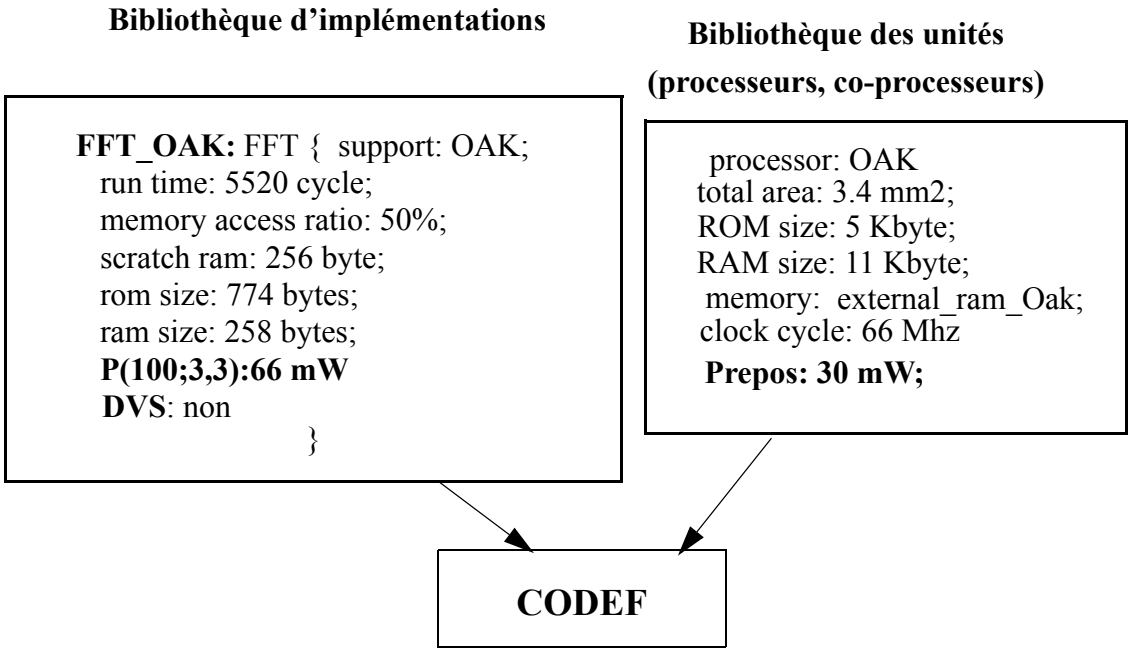
-bibliothèque d'implémentations :

Aux informations détaillées dans le paragraphe 5.2 du chapitre trois relatives à la bibliothèque d'implémentations, nous avons ajouté la puissance moyenne d'exécution d'une tâche sur une unité. Pour les unités capables de changer de tension et de fréquence conjointement, nous avons indiqué pour chaque possibilité d'exécution de

tâches par les unités la valeur de la puissance moyenne consommée par couple <fréquence,tension> : P(f;V).

Sur la figure 18 est décrit un exemple de bibliothèque renseignée pour l'estimation de la consommation**.**

FIGURE 18. Bibliothèques de CODEF



La puissance de repos du DSP OAK est de 30 mW et la puissance moyenne d'exécution de la tâche FFT sur ce processeur est de 66 mW.

3.2.Méthode d'estimation de la consommation.

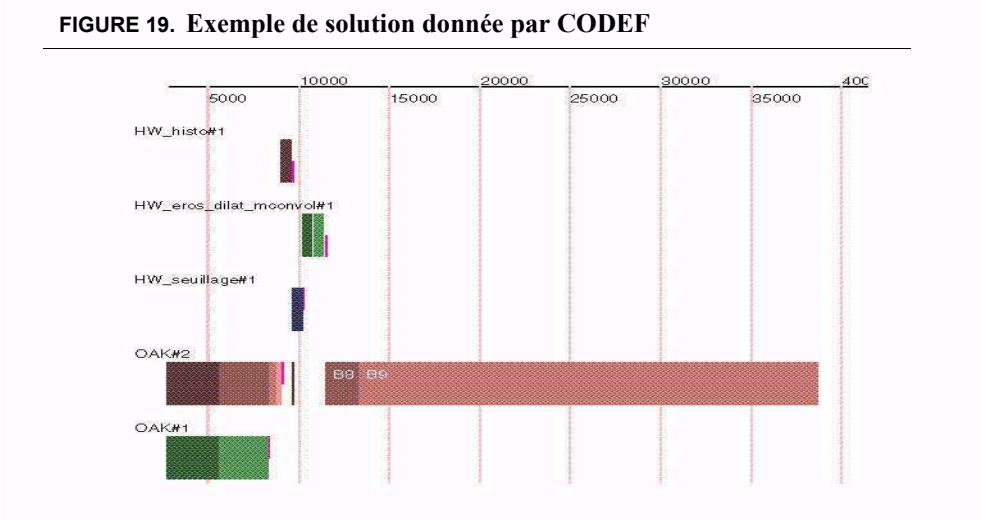
3.2.a.Energie dissipée par les unités.

L'objectif est d'estimer l'énergie dissipée lors de l'exécution d'une application par l'architecture construite par CODEF, ainsi que les pics de puissance.

On estime l'énergie de l'application en tenant compte des énergies dues aux exécutions des tâches sur les unités, des énergies dues aux diverses périodes de repos des unités, et celles dues aux accès mémoires et à leurs périodes de repos.

Dans l'exemple proposé sur la figure 19, obtenu à partir de CODEF, les tâches de l'application sont allouées et ordonnancées sur le processeur OAK et sur des accélérateurs matériels.

FIGURE 19. Exemple de solution donnée par CODEF



Chaque tâche allouée à une unité matérielle ou logicielle (unité matérielle ou processeur type DSP ou RISC) possède une date de début et une date de fin d'exécution. A partir des puissances consommées pour chaque tâche indiquée dans la bibliothèque des implémentations et des puissances de repos des processeurs précisées dans la bibliothèque des entités, il est possible de calculer une estimation de l'énergie consommées par l'exécution de toute l'application sur l'architecture construite :

$$E_{total} = \sum_{i=1}^{U}\left(E_{repos_i} + \sum_{j=1}^{N_i} E(\tau_{a(j,i)})\right) \text{ avec } E_{repos_i} = \left(T_{max} - \left(\sum_{j=1}^{N_i}(t^f_{\tau_{a(j,i)}} - t^d_{\tau_{a(j,i)}})\right)\right) P^i_{repos} \text{ et}$$

$$E(\tau_{a(j,i)}) = (t^f_{\tau_{a(j,i)}} - t^d_{\tau_{a(j,i)}}) \times P^i(\tau_{a(j,i)}) + E^i_{accès}(\tau_{a(j,i)}), \text{ où}$$

- U est le nombre d'unités dans l'architecture,
- N_i le nombre de tâches de l'application allouées à l'unité i,
- Tmax est le temps total d'exécution de l'application de l'architecture,
- a(j,i), la fonction inverse d'allocation indiquant les numéros de tâches allouées à l'unité i,
- t^f (respectivement t^d), la date de fin (respectivement de début) d'exécution de la tâche sur l'unité. Ces dates de début et de fin d'exécution sont données par CODEF à l'issue du partitionnement/ordonnancement.
- la valeur $P^i(\tau_{a(j,i)})$ est la puissance d'exécution de la tâche sur l'unité i.
- l'énergie $E^i_{accès}(\tau_{a(j,i)})$ est due aux accès à la mémoire externe (figure 20) lorsque la tâche est allouée par CODEF à une unité i dont l'espace mémoire interne libre n'est pas suffisant pour contenir entièrement la tâche. Cette

énergie pour une tâche τ_k exécutée par l'unité i connectée à une mémoire externe m se calcule par :

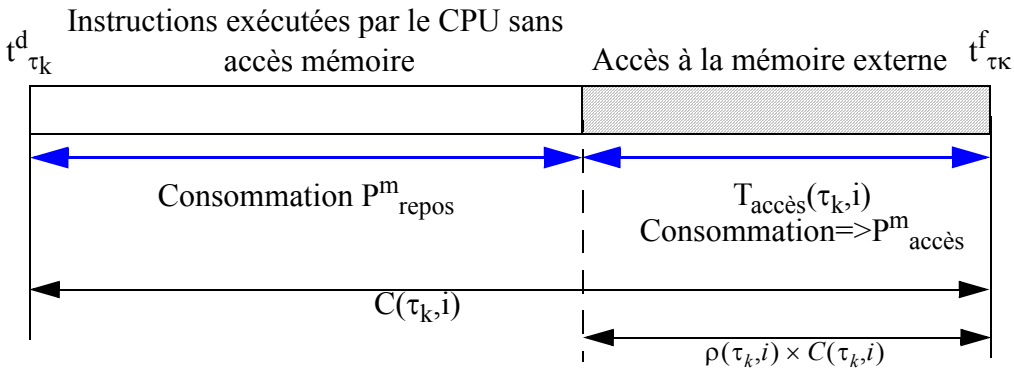
$$E_{accès}^{i}(\tau_k) = T_{accès}(\tau_k,i) \times P^{m}_{accès} + (t^{f}_{\tau_k} - t^{d}_{\tau_k} - T_{accès}(\tau_k,i)) \times P^{m}_{repos},$$

avec $T_{accès}(\tau_k,i) = \rho(\tau_k,i) \times C(\tau_k,i) \times t_m$ où

- $C(\tau_k,i)$ est le nombre de cycles pour exécuter τ_k sur l'unité i (sans tenir compte des cycles d'attente dus au temps de réponse de la mémoire externe),
- $\rho(\tau_k,i)$ est le taux de cycles de $C(\tau_k,i)$ où la tâche accède à la mémoire,
- t_m, le temps d'accès à la mémoire externe,
- la quantité $T_{accès}(\tau_k,i)$ représente le temps pendant lequel la tâche exécutée par l'unité i accède à la mémoire externe (figure 20),
- la valeur $P^{m}_{accès}$ représente la puissance due à un accès mémoire et P^{m}_{repos} celle liée à la mémoire externe lorsqu'elle est au repos.

Les valeurs P^{i}_{repos}, $P^{i}(\tau_k)$, $P^{m}_{accès}$, P^{i}_{repos}, $\rho(\tau_k,i)$, $C(\tau_k,i)$, t_m sont indiquées dans la bibliothèque de CODEF. Les autres valeurs N_i, $a(j,i)$, $t^{f}_{\tau a(j,i)}$, $t^{d}_{\tau a(j,i)}$ sont issues du résultat de partitionnement.

FIGURE 20. Contribution de la mémoire externe à l'énergie globale



3.2.b.Pics de puissance.

Pour estimer les valeurs des pics de puissance, on détermine les intervalles où les unités exécutent les tâches de l'application. Ces intervalles sont déduits des dates de début et de fin des tâches données par CODEF. On ajoute alors les puissances des tâches dont les exécutions ont un intervalle de temps en recouvrement, et on ajoute également la puissance de repos des unités en attente de tâches à exécuter. On rappelle que les pics de puissance sont à éviter afin d'améliorer la fiabilité du circuit et de prolonger l'autonomie des batteries.

A partir des résultats de l'ordonnancement et de l'allocation fournis par CODEF, on construit l'ensemble $D = \{\langle t_{\tau_k}^{e}, e\rangle\}$ des évènements de début et de fin des tâches de l'application. L'ensemble D est ordonné par ordre croissant des dates t_{τ_k} où e désigne un début (d) ou une fin (f) de tâche. Soit ϕ_k, la fonction d'allocation qui à la tâche τ_k fait correspondre l'unité u_i sur laquelle la tâche est allouée.

Le calcul des pics de puissance se calcule en parcourant itérativement les évènements de D :

$$P_0 = \sum_{i=1}^{U} P^{i}_{repos};$$

Pour $x \in D$, faire

- si x.e = d, alors $P_x = P_x\text{-}(P^{\varphi_k}_{repos}) + P^{\varphi_k}(\tau_k)$;
- si x.e = f, alors $P_x = P_x + P^{\varphi_k}_{repos} + P^{\varphi_k}(\tau_k)$;

fin pour;

A partir de l'évaluation de la puissance P_x, il est aisé d'en déterminer les pics de puissance. Pour compléter cette analyse de consommation sous forme de pics de puissance ou d'énergie, il est nécessaire de tenir compte des transferts de données entre les unités.

3.3.Prise en compte de la consommation due aux communications.

Le problème de la prise en compte de la consommation due aux communications est complexe du fait qu'elle est fortement dépendante du placement-routage et de la topologie de l'interconnexion définitive. Ceci rend difficile la définition de modèles de consommation des bus ou des réseaux sur puce. Dans notre cas, nous avons choisi d'associer un coût moyen dû aux communications.

Les outils de synthèse de micro-réseaux présentent l'avantage de fournir une description structurelle qui peut être exploitée par des outils d'analyse de la consommation. Ceci nécessite de raffiner fortement l'architecture et l'application, ce qui est peu compatible avec une approche initiale de niveau système. Par ailleurs, CODEF fournit la durée d'un transfert, à partir d'un nombre de données à transférer entre deux nœuds, et du débit du bus utilisé.

On peut donc définir un modèle prenant en compte le nombre de transactions sur les bus, le nombre d'accès aux mémoires de communication. En fait, il s'agirait de distinguer entre les communications synchrones pour lesquelles seul le coût dû aux transferts sur le bus est à considérer et les communications asynchrones, pour lesquelles il faut prendre en compte également le coût relatif aux mémoires de com-

munication. CODEF est optimisé de manière à maximiser les communications synchrones. Le coût en consommation le plus important à prendre en compte est donc le coût dû aux transactions sur les bus. Nous avons intégré un modèle de consommation des transactions sur les bus dans notre estimation qui prend en compte les valeurs des données de manière statistique, après simulation sur des outils commerciaux tels que CADENCE.

Pour les communications asynchrones, il est possible d'évaluer une consommation liée aux accès à la mémoire de communication, mais dans le cas synchrone, il est difficile d'obtenir une estimation de la consommation avec une précision compatible avec celle des modèles précédents. Des travaux sur ce thème sont nécessaires.

3.4.Résultats de l'estimateur de consommation.

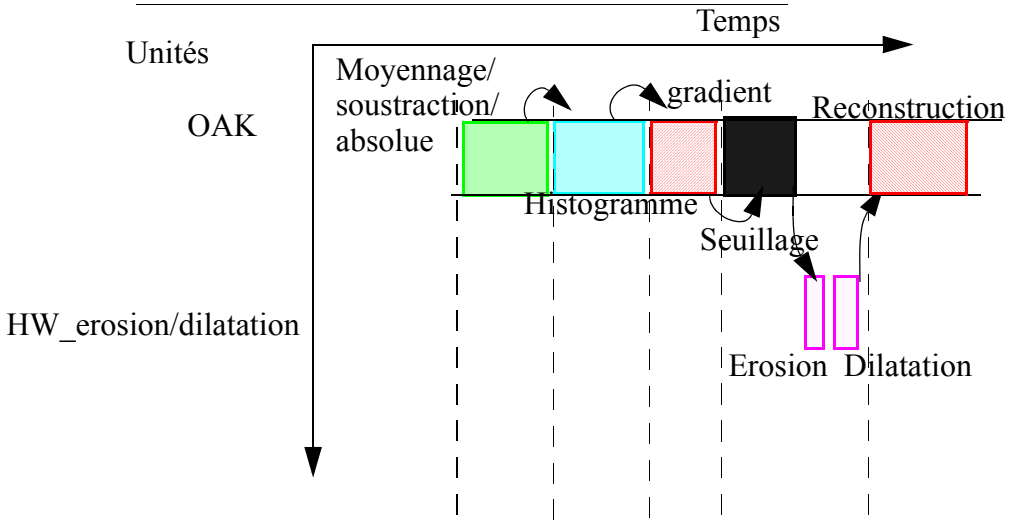
Nous avons testé notre estimateur de consommation sur une application de détection de mouvement dans une caméra embarquée présentée au paragraphe 1.3.cde ce chapitre. Nous présentons ici quelques résultats et tout d'abord, voyons quelles solutions CODEF a produit pour cette application.

Pour les différentes tâches de l'application, nous avons utilisé l'outil VESTIM pour déterminer les estimations en performance et en consommation du processeur OAK. Ce processeur n'est pas le plus adapté à ces traitements mais on peut aisément transposer l'approche à un autre processeur, à condition qu'un modèle de consommation soit défini pour ce processeur. En ce qui concerne la consommation des implémentations matérielles des tâches, nous avons procédé comme indiqué précédemment à partir d'une description VHDL. Dans un premier temps, nous avons réalisé un partitionnement ne tenant compte que des contraintes de temps et d'une minimisation de la surface de silicium. Nous avons obtenu deux solutions.

Voyons maintenant une solution en détail.

Elle utilise un accélérateur matériel pour les tâches d'érosion et de dilatation (figure 21).

FIGURE 21. Une solution fournie par CODEF



Nous voyons sur la figure 21 la répartition temporelle des tâches et leur allocation sur les unités.

A partir des estimations des puissances consommées des unités et des tâches, nous avons estimé la consommation en énergie et en puissance de chacune des deux solutions. Nous avons supposé que les parties matérielles au repos ont une consommation négligeable, car dans leur cas, il est aisé d'utiliser la technique du *clock gating*. Le processeur, lui, consomme toujours, même au repos. Les résultats d'estimation de l'énergie consommée et des pics de puissance sont résumés dans le tableau 8 ci-dessous:

Tableau 8: Résultats de l'estimation en consommation des solutions trouvées

solution	temps d'exécution(ms)	surface (mm^2)	Energie (μJ)	pic de puissance max (mW)
sol0	33.35	5.051	244	144
sol1	39.76	3.409	450	112

La deuxième solution sol1 est composée du OAK, et d'un seul accélérateur matériel exécutant les tâches d'érosion et de dilatation. La première est constituée d'un autre processeur, le StrongARM et de quatre accélérateurs matériels.

Notons ici simplement la variation sur les pics de puissance et l'énergie. Dans ce cas précis, la raison en est que la solution sol0 comporte à la fois plus d'accélérateurs que la solution sol1 (les accélérateurs matériels sont peu coûteux en énergie mais ajoute de la surface), et à la fois un processeur différent du OAK présent dans la

sol1, le StrongARM dont l'architecture est pensée «économie d'énergie». De larges possibilités d'optimisation sont donc ouvertes.

4•Conclusion

Dans ce chapitre, nous avons présenté différentes étapes relatives à l'estimation de la consommation d'une architecture système.

La première concerne la modélisation de la consommation de processeurs de traitement du signal, comme le OAK et le PALM. Cette modélisation a été réalisée au niveau des unités fonctionnelles par analyse des consommations des instructions assembleurs. Les modèles de consommation de ces DSP ont été implémentés dans un estimateur de performances adapté aux architectures de ces processeurs. Ainsi, il est possible d'obtenir rapidement l'estimation en performance et en consommation d'un code C exécuté par un DSP. Pour les accélérateurs matériels, les modules sont développés en VHDL, simulés, synthétisés et estimés en consommation à l'aide d'un logiciel commercial. L'ensemble de ces informations est ensuite placé dans les bibliothèques de CODEF, ce qui permet d'estimer la consommation au niveau système d'une application décrite par un graphe de flots de données.

Cette estimation prend en compte également des coûts moyens de la consommation des mémoires et des communications.

Ces coûts moyens seraient à affiner par une étude plus poussée afin de définir des modèles plus précis de consommation.

Nous nous sommes dotés de méthodes d'estimation au niveau système de la consommation en énergie et des pics de puissance, ce qui nous permet maintenant de développer des techniques d'optimisation. Ces techniques font l'objet du chapitre suivant.

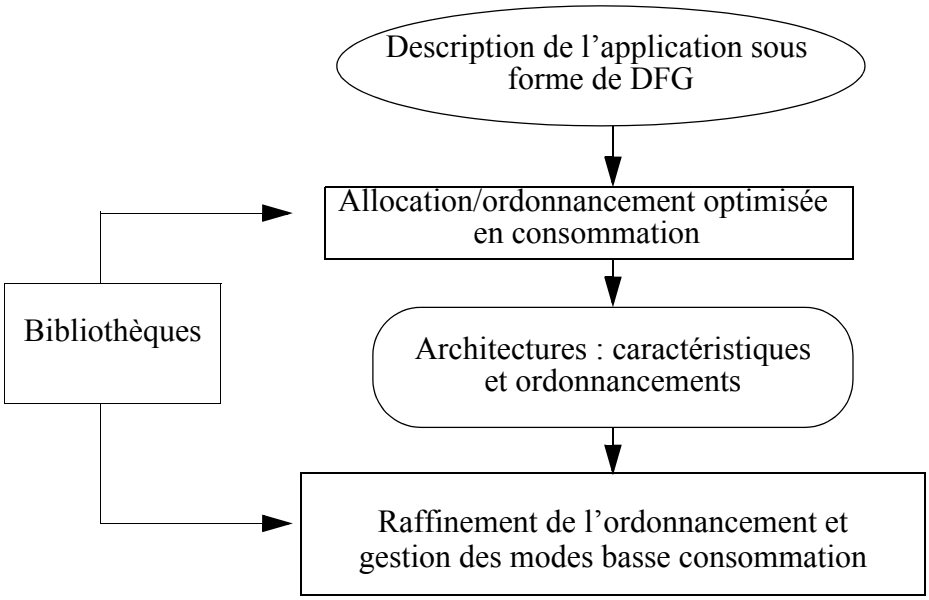
Chapitre V : Optimisation de la consommation lors de la conception et Résultats

Introduction

Maintenant que nous disposons d'une méthode d'estimation de la consommation d'une architecture système, nous pouvons passer à l'étude de la conception optimisée en consommation de systèmes autonomes.

Dans ce chapitre, nous décrivons les stratégies d'optimisation explorées qui interviennent durant l'étape d'allocation/ordonnancement de CODEF, et également sous forme de post-traitements par raffinement de l'ordonnancement. La figure 23 situe ces interventions citées.

FIGURE 23. **CODEF-LowPower**



Dans ce chapitre, nous décrivons :

- les modifications de l'étape d'allocation/ordonnancement afin d'explorer des allocations qui privilégient une faible consommation,
- l'étape de raffinement d'ordonnancement : par ajustement conjoint en tension et en fréquence (DVS/DFS) des entités de type processeur. Les stratégies de raffinement d'ordonnancement visent à exploiter au mieux les techniques de *Dynamic Voltage Scaling* (DVS) et de *Dynamic Frequence Scaling* (DFS).
- l'étape de gestion des modes basse consommation (*idle, sleep*) afin de profiter de ces modes lorsque les ajustements DVF/DFS ne peuvent être appliqués efficacement.

Ces optimisations ont été testées d'une part, sur des exemples pris dans la littérature, et d'autre part, sur une application de détection de mouvement pour caméra embarquée.

1•Etape d'allocation/ordonnancement optimisée

L'objectif de cette optimisation est de sélectionner les implémentations les moins coûteuses en énergie tout en respectant prioritairement les contraintes de temps. Le choix des tâches à ordonnancer est réalisé dans le sens de l'urgence temporelle et elles sont rangées dans les listes en fonction de la surface et de la consommation. La sélection s'effectue :

- en déterminant la liste dans laquelle est choisie l'implémentation, ce qui minimise la surface,
- en déterminant l'implémentation la moins coûteuse en énergie ou en puissance (tâche sur unité) parmi les implémentations possibles présentes dans la liste sélectionnée à l'étape précédente. Cette étape minimise l'énergie ou la puissance.

Détaillons ces étapes.

1.1.Méthode d'allocation optimisée en consommation.

Dans CODEF, la priorité des listes est principalement temporelle, ce qui consiste à choisir une implémentation parmi les listes dans l'ordre suivant : L^1_{reuse}, L^2_{reuse}, L^1_{new}, L^2_{new}, L_{other}. Au sein d'une liste, les implémentations sont organisées afin de minimiser la surface. Ainsi, dans cette première optimisation en consommation, nous trions les implémentations en fonction de l'énergie au sein de la liste choisie.

Formalisons cette optimisation. Soit une application constituée de n tâches et un ensemble U d'unités de calcul. L'ensemble des implémentations des tâches τ_i sur les unités est représenté par $I = \{<\tau_i,u_k>\} = I_1 \cup I_2 \cup I_3 \cdots \cup I_n$, avec

$I_i = \{<\tau_i,u_k> \text{ et } u_k \in U\}$.

Le partitionnement réalisé par CODEF consiste à construire une application injective de I dans L, où L est l'ensemble des listes L^1_{reuse}, L^2_{reuse}, L^1_{new}, L^2_{new}, L_{other} correspondant à l'allocation des implémentations des tâches. Les listes de L sont constituées par distribution des éléments de I_i en fonction du critère de temps. Pour une tâche τ_i, l'allocation privilégie d'abord L^1_{reuse}, puis L^2_{reuse}, qui correspondent à des unités u_k déjà présentes dans l'architecture. La différence entre ces deux listes tient au fait que L^2_{reuse} correspond à des implémentations $<\tau_i,u_k>$ vérifiant que l'écart entre la contrainte de temps et la longueur du chemin temporel issu de τ_i vers les tâches terminales du graphe est supérieur à un seuil donné, considéré comme facteur discriminant d'un point de vue de la criticité en temps.

Or, on peut considérer que les implémentations de L^1_{reuse} et L^2_{reuse} ne sont pas critiques en temps. Par conséquent, on considère dans la phase d'allocation une seule liste $L_{reuse}=L_{reuse}^{1} \cup L_{reuse}^{2}$ pour se donner plus de possibilités d'optimisation en consommation. Les implémentations $<\tau_i,u_k>$ de L_{reuse} sont triées par ordre croissant de leur consommation d'énergie et, s'il y a égalité entre les énergies de deux implémentations, c'est le coût en puissance qui est considéré.

Si L_{reuse} ne permet pas d'utiliser une unité déjà présente pour exécuter une autre tâche, le choix se fera parmi les autres listes. Cette étape est à itérer pour les n tâches.

Décrivons comment sont organisées les listes en fonction de l'énergie ou de la puissance.

Pour trier les implémentations au sein d'une liste, nous avons remplacé le terme Ω, décrit dans le chapitre trois, par la paramètre ζ prenant en compte l'énergie et la puissance. L'implémentation ayant la valeur de ζ minimum pour réaliser la tâche la plus critique en temps sera sélectionnée.

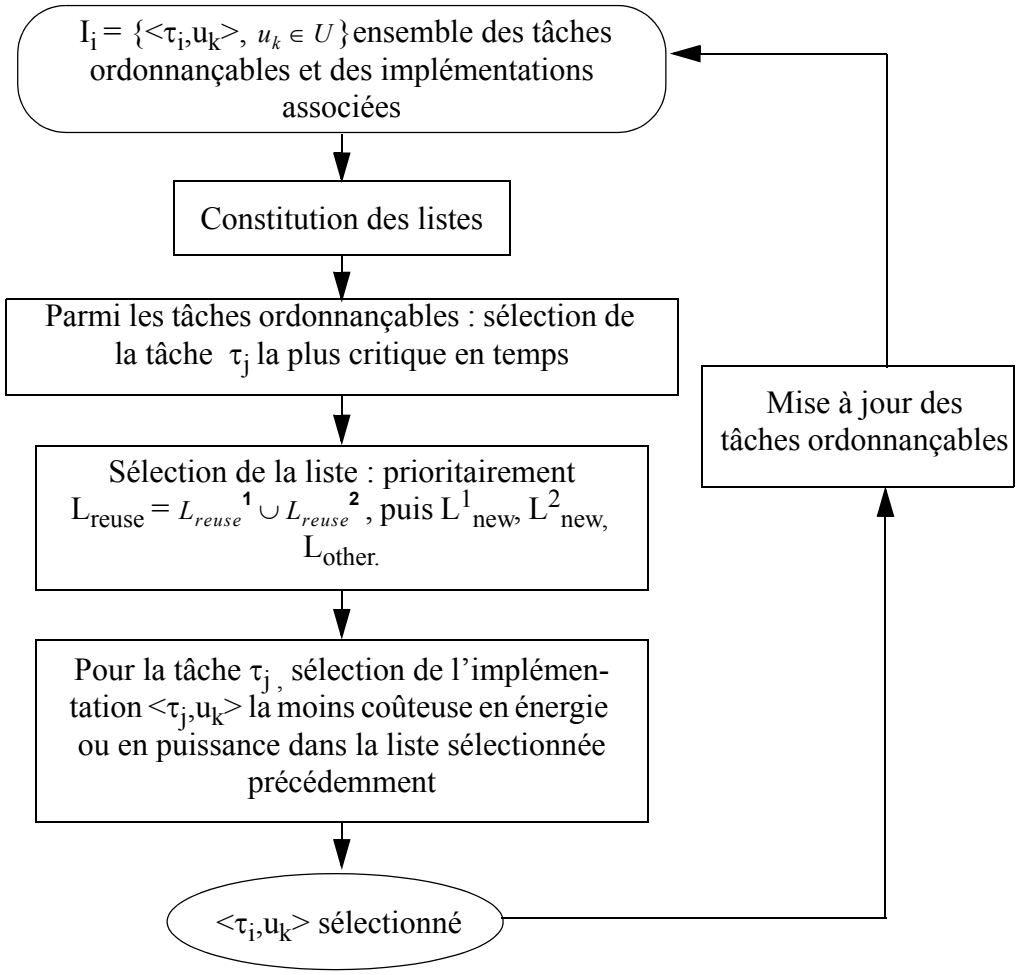
Ce terme, $\zeta_E(i, l)$, est introduit en considérant l'énergie dissipée par l'exécution de chaque tâche sur chaque unité capable de l'exécuter :

$\zeta_E(i, l) = \dfrac{E(\tau_i,u_l)}{\frac{1}{m} \times \left(\sum_{j}^{m} E(\tau_i, u_j) \right)}$, avec $E(\tau_i,u_l)$, l'énergie du couple $<\tau_i,u_l>$, $\left(\sum_{j}^{m} E(\tau_i, u_j) \right)$, la somme des énergies de toutes les implémentations possibles pour τ_i sur les unités u_j capables d'exécuter τ_i, et m, le nombre total de ces implémentations.

De même, l'algorithme peut privilégier la puissance grâce à un terme équivalent lorsque deux implémentations ont la même énergie : $\zeta_P(i, l) = \dfrac{P(\tau_i,u_l)}{\frac{1}{m} \times \left(\sum_{j}^{m} P(\tau_i, u_j) \right)}$, avec $P(\tau_i,u_l)$, la puissance moyenne du couple $<\tau_i,u_l>$, et $\left(\sum_{j}^{m} P(\tau_i, u_j) \right)$, la somme des puissances de toutes les implémentations possibles pour τ_i sur les unités u_j capables de réaliser τ_i et m, le nombre total de ces implémentations.

Le principe général de la sélection de l'allocation basse consommation est illustré par la représentation de la figure 24 :

FIGURE 24. Allocation/ordonnancement basse consommation



1.2.Résultats.

Dans ce paragraphe, nous présentons les résultats de l'optimisation lors de l'allocation. Nous comparons ensuite la solution obtenue avec celle obtenue par Programmation Linéaire en nombres entiers. Nous verrons que le choix de la solution optimisée sera alors confirmé. Afin d'obtenir de nouvelles solutions, nous affaiblissons ensuite les contraintes en surface et en temps (allocation avec réutilisation au maximum). La conséquence est de diminuer l'énergie et la puissance au prix d'une augmentation de surface.

Nous avons testé les optimisations d'allocation sur l'application de détection de mouvement. Dans un premier temps, on utilise la version initiale de CODEF pour

explorer des solutions contraintes suivant le critère de surface. Ces solutions sont ensuite comparées à celles obtenues par CODEF-LowPower qui réalise une allocation optimisée en consommation.

1.2.a.Solutions générées par CODEF pour l'application de détection de mouvement.

La figure 25 permet de visualiser rapidement les caractéristiques en surface et en énergie des solutions générées par CODEF, avant optimisation.

Rappelons que CODEF explore plusieurs solutions qui correspondent à des variations des seuils T_{reuse} et T_{new}. Dans le cas de la figure 25, cinq solutions ont été obtenues ayant des temps d'exécution compris entre 34 ms et 40 ms. L'ensemble des caractéristiques de ces solutions est détaillé dans les tableaux 9 et 10.

FIGURE 25. Comparaison des solutions avant optimisation

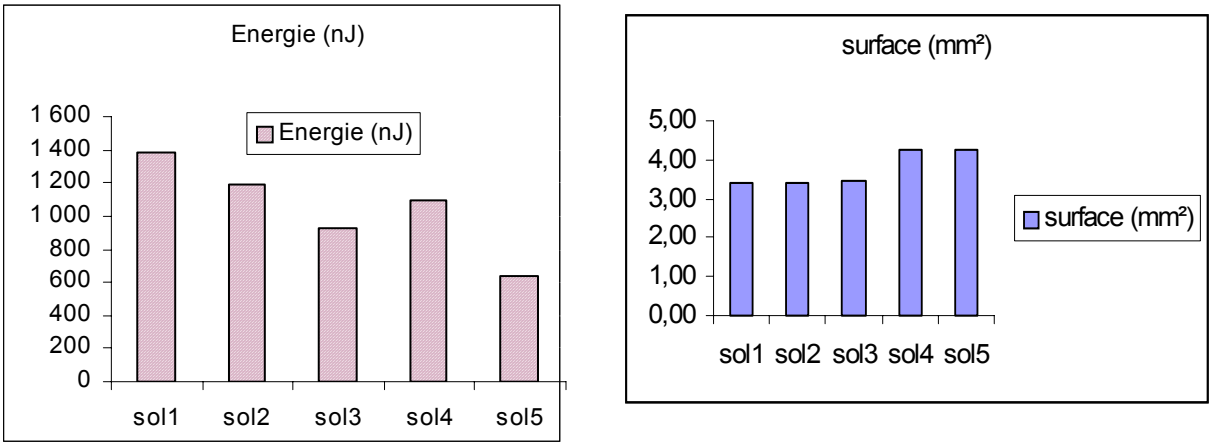


Tableau 9: Caractéristiques des solutions obtenues par CODEF

sol	temps d'exécution (ms)	surface (mm^2)	Energie (nJ)	pic de puissance (mW)
sol1	39.76	3.409	1 389	112
sol2	37.32	3.416	1 189	112
sol3	36.188	3.437	931	112
sol4	38.529	4.233	1 092	112
sol5	34.952	4.261	642	112

Tableau 10: Les architectures produites par CODEF

unités / solution	OAK	HW_erosion/ dilatation	HW_moyennage /absolue	HW_seuillage	HW_histo
sol1	X	X			
sol2	X	X		X	
sol3	X	X		X	X
sol4	X	X	X		
sol5	X	X	X	X	X

De manière générale, on voit avec ces solutions, que plus le parallélisme est augmenté (i.e. la surface), plus l'énergie est diminuée. Toutes ces solutions ont le même pic de puissance.

Après application de la stratégie décrite précédemment, CODEF-LowPower génère la solution sol5. En effet, cette solution est la plus économique en énergie. Bien que la surface soit plus grande que pour les autres, elle permet une exécution plus rapide et plus économique en énergie.

A ce niveau, essayons de voir si le compromis choisi par CODEF-LowPower est bien le meilleur.

1.2.b.Comparaison avec un partitionnement réalisé par programmation linéaire en nombres entiers.

L'utilisation de la programmation linéaire en nombres entiers nous permettra de retrouver l'architecture compromis trouvée dans les paragraphes précédents et de prouver ainsi que ce compromis est bien le meilleur.

Détaillons les notations:

Pour une tâche τ_i, son exécution par le module software (respectivement par le module matériel) aura une durée d_{softi} (respectivement, d_{hardi}) et une puissance P_{softi} (respectivement, P_{hardi}), donc, globalement, pour τ_i, on a les caractéristiques: (d_{softi},d_{hardi}; P_{softi}, P_{hardi}).

Définissons des booléens b_i, représentant l'instanciation en logiciel ou matériel, i étant le numéro de la tâche. Si $b_i = 1$, la tâche τ_i est exécutée en logiciel, sinon elle est exécutée en matériel.
Nous explicitons le problème de la manière suivante: la fonction à minimiser concerne l'énergie du système, et les équations suivantes concernent les contraintes.
La fonction concernant l'énergie d'un système à deux tâches donne cette équation:

$b_1*(P_{soft1}*d_{soft1})+(1-b_1)*(P_{hard1}*d_{hard1}) + b_2*(P_{soft2}*d_{soft1})+(1-b_2)*(P_{hard2}*d_{hard2})$.

Il faut donc minimiser, pour une application de n tâches:

$$\min: \sum_{i=1}^{n} b_i \times (P_{softi} \times d_{softi}) + (1-b_i) \times (P_{hardi} \times d_{hardi})$$

De même, les contraintes doivent être prises en charge, telles que:

-la surface: $\sum_{\text{ensemble des unités présentes}} b_i \times S \times xi + (1-b_i) \times S_{hardware} < \text{contrainte en surface}$

avec S, la surface du processeur logiciel, xi=1 si le processeur n'a pas déjà été instancié, xi=0 si il l'a déjà été.

-le temps: $\sum_{i=1}^{n} b_i \times d_{softi} + (1-b_i) \times d_{hardi} < \text{contrainte de temps}$.

Comme les tâches sont séquentielles, il n'y a pas d'autres contraintes de dépendances à ajouter.

Le résultat obtenu est:

b1	1
b2	0
b3	0
b4	1
b5	0
b6	0
b7	0
b8	1
b9	1

Les tâches numéros 1, 4, 8 et 9 sont en logiciel, et les tâches numéros 2, 3, 5, 6, 7 sont en matériel, ce qui correspond à la solution sol5.
Nous retrouvons bien le compromis sélectionné par CODEF dans le but d'économiser de l'énergie.

Afin d'exploiter les caractéristiques d'un processeur doté de techniques de gestion de puissance, nous avons ajouté dans les bibliothèques le StrongARM aux diverses fréquences. Nous présentons dans le tableau qui suit les solutions générées par CODEF. Le graphe de la figure 26 permet de visualiser les solutions comportant le StrongARM, sans le OAK et sans optimisation en consommation.

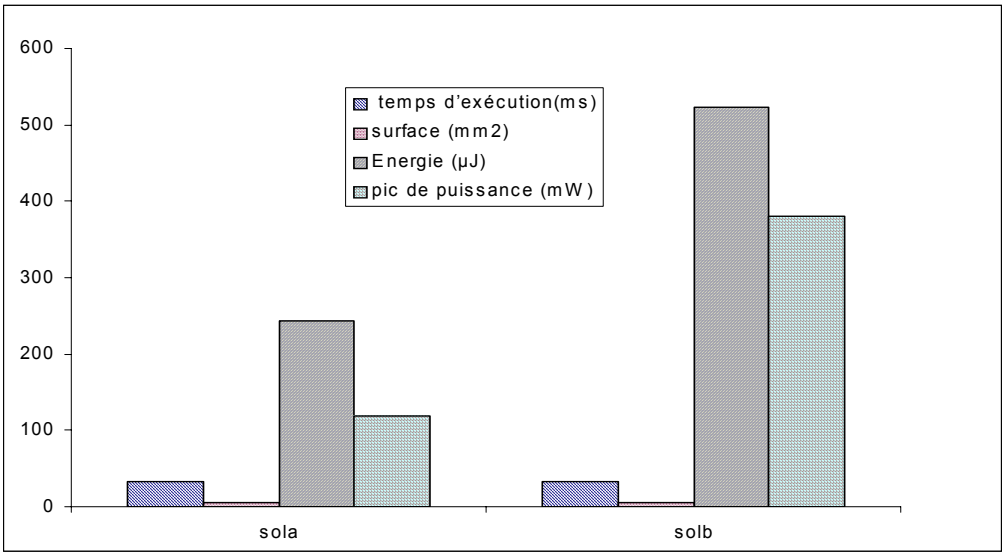


FIGURE 26. solutions avant optimisation avec le StrongARM

Tableau 11: Architectures, contenant le StrongARM proposées par CODEF pour l'application de détection de mouvement par une caméra embarquée

solution	temps d'exécution(ms)	surface (mm^2)	Energie (µJ)	pic de puissance (mW)
sola	33.35	5.051	244	118
solb	32.60	5.051	524	380

Ci-dessous, un tableau détaille les architectures des deux solutions.

Tableau 12: Détails des architectures du tableau 11

unités / sol	HW_erosion /dilatation	HW_seuillage	HW_histo	HW_grad	SA89	SA206
sola	X	X	X	X	X	
solb	X	X	X	X		X

Les solutions sola et solb ont les mêmes unités allouées, seul l'ordonnancement diffère. Dans le cas de la solution sola, une des unités exécute une tâche à une fréquence différente (plus haute dans solb que dans sola), ce qui augmente l'énergie, mais

diminue le temps. Surtout, cet ordonnancement, solb qui diffère de peu de celui de sola, explique le pic de puissance qui est nettement plus important : le fait d'exécuter une tâche à 206 MHz, au lieu de 89 MHz, fait passer le pic respectivement de 380 mW à 118 mW.

1.2.c.Effet de l'optimisation en consommation sur l'allocation avec réutilisation au maximum.

Si dans un premier temps, on sélectionne au maximum les implémentations dans la liste L_{reuse}, le choix doit se faire en réutilisant au maximum les unités présentes dans l'architecture, l'optimisation se fait une fois les unités déjà instanciées lors de l'analyse des premières tâches du DFG. L'optimisation a alors lieu pour les allocations suivantes où l'algorithme cherche à réutiliser les unités déjà instanciées. C'est le cas pour les solutions que CODEF a généré (sola et solb).

Dans cette section, le StrongARM est utilisé car nous avons choisi d'étudier l'impact du DVS/DFS dans les optimisations de l'étape d'allocation. Ainsi, nous avons renseigné la bibliothèque avec le StrongARM. Comme ce dernier est étudié pour l'économie d'énergie, et CODEF étant maintenant optimisé dans ce sens, le choix se portera sur le StrongARM plutôt que le oak pour les tâches dont l'unique allocation possible est logicielle.

A ce niveau, cette optimisation consiste à sélectionner la meilleure solution qu'offre CODEF suivant le critère consommation. En effet, lors de l'étape d'allocation/ordonnancement, on choisit les unités à allouer aux tâches restantes, parmi les unités déjà utilisées. On ne crée pas de nouvelles solutions, mais comme CODEF a pour priorité de limiter la surface en respectant la contrainte de temps (grâce à la sélection des implémentations dans les listes à prendre dans l'ordre prioritaire L^1_{reuse}, L^2_{reuse}, $L^1_{new,}$ $L^2_{new,}$ et L_{other}), les solutions générées dans ce cas correspondent à un compromis de l'ensemble des solutions de CODEF.

Dans cette optique, CODEF sélectionne la solution sola qui est de loin la plus économique en énergie et en puissance.

Cet exemple illustre que les gains en énergie obtenus sont insuffisants par cette seule technique de réutilisation. Dans la suite, elle est couplée à celle de sélection et d'introduction de l'unité la moins coûteuse en énergie: optimisation par allocation et sélection basse consommation. On affaiblit la contrainte de surface pour générer de nouvelles solutions plus économique en énergie et en puissance.

1.2.d.Résultats de l'optimisation par allocation et sélection basse consommation.

Après optimisation, nous obtenons la solution du tableau 13:

Tableau 13: Architecture optimisée pour la détection de mouvement par une caméra embarquée

solution	temps d'exécution(ms)	surface (mm^2)	Energie (nJ)	pic de puissance (mW)
solc	33.37	5.884	124 000	60.7

CODEF optimisé génère une nouvelle solution. Afin de gagner en énergie et en puissance dans cette solution, un accélérateur matériel a été ajouté. Pour la même durée d'exécution, une surface augmentée de 14% permet des gains en puissance et en énergie atteignant entre 50 et 80% par rapport, respectivement, aux solutions sola et solb.

Nous avons représenté ces trois solutions sur la figure 27 afin de mieux les comparer :

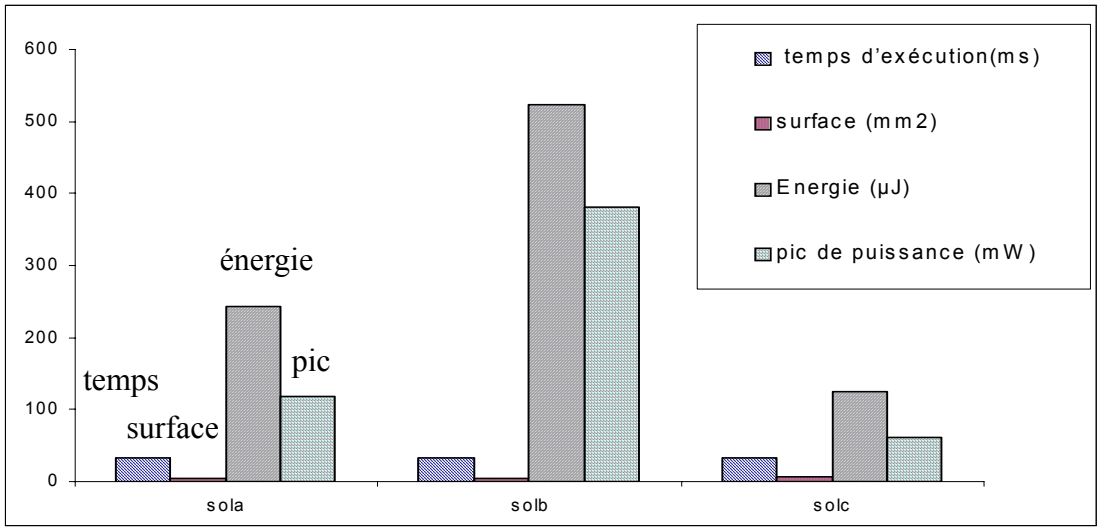


FIGURE 27. solutions avant (sola et solb) et après (solc) optimisation avec le StrongARM

Les résultats sont intéressants, mais ont nécessité d'ajouter des accélérateurs matériels qui augmentent la surface. Toutefois, cet ajout permet de fortement réduire la consommation (énergie et pic de puissance).

Nous avons décrit dans ces paragraphes les modifications apportées à CODEF pour sélectionner en priorité les unités les moins consommatrices.

Ainsi, une fois l'ordonnancement connu, nous proposons d'exploiter la technique de DVS/DFS en faisant varier les couples (fréquence, tension) du StrongARM. Ceci nécessite de déterminer une stratégie de gestion de la fréquence et de la tension.

2•Exploitation de la technique d'ajustement conjoint en tension et en fréquence

L'objet de ce paragraphe est l'étude des stratégies de raffinement d'ordonnancement. Ces raffinements sont basés sur l'exploitation de la technique d'ajustement conjoint tension/fréquence ou le *Dynamic Voltage Scaling (DVS)* et *Dynamic Frequence Scaling (DFS)*.

Comme l'indique la figure 28, notre intervention pour ajuster la fréquence et la tension se situe après l'allocation/ordonnancement et l'analyse de la consommation.

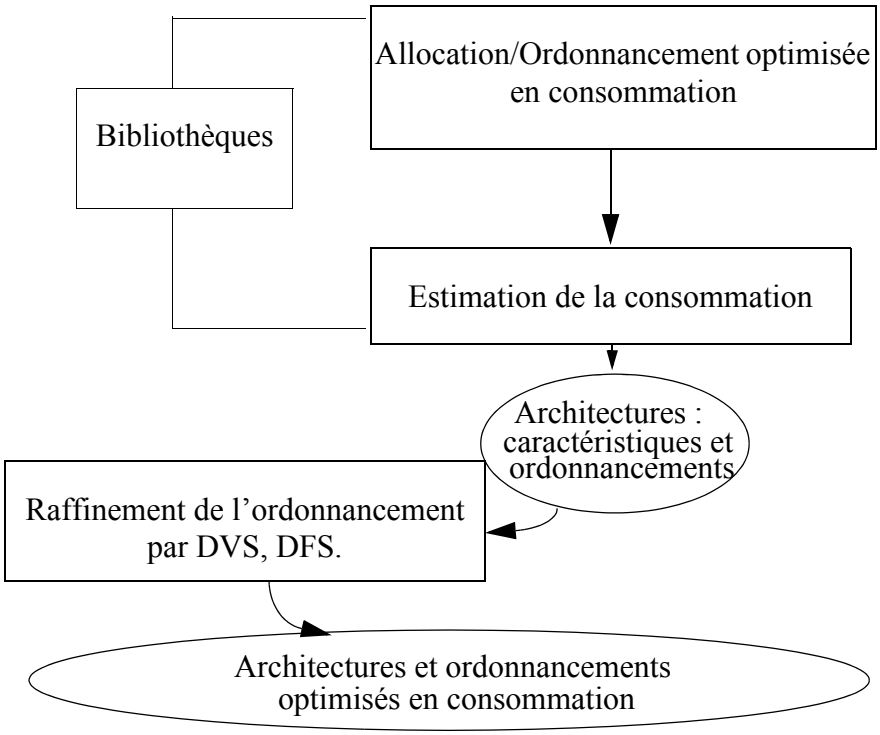
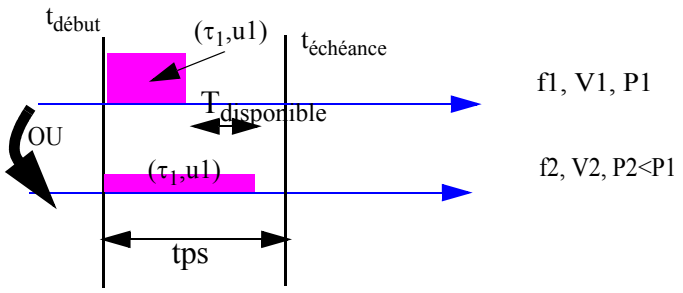


FIGURE 28. Place du raffinement par DVS/DFS dans la méthode

Afin de tenir compte des architectures réelles, notre bibliothèque permet de déclarer des unités munies ou non du DVS. Rappelons brièvement sur la figure 29 la technique d'ajustement conjoint en tension et en fréquence.

FIGURE 29. Ajustement conjoint en tension et en fréquence (Dynamic Voltage Scaling)



Rappelons que les unités munies du DVS, ont la possibilité de changer de fréquence en prenant des valeurs pré-déterminées. Autrement dit, l'ensemble des couples (tension/fréquences) possibles est un ensemble discret.

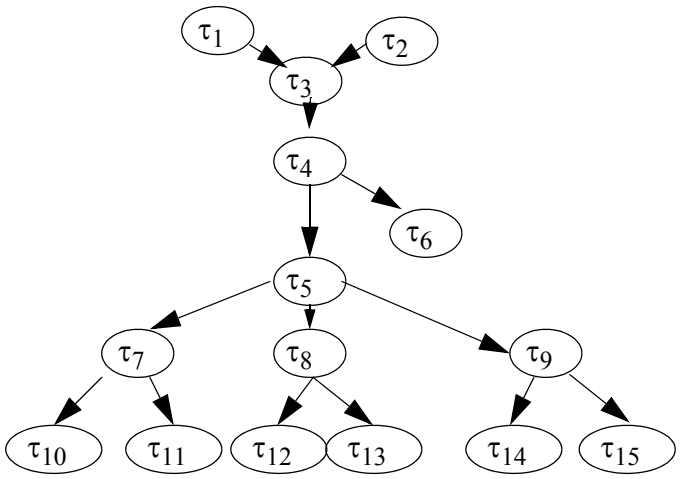
Il s'agit de réduire autant que faire se peut et de manière conjointe, la fréquence et la tension des unités qui ont cette capacité en suivant la loi [LJ00] : $T_{min} = \frac{1}{f_{max}} = \frac{k \times V_{dd}}{(V_{dd}-V_{seuil})^2}$ qui montre le lien entre la période minimum T_{min} de l'horloge et la tension d'alimentation.

Sur la figure 29, considérons la tâche τ_1 exécutée par l'unité u_1. Cette tâche τ_1 est exécutée à la fréquence f_1, et la tension V_1. Il reste du temps libre, noté $T_{disponible}$, jusqu'à l'échéance ou la date définie par la dépendance de τ_1 avec ces successeurs. Il est donc possible de diminuer la fréquence et la tension, donc de ralentir l'exécution de la tâche, afin de gagner en énergie et en puissance. Pour déterminer la nouvelle fréquence f_2 à laquelle l'unité peut exécuter la tâche τ_1, on considère le nouveau temps imparti à la tâche, tps = (temps d'exécution à la fréquence f_1 + temps libre $T_{disponible}$). Ce temps tps est divisé par le nombre de cycles n d'exécution de la tâche, afin de calculer la période du processeur. L'inverse de cette période est approximée à la fréquence disponible la plus proche qui conduit au temps tps : $f_2 \leq \frac{n}{tps}$. Ainsi, en baissant la fréquence f_1 à f_2 et en baissant la tension V_1 à V_2 où V_2 est la tension correspondant à la fréquence f_2, on réalise un gain en puissance de $\frac{f_2 \times V_2^2}{f_1 \times V_1^2}$ et en énergie de $\frac{V_2^2}{V_1^2}$.

Faisons en premier une étude prospective afin de déterminer les possibilités de gains en énergie et en puissance.

Nous nous sommes donnés un cas d'étude illustré par le DFG de la figure 30.

FIGURE 30. Cas d'étude 1



Il s'agit d'une application fictive de 15 tâches dont l'échéance sur les tâches terminales est 2500 µs. Voyons maintenant les solutions proposées par CODEF et les optimisations obtenues avec l'ajustement local en tension, fréquence.

Nous considérons que chaque tâche à une fréquence donnée a une puissance moyenne constante pour chaque implémentation (unité sur laquelle elle est exécutée et chaque fréquence à laquelle elle peut être exécutée).

L'objectif de notre étude est d'étudier les possibilités de gains en énergie et en puissance.

Tableau 14: solutions initiales

solution	Aire (mm^2)	temps d'exécution µs	P (mW)	E (nJ)
sol0_A	5.021	1630	244	977 759
sol0_B	5.021	1525	308	1 154 272

En faisant varier diverses échéances locales à des tâches, nous avons cherché quels types de solutions pouvaient être obtenues. Les tâches sont exécutées à des fré-

quences différentes par le StrongARM. Ces diverses fréquences sont indiquées dans la colonne de droite du tableau 15:

Tableau 15: Caractéristiques des solutions optimisées par ajustement local manuellement du cas d'étude 1

solution	Surface (mm^2)	temps d'exécution μ s	E (nJ)	P (mW)	Fréquences des tâches (MHz)
sol0_A_A	5.021	2499	727 109	381	59, 89, 148, 177
sol0_B_A	5.021	2453	755 515	310	59, 89, 148, 192
sol0_B_B	5.021	1619	1 060 927	381	89, 118, 148, 192, 206
sol0_B_C	5.021	1979	912 761	390	59, 89, 118, 148, 192, 206
sol0_B_D	5.021	1979	950 282	414	59, 89, 118, 177, 192, 206

Pour avoir une vue d'ensemble, représentons les résultats sous forme graphique (figure 31) :

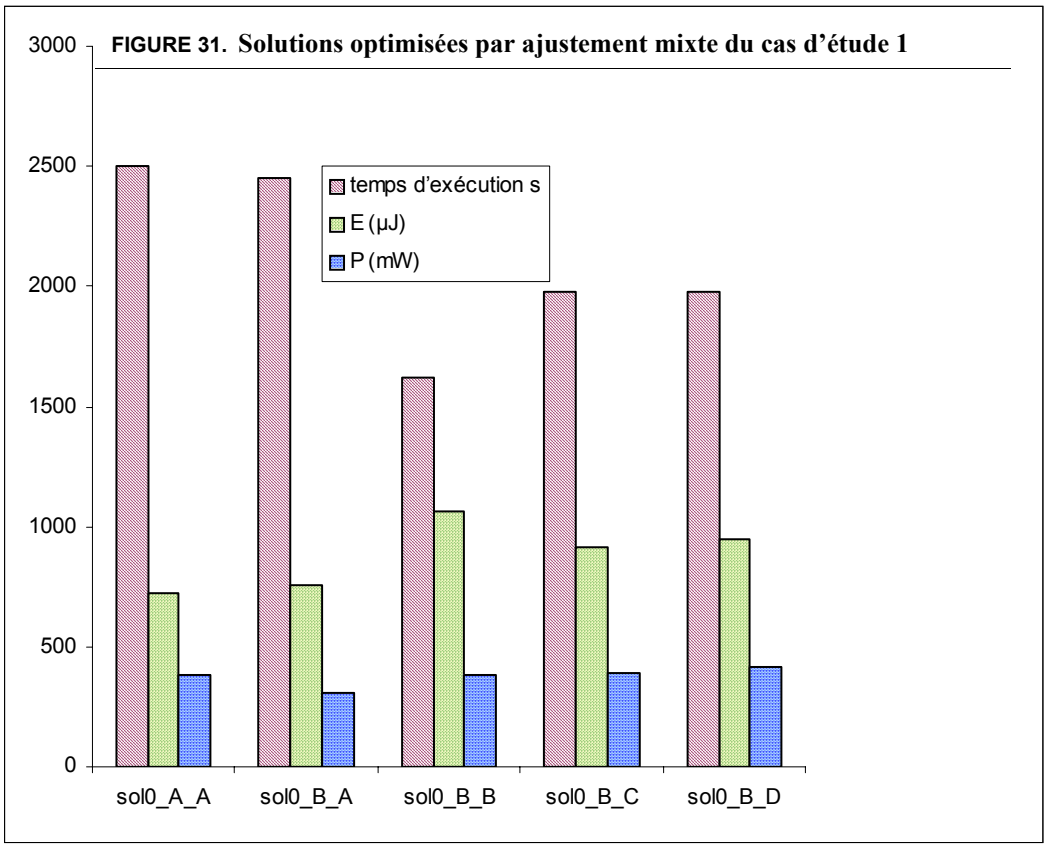


FIGURE 31. Solutions optimisées par ajustement mixte du cas d'étude 1

Par rapport aux solutions de départ non optimisées, sol0_A et sol0_B, on obtient les gains en énergie et en puissance présentés dans le tableau 16:

Tableau 16: Gains en énergie et en puissance

solution	gain en énergie	gain en puissance
sol0_A_A	26%	-35%(augmentation)
sol0_B_A	34,5%	~0%
sol0_B_B	8%	-23%(augmentation)
sol0_B_C	21%	-26%(augmentation)
sol0_B_D	18%	-34% (augmentation)

Depuis un ordonnancement initial, il est possible de repérer les tâches qu'il est judicieux de ralentir par ajustement local, en affinant au niveau des échéances des tâches. Puis, en ajustant conjointement en tension et en fréquence, des solutions différentes sont obtenues. Ainsi, on obtient une solution, sol0_B_C, dont la durée d'exécution respecte l'échéance, permettant un gain en énergie de 21% mais en augmentant le pic de puissance.

La solution sol0_B_B est aussi intéressante, puisque la durée n'est augmentée que de 6% pour un gain de 8% en énergie.

Certes, le gain en énergie est moins important que pour la solution sol0_B_C, mais la durée d'exécution est peu modifiée.

Pour ces dernières solutions, le pic de puissance a été augmenté en raison d'un recouvrement modifié des exécutions des tâches.

De même, pour la solution sol0_B_D, la puissance augmente, car, en permettant à des tâches d'être ralenties, certaines se retrouvent exécutées en même temps que la tâche 7 exécutée par un accélérateur matériel, ce qui n'était pas le cas auparavant.

Ainsi, le recouvrement modifié des exécutions des tâches permet d'optimiser en énergie, mais pas toujours en puissance. Toutefois, sur notre exemple comportant 15 tâches, les résultats sont encourageants : pour une surface inchangée et des durées d'exécutions respectant les contraintes, ces solutions permettent de gagner en énergie.

Partant de cette étude, nous formalisons ce type d'approche dans la suite afin de déterminer des méthodes systématiques qui sont présentées ci-après. D'une part, nous chercherons à optimiser en énergie en ajustant localement les tâches. D'autre part, il sera nécessaire de diminuer les pics de puissance.

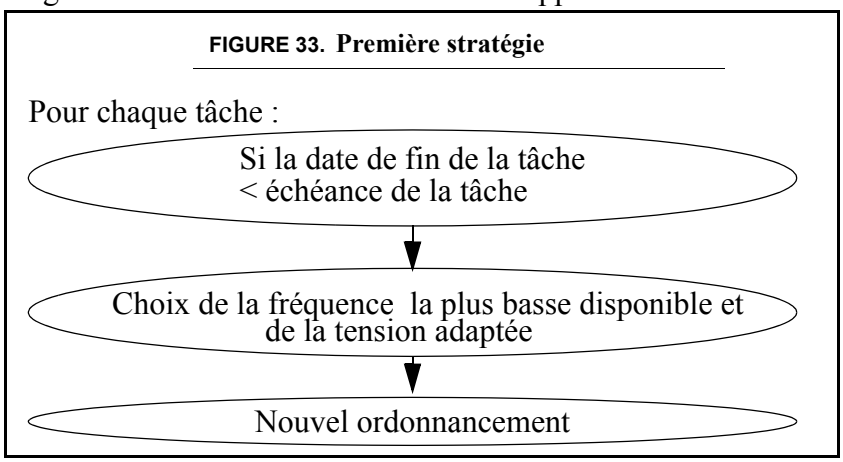
2.1.Ajustement en tension/fréquence local à une tâche.

2.1.a.Présentation de l'ajustement local en tension/fréquence local.

Cette première approche consiste à ralentir localement le processeur exécutant une tâche quand celle-ci a un temps d'exécution effectif inférieure à son échéance : l'algorithme individualise la vitesse en fonction de chaque tâche. Ce ralentissement permet de baisser la tension, diminuant du même coup l'énergie.

Sur la figure 33 est donné un résumé de cette approche :

FIGURE 33. Première stratégie



Les échéances des tâches sont soit déterminées par rapport au chemin critique, soit spécifiées par l'utilisateur.

Comment est déterminée l'échéance lorsqu'elle n'est pas spécifiée par l'utilisateur ?

Elle est déterminée de manière à respecter les dépendances de données, et à exploiter au maximum la mobilité de la tâche. Définissons cette mobilité : il s'agit de la différence entre la date la plus tardive à laquelle elle doit être exécutée, ALAP (As Last As Possible) et la date au plus tôt à laquelle elle peut être exécutée, ASAP (As Soon As Possible), calculées toutes les deux à la fréquence normale.

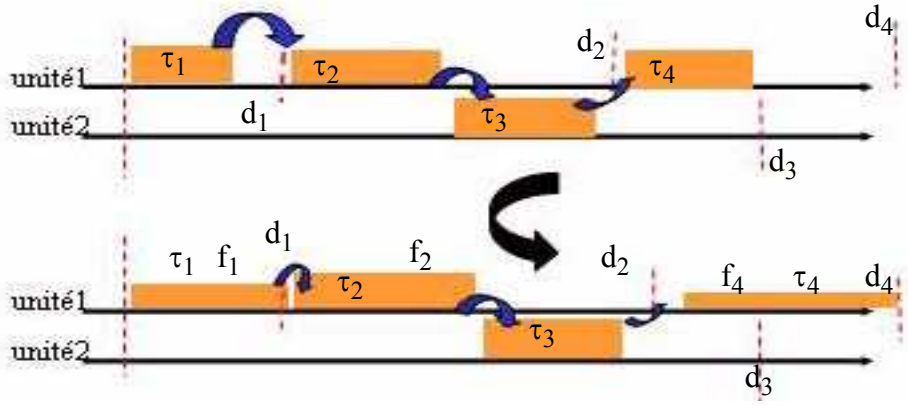
Ainsi, l'algorithme a pour objectif de déterminer les échéances pour chaque tâche et à associer une tension et une fréquence qui minimise l'énergie dans le respect de ces échéances :

- calcul de son échéance, de sa mobilité,
- recherche de la tâche la plus «rentable» à ralentir : celle pour laquelle, le plus important gain énergétique est obtenu, et si il y a égalité entre les énergies de plusieurs tâches, la sélection de la tâche se fait en fonction de la mobilité croissante. A mobilité et énergie égales, la tâche sélectionnée est la plus tardive dans l'ordonnancement (celle à exécuter le plus tard),
- modification de la tension et de la fréquence de cette tâche,
- mise à jour de l'ordonnancement.

Ces quatre étapes sont itérées avec toutes les tâches ayant une mobilité exploitable en fonction des vitesses de fonctionnement possible des unités.

Pour avoir une vue globale de l'impact de ce type de raffinement, étudions son exploitation sur un exemple (figure 34) :

FIGURE 34. Principe de la première stratégie



Les dépendances de données sont symbolisées par les flèches entre les tâches et les échéances des tâches sont fixées par les valeurs d_i.

Dans ce schéma, deux unités, unité1 et unité2, exécutent les tâches τ_1, τ_2, τ_4 et τ_3 d'une application. τ_1 et τ_4 ont du temps disponible jusqu'à leur échéance. En revanche, τ_2 est contraint en raison de τ_3 qui fournit ses résultats à τ_4. On calcule donc les nouvelles fréquences possibles d'exécution en fonction du temps disponible pour τ_1, τ_2, et τ_4. Après optimisation, les tâches τ_1, τ_2, et τ_4 sont exécutées chacune à une fréquence permettant d'aller au plus près de leurs échéances respectives d_1, d_2, et d_4, tout en respectant la dépendance de données de τ_3. Ici, les échéances sont données, si elles n'étaient pas indiquées, elles seraient déterminées à partir des mobilités, comme indiqué précédemment.

L'utilisateur peut modifier les échéances des tâches afin d'exploiter au maximum le temps globalement disponible. Cela permet de retoucher et d'affiner «à la main», afin de tester diverses configurations. Ces «retouches» peuvent s'avérer fort utiles.

Et les pénalités, pourrait-on objecter ?

Rappelons que lors des changements de <fréquence,tension>, la stabilisation de la PLL entraîne des surcoûts en temps et en consommation. L'intervalle de temps disponible à considérer doit être réduit du temps dû à cette stabilisation et un surcoût en énergie doit être compté. Ainsi, afin de prendre en compte les pénalités en temps et en consommation, nous proposons de considérer des intervalles disponibles minorés d'un temps égal au temps nécessaire au changement de <fréquence,tension>, et d'ajouter un coût moyen en énergie pour chaque changement. Dans les faits, ces

pénalités sont connues a priori en raison de l'ensemble discret des couples <fréquence,tension> disponible pour un processeur.

Toutefois, pour les exemples où nous nous comparons à d'autres travaux, nous avons choisi de négliger toute pénalité, de la même manière que ces derniers.

Signalons aussi que d'une année à l'autre des progrès importants sont réalisés dans les processeurs basse consommation, abaissant les puissances moyennes et les pénalités.

2.1.b.Résultats de l'algorithme d'ajustement local en tension/fréquence :

- **Cas d'étude 1 (figure 30) :**

solution	Aire (mm^2)	temps d'exécution μs	P (mW)	E (nJ)
sol0	5.021	1438.56	390	1 337 378

Cette solution est composée d'un processeur StrongARM à (206 MHz, 1.5 V) et d'un module matériel réalisant la tâche τ_7 du DFG de l'application.

Voici la solution optimisée avec l'algorithme décrit dans ce paragraphe :

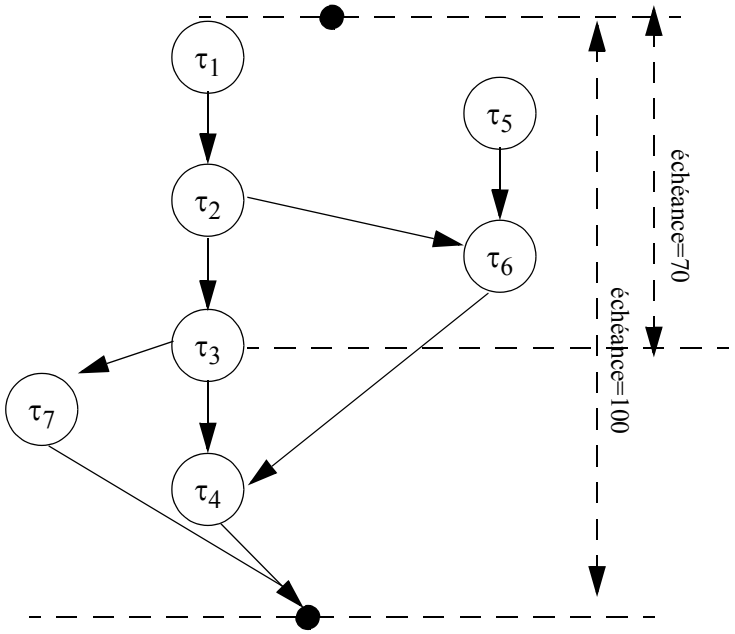
solution	Aire (mm^2)	temps d'exécution μs	P (mW)	E (nJ)
sol0raffinée1	5.021	2415	383	802 243

Dans cette solution, le StrongARM fonctionne avec les paramètres (89MHz, 0.9V), (118MHZ, 1.05V) et (206MHz, 1.5V). Ainsi, pour une surface équivalente, et une durée d'exécution respectant l'échéance, des gains de 40% en énergie et de 20% en puissance par rapport à la solution non optimisée sont obtenus.

- **Cas d'étude 2** [BG00]

Nous avons étudié l'exemple présenté dans [BG00] dont le DFG est illustré par le DFG de la figure 35.

FIGURE 35. **DFG cas d'étude 2[BG00]**



Les caractéristiques de temps d'exécution et d'énergie des implémentations des tâches, sur deux processeurs PE1 et PE2 sont donnés dans les tableaux 17 et 18 :

Tableau 17: Energie de chaque tâche sur les processeurs

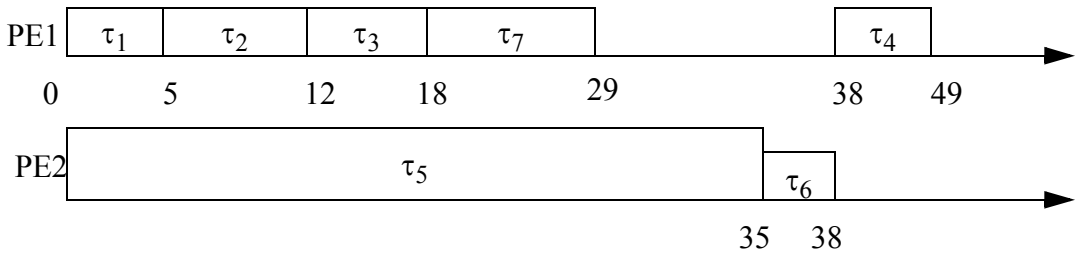
Energie / Tâche	PE1	PE2
τ_1	10	20
τ_2	15	30
τ_3	10	20
τ_4	11	7
τ_5	32	41
τ_6	7	10
τ_7	12	22

Tableau 18: Temps d'exécution de chaque tâche sur les processeurs

Temps / Tâche	PE1	PE2
τ_1	5	8
τ_2	7	12
τ_3	4	7
τ_4	8	5
τ_5	20	35
τ_6	2	3
τ_7	11	17

Soit l'ordonnancement suivant (figure 36[BG00]) : PE1 exécute τ_1, τ_2, τ_3, τ_7 et τ_4, et PE2 exécute τ_5 et τ_6.

FIGURE 36. Premier ordonnancement [BG00]

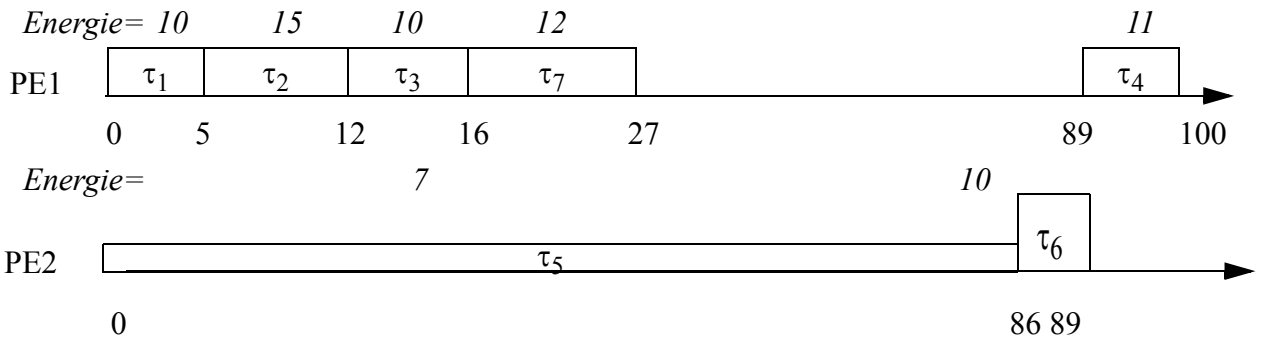


L'énergie dissipée est de 109 et le pic de puissance est de 3.33 [BG00].

Appliquons le raffinement local sur cet ordonnancement. La tâche la plus coûteuse en énergie est la tâche τ_5. Nous allons donc regarder s'il est possible de la ralentir. Calculons sa mobilité: m_{τ_5}=ALAP(τ_5)-ASAP(τ_5)=51-0=51. Les contraintes de temps permettent d'exécuter τ_5 entre les dates 0 et 86. Supposons qu'une fréquence de l'unité permette un ralentissement qui fait passer le temps d'exécution de la tâche de

35 unités à 86 unités. En faisant l'hypothèse que la tension V_{dd} n'est pas proche de la tension de seuil V_t, on peut approximer le gain en énergie par le carré du rapport entre les temps d'exécution. Dans l'exemple, on obtient un gain en énergie de $\left(\frac{86}{35}\right)^2 \sim 6$, soit une énergie de τ_5 égale à $\frac{41}{6} \sim 7$. Le nouvel ordonnancement modifié est donné sur la figure 37.

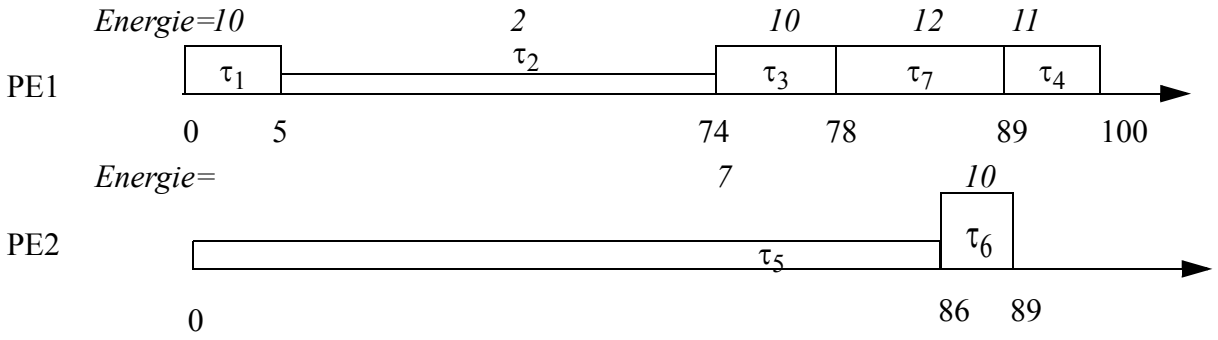
FIGURE 37. Deuxième ordonnancement de l'application [BG00]



L'énergie dissipée par ce nouvel ordonnancement est de 10 + 15 +10 +12 +11 + 7 + 10 = 65, soit un gain en énergie de 40% par rapport à la solution non optimisée. Le pic de puissance est à 3,26, soit un gain de 2%.

Une fois l'ordonnancement mis à jour, nous cherchons à ralentir une nouvelle tâche. Hormis τ_5, la tâche d'énergie la plus coûteuse à la suite est τ_2. Calculons d'abord sa mobilité: m_{T2}= ALAP(τ_2) - ASAP(τ_2)=69-5=64. Les dates de début et de fin d'exécution de τ_2 utilisant cette mobilité sont 5 et 74 respectivement. Faisons les mêmes hypothèses que précédemment, à savoir qu'il y a possibilité de réduire la vitesse de cette tâche pour durer 69 unités de temps. On obtient le nouvel ordonnancement présenté en figure 38.

FIGURE 38. Troisième ordonnancement de l'application [BG00]



L'énergie est de 58, et le pic de puissance, dû à τ_6 et τ_7 exécutées en même temps, est d'environ 4,4. Le gain en énergie est alors d'environ 47% par rapport à l'ordonnancement non optimisé (figure 36). Plus aucune tâche n'a de mobilité, le raffinement est fini. Ainsi, le nouvel ordonnancement permet un gain en énergie intéressant, mais le pic de puissance est augmenté.

Cette première méthode nous permet d'obtenir des gains significatifs en énergie, mais pas toujours en puissance. L'optimisation est faite sur les premières tâches rencontrées par l'algorithme, ce qui annule rapidement les mobilités des autres tâches. Ainsi, il peut arriver, comme dans l'exemple précédent, que des tâches soient exécutées dans le même temps après le raffinement (ce qui n'était peut-être pas le cas avant), avec une seule des contributions réduites. En définitive, la somme des puissances des deux tâches dépasse chaque puissance précédente qui était comptée séparément, le pic de puissance s'en trouve, en définitive, augmenté.

Pour pallier ces défauts, l'objectif serait de ralentir un plus grand nombre de tâches, plutôt que de ralentir au maximum quelques unes. Aussi, nous considérons dans le paragraphe suivant une approche basée sur un ajustement global en tension et en fréquence.

2.2.Ajustement en tension/fréquence de manière globale.

2.2.a.Présentation de l'Ajustement en tension/fréquence de manière globale.

La stratégie décrite dans le paragraphe précédent est locale et peut entraîner des changements de <tension/fréquence> entre deux tâches successives. Or, ces changements peuvent diminuer les bénéfices en consommation et en temps. En effet, comme nous l'avons vu dans le chapitre de l'Etat de l'Art, la stabilisation de la PLL (pour resynchroniser les signaux d'horloge) et les coûts induits par des circuits adaptant la tension peuvent être pénalisants.

Par ailleurs, elle permet de ralentir que quelques tâches en se focalisant sur les premières tâches rencontrées ayant une grande mobilité. Celles-ci sont ralenties au maximum, annulant ainsi les mobilités éventuelles d'autres tâches. Aussi, nous avons cherché à explorer d'autres voies d'optimisation qui ralentiraient un plus grand nombre de tâches. Dans la partie précédente, nous avons négligé ces coûts, car sur des applications présentant une grande mobilité des tâches et nécessitant peu de changements de tension ou de fréquence, cette approche d'optimisation locale peut être efficace.

Cette deuxième approche consiste à ralentir le processeur ou les unités et ce, de manière globale, lorsque la durée effective d'exécution de l'application est inférieure à la contrainte de temps, c'est-à-dire l'échéance de l'application. On calcule ici une vitesse de fonctionnement unique pour l'unité pendant toute la durée d'exécution de l'application.

Ainsi, chacune des unités sera globalement ralentie au mieux de ce qu'autorise les contraintes de l'application.

Dans notre approche, les unités présentes, capables de changer conjointement la tension et la fréquence, sont étudiées. Pour chacune d'elle, nous regardons s'il est possible de ralentir l'unité globalement pour toutes les tâches qu'elle exécute (elles seront donc toutes exécutées à la même fréquence). On déduit un facteur d'échelle appliqué sur la fréquence, qui est calculé en divisant la durée maximale d'exécution de l'application par la durée totale de l'application :

$$\text{facteur d'échelle} = \frac{\text{Echéance de l'application}}{\text{temps effectif de l'application}}$$

Afin de vérifier que toutes les dépendances de données permettent ce ralentissement global, nous avons défini un critère de faisabilité $\eta \leq 1$ qui doit être respecté pour chaque tâche exécutée par l'unité.

Comme précédemment, on considère que chaque unité dispose d'un ensemble discret de couples <fréquence, tension>. Pour l'unité munie du DVS considérée, ce critère $\eta \leq 1$ est calculé de manière itérative à chaque tâche exécutée par une unité munie du DVS, et ce, à chaque mise à jour de l'ordonnancement.

Ce terme est calculé de la manière suivante :

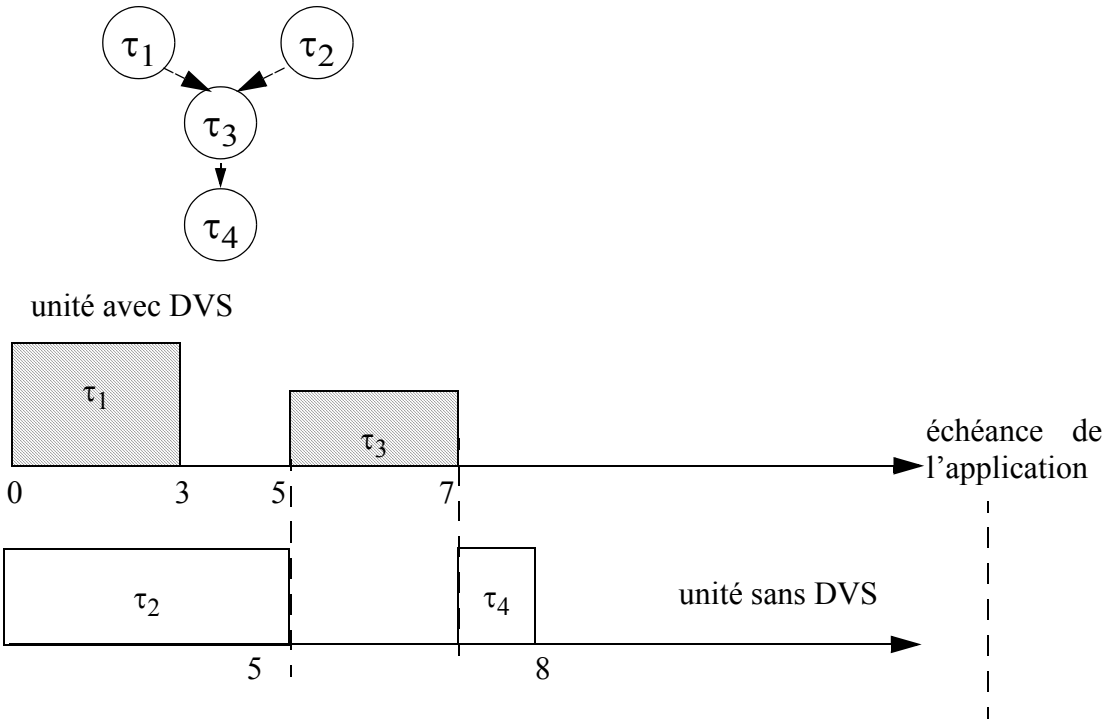
$\eta = \frac{f_s \times t_i}{m_{\tau_i}}$, avec f_s, facteur d'échelle, t_i le temps d'exécution effectif de la tâche par cette unité, m_{τ_i}, la mobilité dont disposent les tâches exécutées par l'unité.

La condition de faisabilité est $\eta \leq 1$, cette condition est à calculer pour toutes les tâches en tenant compte des modifications temporelles effectuées pour les tâches prédécesseurs.

Dans le cas où la condition n'est pas vérifiée, on déduit une nouvelle valeur de f_s qui respecte $\eta \leq 1$: $\eta = \frac{f_s \times t_i}{m_{\tau_i}} \leq 1 \Rightarrow f_s \leq \frac{m_{\tau_i}}{t_i}$.

Voyons ces principes sur un exemple (figure 39) :

FIGURE 39. Exemple avec son ordonnancement



Sur cet exemple, quatre tâches ont été ordonnancées. A partir de cet exemple, calculons le facteur d'échelle que nous devrions appliquer avec une échéance de l'application égale à 16 : f_s=16/8= 2.

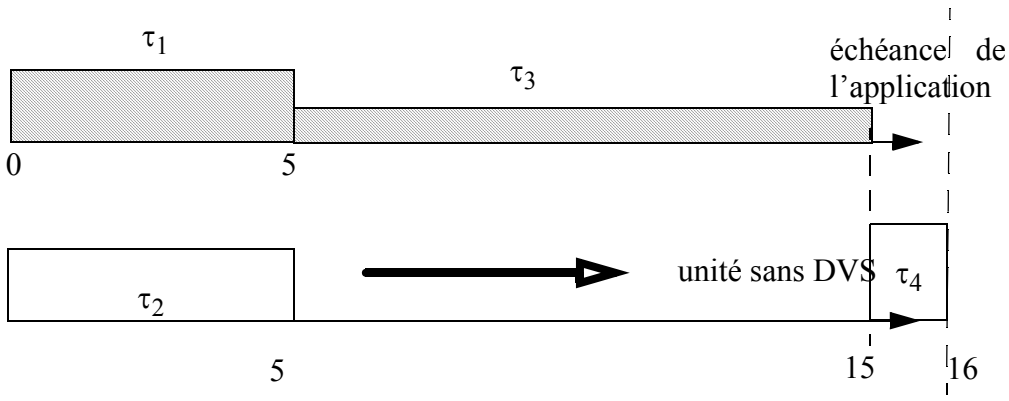
Pour la tâche τ_1 exécutée par une unité avec DVS: η_1=2*3*(1/5) =6/5 qui est supérieur à 1, donc le critère de faisabilité n'est pas respecté pour τ_1. En appliquant le critère de faisabilité, on déduit une nouvelle valeur pour le facteur de ralentissement qui est : 5/3 ~1,6. Donc, si on applique ce facteur d'échelle, les nouvelles dates de début et fin d'exécution de τ_1 seront respectivement 0 et 3*5/3= 5, au lieu de 0 et 3.

Avec l'ordonnancement mis à jour, réitérons ces étapes pour la tâche τ_3. Le facteur d'échelle est toujours de 2. Calculons le critère de faisabilité pour cette tâche, en tenant compte du décalage possible de τ_4. En effet, si τ_4 est décalée au maximum vers la date échéance au plus tard de l'application, qui est de 16, il reste un temps disponible de 10 pour τ_3 : η_3=2*2/10=4/10 qui est bien inférieur à 1.

Les nouvelles dates de début et de fin d'exécution de τ_3 sont donc respectivement 5 et 15, au lieu de 5 et 7; et les nouvelles dates de début et de fin d'exécution de τ_4 sont respectivement de 15 et 16, au lieu de 7 et 8. Les tâches sont ordonnancées au plus tôt.

Voici ci-dessous le nouvel ordonnancement (figure 40) :

FIGURE 40. Nouvel ordonnancement



Supposons que les énergies respectives des tâches 1 et 3 étaient de 3 et 4 avant le ralentissement, et soient maintenant de 1 et de 0,04 en appliquant des facteurs de $\left(\frac{5}{3}\right)^2$ et $\left(\frac{15}{2}\right)^2$. Supposons que les tâches 2 et 4 aient une énergie respectives de 3 et 0,5, on peut alors calculer les gains.

Ainsi, les échéances sont respectées, et le gain en énergie est d'environ 66% et le gain en puissance est d'environ 25% dans notre exemple.

Etudions maintenant ce même problème lorsqu'on a plusieurs unités avec DVS. En effet, quels facteurs d'échelle appliquer à toutes les unités avec DVS pour obtenir plus de rentabilité, tout en respectant les dépendances de données et les échéances.

Il s'agit d'un problème complexe pour lequel une formulation par un programme linéaire en nombres entiers est envisageable, mais permet difficilement de

traiter des DFG à plusieurs centaines de nœuds. Nous considérons donc ici aussi une heuristique.

Pour une architecture hétérogène, nous définissons un ordre de ralentissement entre les unités. En effet, les unités de l'architecture hétérogène peuvent ne pas avoir les mêmes couples <fréquences,tension> disponibles, il est alors nécessaire de ralentir chaque unité individuellement avec le couple <fréquence,tension> adéquat. La procédure est donc la suivante : on ralentit l'unité exécutant les tâches pour lesquelles un changement de fréquence et de tension a le plus d'impact, c'est-à-dire, l'unité pour laquelle, la différence entre les énergies des exécutions de ces tâches aux deux fréquences différentes est la plus grande. Soit l'unité u_1 fonctionnant aux fréquences f_1 et f_1^{new} où f_1^{new} est la plus petite fréquence conduisant à un fonctionnement correct. Soient E_1 et E_1^{new}, les énergies correspondantes. La différence d'énergie est

$\Delta_1 = E_1 - E_1^{new}$.

Prenons les mêmes notations pour l'unité u_2 avec l'indice correspondant.

L'unité à ralentir sera u_1 si $\Delta_1 > \Delta_2$, et u_2 si $\Delta_1 < \Delta_2$.

En généralisant à n unités, l'unité à ralentir en premier sera celle pour laquelle la différence d'énergie est maximale.

Après raffinement, l'ordonnancement est recalculé avec les nouvelles dates à chaque examen d'une unité capable de changer conjointement de tension et de fréquence.

Décrivons l'algorithme correspondant (figure 41).

FIGURE 41. Algorithme de la deuxième stratégie

Faire

Calcul de la durée totale d'exécution de l'application;

Si (date effective de fin d'exécution de l'application < **échéance**)

Pour chaque unité munie du DVS,

choix de l'unité DVS à ralentir prioritairement;

$$\text{facteur d'échelle} = \frac{\text{échéance de l'application}}{\text{temps effectif de l'application}};$$

Si Condition de faisabilité non vérifiée: $\eta \geq 1$

Calcul d'un nouveau facteur: $\eta = \dfrac{f_s \times t_i}{m_{\tau_i}} \leq 1 \Rightarrow f_s \leq \dfrac{m_{\tau_i}}{t_i}$

Fin Si

Ralentissement de cette unité; mise à jour de l'ordonnancement avec ordonnancement de toutes le tâches au plus tôt (ASAP);

Fin Pour

Fin Si

Fin

2.2.b.Résultats de l'ajustement global en tension/fréquence.

- **<u>Cas d'étude 1</u>**

Cette application est présentée au paragraphe précédent (figure 30).

On rappelle la solution obtenue avant ce raffinement global:

Tableau 19: Solutions avant optimisation

solution	Aire (mm^2)	temps d'exécution μs	P (mW)	E (nJ)
sol0	5.021	1438.56	390	1 337 378

Après raffinement global, nous obtenons:

Tableau 20: Solution optimisée suivant l'approche globale

solution	Aire (mm^2)	temps d'exécution μs	P (mW)	E (nJ)
sol0raffinée2	5.021	2316	125	662 409

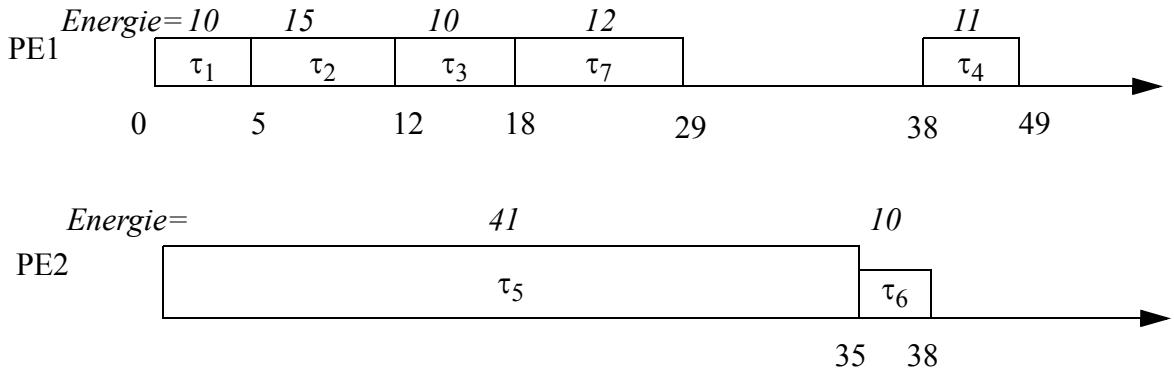
La solution sol0raffinée2 correspond à la solution comportant un StrongARM et un module matériel pour la tâche numéro 7. Les unités présentes sont les mêmes que pour sol0. Ici, le StrongARM exécute toutes ses tâches à (118 MHZ, 1.05V) au lieu de (206 MHz, 1.5V) avant raffinement. Ainsi, le gain en énergie est de 51%, le gain en puissance est de 65%, pour une durée respectant l'échéance.

- **Cas d'étude 2** [BG00]:

Nous avons étudié l'exemple présenté dans [BG00]. La description en est donnée figure 35 et dans les tableaux 17 et 18. Nous considérons que l'architecture qui réalise l'exécution est hétérogène afin de se placer dans le cas général.

Soit l'ordonnancement non optimisé suivant : PE1 exécute τ_1, τ_2, τ_3, τ_7 et τ_4, et PE2 exécute τ_5 et τ_6.

FIGURE 42. Ordonnancement initial [BG00]

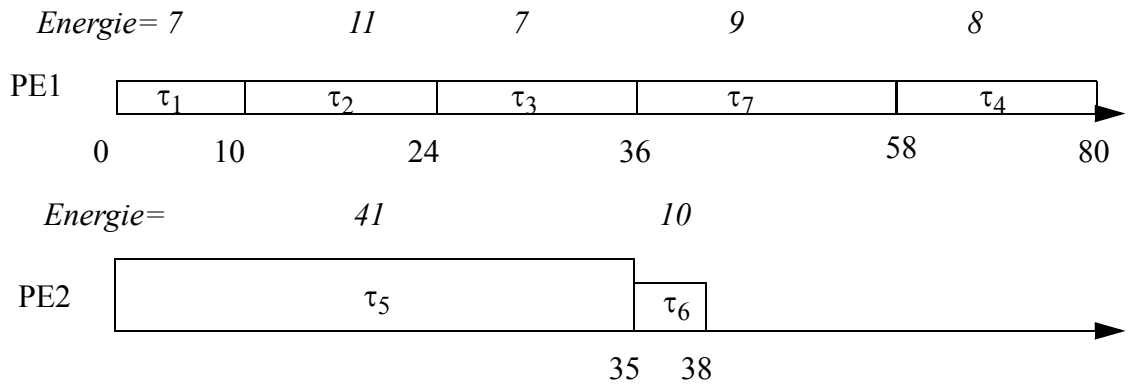


L'énergie est de 109 unités et le pic de puissance est de 3,33 (dû à τ_6), d'après [BG00]. Appliquons l'algorithme :

- Considérons d'abord que PE1 possède une unité DVS, contrairement à PE2. Le facteur d'échelle correspondant est f_s=100/49=2,04, du fait que la contrainte de temps est de 100 et la date effective de fin d'exécution est de 49.

Après vérification du respect du critère de faisabilité, la nouvelle solution obtenue est présentée à la figure 43.

FIGURE 43. Premier ordonnancement obtenu par optimisation du facteur global

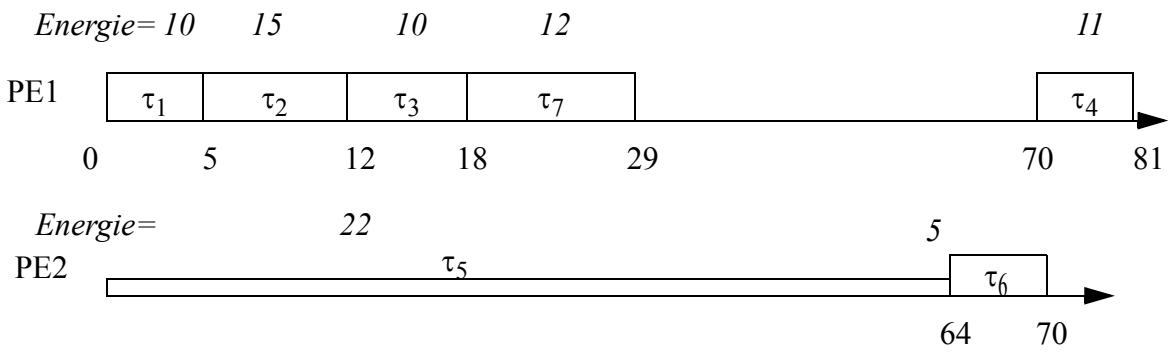


τ_4 est à la suite directement de τ_7 car elle est ordonnancée au plus tôt afin d'éliminer au maximum le temps libre. Avec ce nouvel ordonnancement, l'échéance est respectée et l'énergie atteint 93, ce qui représente un gain de 15 % par rapport à la solution non optimisée de départ [BG00].

Le pic de puissance passe de 3,33 avant optimisation à 3,9 après optimisation. Cette augmentation du pic est due au recouvrement modifié des tâches. En effet, dans cet ordonnancement, le pic est dû aux exécutions simultanées des tâches τ_6 et τ_3. Précédemment, la tâche τ_6 était exécutée seule.

- Considérons maintenant que PE2 est une unité avec DVS, et PE1, une unité sans DVS. Le facteur d'échelle correspondant est f_s=70/38=1,84, car l'échéance de τ_6 est de 70. Après vérification du respect du critère de faisabilité, le nouvel ordonnancement est:

FIGURE 44. Ordonnancement avec raffinement par facteur global de 1,84 sur PE2



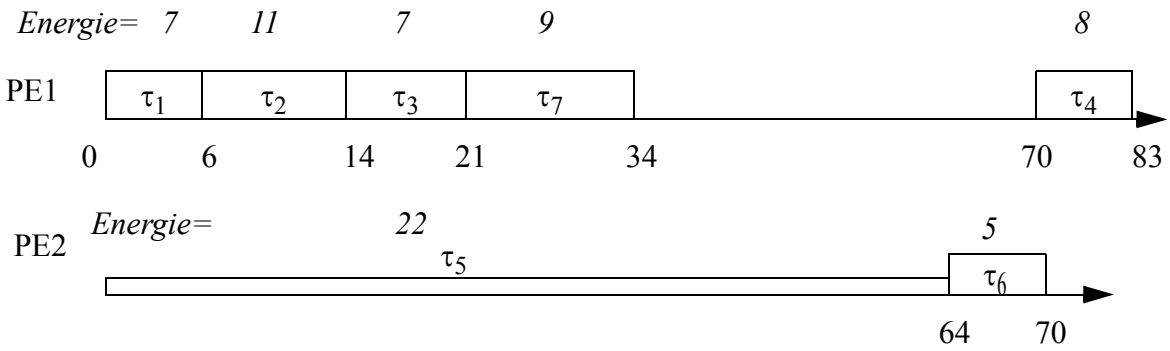
Pour ce nouvel ordonnancement, l'échéance est encore respectée, et on obtient une énergie de 85, soit un gain de 22% par rapport à la solution non optimisée. Dans cette nouvelle optimisation, le pic de puissance passe de 3,33 avant optimisation, à 0,83, le gain est de 75% par rapport à la solution non optimisée. Le gain important en puissance est dû à τ_6 qui a été ralentie d'un facteur 1,84.

- Considérons maintenant que PE1 et PE2 sont des unités avec DVS.

Nous allons d'abord chercher l'unité à ralentir prioritairement. Nous prendrons celle dont la différence d'énergie entre les deux réalisations des tâches aux fréquences différentes sera la plus grande. Pour PE1, la différence est de 16, et pour PE2, la différence est de 24. Nous ralentissons donc PE2 (figure 45).

Appliquons la méthode au processeur PE1 avec ce nouvel ordonnancement. Le nouveau facteur de ralentissement est de 100/81=1,23.

FIGURE 45. Ordonnancement après ralentissement avec le facteur 1,23 sur PE1



L'énergie totale est de 69 unités, ce qui représente un gain de 37% par rapport à la solution non optimisée. Quant au pic de puissance, il atteint maintenant 1,72, soit un gain de 48% par rapport à la solution non optimisée.

Le temps disponible ne permet pas de poursuivre encore les optimisations avec cette méthode.

- **Cas d'étude 3: exemple de COSYN-LP**[BL97]

Soit le graphe de l'application de la figure 46 avec les caractéristiques d'énergie et de temps d'exécution données dans le tableau 21 :

FIGURE 46. DFG exemple de COSYN-LP

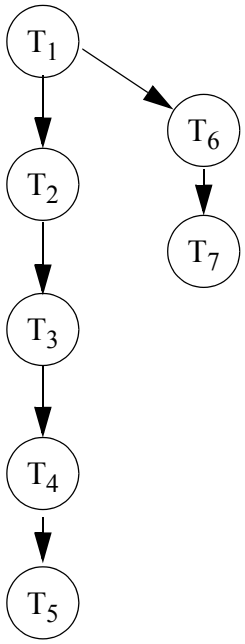
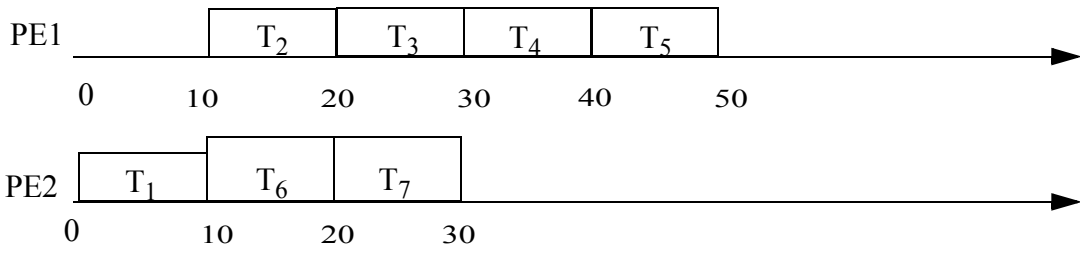


Tableau 21: Caractéristiques de l'application testée dans [BL97]

Tâches	Puissance	Temps	Puissance à la nouvelle fréquence (f_s=2)	Puissance à la nouvelle fréquence (fs=1.67)
T_1	0.5	10	0.1	0.2
T_2	0.5	10	0.1	0.2
T_3	0.5	10	0.1	0.2
T_4	0.5	10	0.1	0.2
T_5	0.5	10	0.1	0.2
T_6	1.5	10	0.4	0.7
T_7	1.5	10	0.4	0.7

Dans [BL97], un ordonnancement établi par COSYN-LP est proposé :

FIGURE 47. Ordonnacement initial [BL97]



Il est donc optimisé en consommation, nous allons affiner cette optimisation.

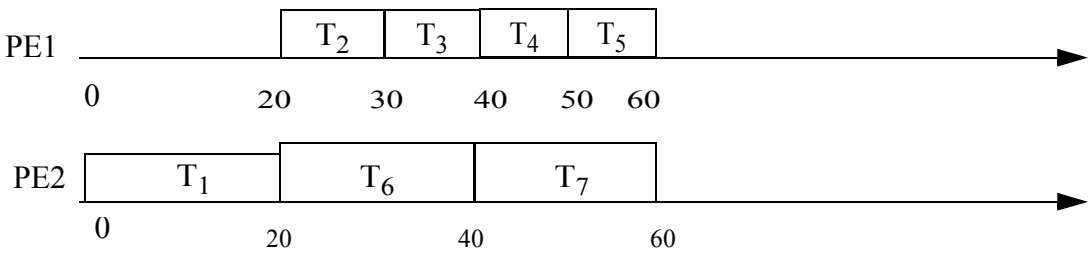
L'échéance de l'application (100) n'est pas atteinte, aussi appliquons notre approche d'ajustement global. D'après l'article, les deux unités instanciées sont identiques. Cet ordonnancement correspond à une énergie de 55 et un pic de puissance de 2 (dû aux exécutions en parallèle de T_7 et T_3 d'une part, et T_2 et T_6 d'autre part). Voyons maintenant les possibilités d'ajustement.

En procédant à l'ajustement en fréquence de PE1, le facteur d'échelle calculé est de 2. On obtient une différence d'énergie entre les deux exécutions aux deux fréquences différentes de: $\Delta_1 = 20 - 8 = 12$.

Considérons le cas de l'unité PE2, avec le même facteur d'ajustement, 2. La différence d'énergie est alors de $\Delta = 35 - 18 = 17$, ce qui est plus que pour la première unité. Appliquons donc l'ajustement en fréquence à la deuxième unité.

En appliquant l'algorithme, on trouve que la condition de faisabilité est vérifiée pour chaque tâche. Le nouvel ordonnancement avec fs=2 pour PE2 est donné figure 48.

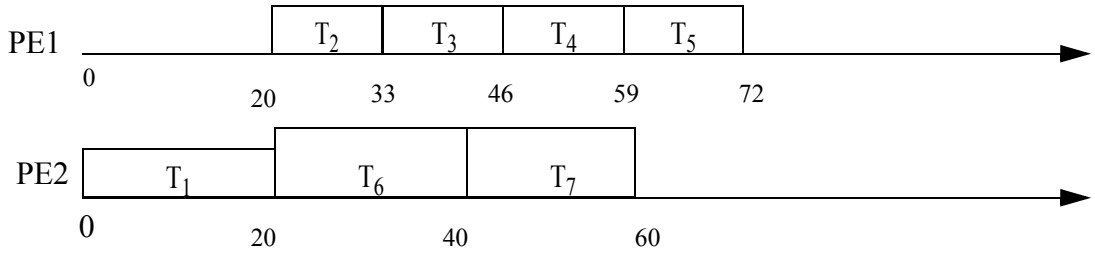
FIGURE 48. Ordonnancement de l'exemple de [BL97] après ajustement de l'unité 2 avec la facteur 2



Cette solution correspond à une énergie de 38 et un pic de puissance de 1,3, soit des gains respectifs de 31% et de 35% par rapport à la solution non optimisée.

Essayons maintenant d'ajuster la fréquence de PE1 à partir de cet ordonnancement. Le facteur d'échelle f_s est égal à 100/60=1,67 et le critère de faisabilité est respecté. Ainsi, la solution finale est illustrée à la figure 49 :

FIGURE 49. Ordonnancement de l'exemple de [BL97] après ralentissement de PE1 avec le facteur 1,67



Avec ce dernier ordonnancement, l'énergie atteint 31 et le pic de puissance atteint 0,8, les gains en énergie et en puissance sont respectivement de 43% et de 60%.

Les gains sont intéressants, en particulier du fait de l'absence de changement dynamique de tension ou de fréquence. Cependant, cet exemple illustre qu'il est possible d'effectuer des optimisations supplémentaires puisqu'il reste du temps jusqu'à l'échéance. Afin d'optimiser de manière plus poussée, nous présentons dans la suite une troisième stratégie où le ralentissement se fait d'abord de manière globale, puis est affiné avec le premier algorithme.

2.3.Ajustement mixte : Combinaison des deux stratégies précédentes.

2.3.a.Présentation de l'ajustement mixte.

Comme nous l'avons vu, la première stratégie permet d'avoir des raffinements précis, mais peut entraîner de nombreuses pénalités et surtout se focalise sur l'ajustement de quelques tâches (ce qui rend rapidement nulle la mobilité des autres tâches). La deuxième stratégie s'est affranchie de ces pénalités. Ainsi, nous avons étudié une troisième voie d'optimisation tendant à regrouper les avantages des deux stratégies précédentes en couplant ces deux dernières méthodes. La combinaison consiste donc à raffiner une première fois de manière grossière avec la deuxième approche qui opère de manière globale, puis avec le raffinement local.

2.3.b.Résultats de la stratégie mixte.

- **Cas d'étude 2** [BG00]

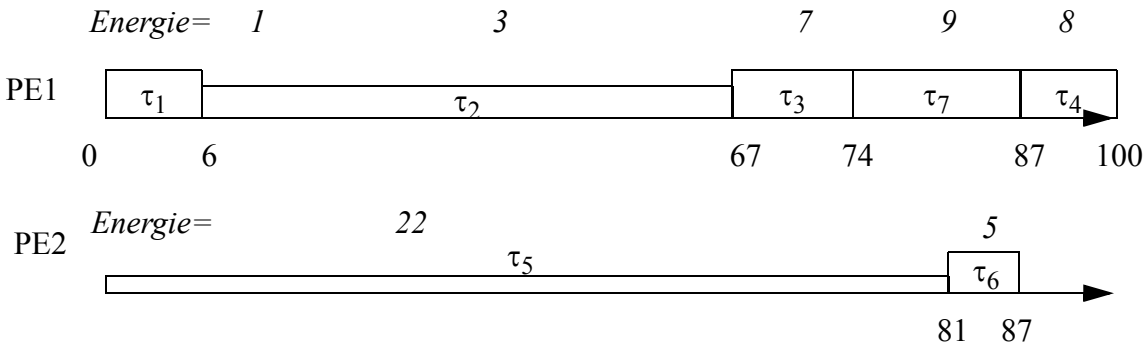
Dans cet exemple, décrit à la figure 35, deux échéances sont données : la première est l'échéance de la tâche τ_3 à 70, et la deuxième, 100, est globale à l'application.

Après avoir appliqué l'ajustement global sur les deux unités (figure 45), appliquons l'ajustement local.

L'énergie totale est de l'ordre de 69 unités, ce qui représente un gain d'environ 37% par rapport à la solution non optimisée. Quant au pic de puissance, il atteint 1,72, soit un gain de 48% par rapport à la solution non optimisée.

Appliquons maintenant la stratégie de ralentissement local, ralentissons tâche par tâche (figure50) :

FIGURE 50. Ordonnancement issu de la stratégie mixte



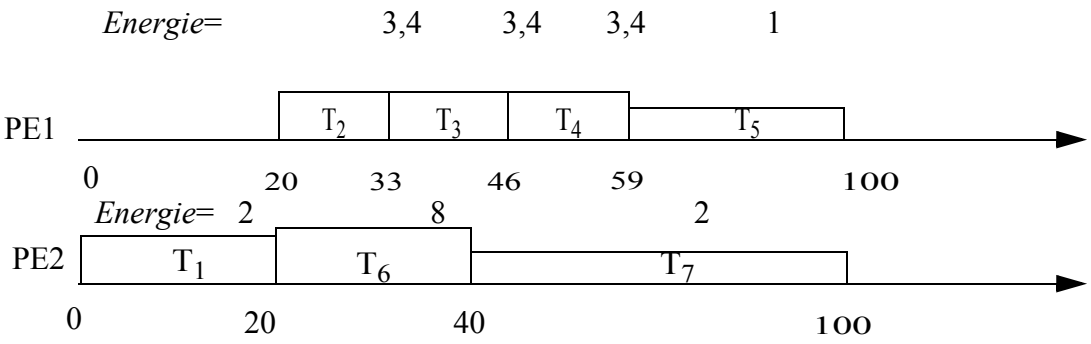
On obtient une énergie globale de 55 et un pic de puissance de 1,52, soit des gains en énergie et en puissance de 50% par rapport à la solution non optimisée. Rappelons que le gain en énergie était de 40% (et 2% de gain en puissance) pour la solution après raffinement par la stratégie d'ajustement local par rapport à la solution non optimisée, et de 37% (et 75% de gain en puissance) pour la solution après raffinement par la stratégie d'ajustement global par rapport à la solution non optimisée.

Avec cette dernière stratégie, les résultats sont meilleurs.

- **Cas d'étude 3: exemple de COSYN-LP** [BL97]

Après les optimisations effectuées précédemment (cf. paragraphe 2.2.b), passons maintenant à la combinaison des deux stratégies. Partons de l'ordonnancement de la figure 49 où les tâches T_2, T_3, T_4, et T_5 ont une mobilité de 28 et les tâches T_1, T_6 et T_7 ont une mobilité de 40. T_1 n'est pas retenue car elle est moins coûteuse en énergie que T_6 et T_5. En effet, à mobilité égale, l'algorithme sélectionne prioritairement les tâches les plus coûteuses en énergie. Ici, les énergies de T_6 et T_7 sont égales, il en est de même pour T_2, T_3, T_4, et T_5. L'algorithme est appliqué, on obtient alors l'ordonnancement donné à la figure 52, en supposant qu'il existe une fréquence à ces unités permettant de ralentir les tâches T_5 et T_7 jusqu'à leurs échéances (figure 52).

FIGURE 52. Ordonnancement après optimisation par ajustement mixte du cas d'étude 3 [BL97]



Dans cette solution, l'énergie obtenue est de 23,2, soit un gain en énergie de 58% par rapport à la solution optimsée de départ donnée par COSYN-LP. Le gain en puissance par rapport à la solution avant notre optimisation est de 60% par rapport à la solution de départ trouvée par COSYN-LP. Rappelons que le gain en énergie était de 43% (et 60% de gain en puissance) pour la solution après raffinement par la stratégie d'ajustement global par rapport à la solution non optimisée.

Ces résultats sont intéressants, et la stratégie mixte donne de meilleurs résultats que les deux autres stratégies développées précédemment.

Ainsi, dans le paragraphe suivant, nous analysons plus finement ces résultats.

2.4.Analyses des résultats de raffinement d'ordonnancement par ajustement conjoint en tension et en fréquence.

La première stratégie (l'ajustement local) permet des gains en énergie et en puissance réduits en raison de sa focalisation sur les premières tâches rencontrées par l'heuristique. Ainsi, les premières tâches ralenties diminuent ensuite les mobilités des autres tâches. Peu de tâches peuvent être ralenties avec cette approche, mais celles qui le sont, le sont fortement. L'énergie est diminuée mais la puissance ne l'est pas toujours, en raison du recouvrement modifié des tâches. En effet, le fait de ralentir une tâche augmente la probabilité que les tâches soient en recouvrement. Ce recouvrement modifié a souvent pour effet d'augmenter le pic de puissance.

La deuxième stratégie (ajustement global) permet des ralentissements d'un plus grand nombre de tâches, optimisant ainsi d'une manière plus «équilibrée». En général, il est encore possible de ralentir individuellement des tâches, car il peut rester du temps libre avant l'échéance des tâches.

La troisième stratégie allie les avantages de ces deux stratégies et réduit les inconvénients de la première réalisée seule. En effet, en appliquant d'abord un ralentissement global, puis un ralentissement individuel des tâches, il y a moins de changements dynamiques des couples <fréquence,tension> et les recouvrements de tâches modifiés le sont avec des tâches ralenties, diminuant ainsi le pic de puissance. C'est avec cette dernière méthode que les meilleurs gains en énergie et en puissance sont obtenus.

Lorsque les optimisations par ajustement conjoint en tension et en fréquence ont été réalisées, il peut être encore possible de réduire la puissance et l'énergie grâce à la gestion des modes basse consommation des unités de l'architecture. En effet, il se peut que le temps disponible des tâches ne permette pas de choisir parmi les valeurs discrètes <fréquences,tension> dont dispose l'unité. De plus, l'étude du comportement de décharge des batteries a montré que les moments de repos du système peuvent être bénéfiques et permettre à la batterie de récupérer une partie de sa capacité énergétique [DN95,MS99]. Etudions ce type d'optimisation dans le paragraphe suivant.

3•Gestion des modes basse consommation

3.1.Principe.

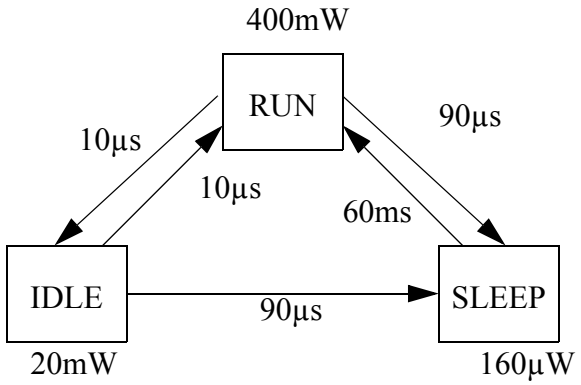
Les processeurs actuels ont généralement au moins trois modes de fonctionnement :

- mode de fonctionnement normal ou mode *run*,
- mode repos, *idle,* mode pour lequel quelques périphériques restent alimentés de manière à revenir rapidement au mode de fonctionnement normal,

- mode repos plus profond, *sleep,* mode pour lequel moins de périphériques sont alimentés par rapport au mode repos *idle.* La consommation est donc encore diminuée.

Pour exemple, voici les modes du StrongARM[1] avec les temps de changement de mode et les coûts en puissance associés (figure 53).

FIGURE 53. Modes basse consommation du StrongARM



Cherchons à partir de quelle durée d'inactivité il est rentable de mettre chaque processeur dans un de ces modes. En effet, comme indiqué dans la figure 53, le passage du mode *sleep* au mode fonctionnement implique une pénalité en temps et en consommation qu'on ne peut ignorer. Par exemple, la pénalité en temps pour le changement du mode *run* au mode *idle* et vice-versa est de 10µs pour le StrongARM.

Un autre élément intervenant ici est le comportement des batteries. En effet, lors de périodes de repos des processeurs, les batteries récupèrent une petite partie de leur capacité énergétique, augmentant ainsi leur durée de vie. Il peut donc s'avérer intéressant de favoriser des périodes de repos. L'idée exploitée ici est que si l'unité reste inactive un temps suffisant, il est préférable de la mettre dans un des modes basse consommation. Cependant, nous avons vu dans le chapitre deux que les pénalités engendrées par ces changements de modes peuvent anéantir tout effort. Nous proposons ici une méthode pour déterminer l'opportunité d'appliquer un changement de modes en se basant sur les deux seuils Δ_{idle} et Δ_{sleep}.

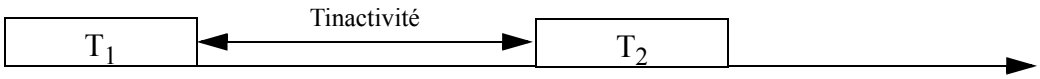
1. http://www.intel.com/design/strong/sa110.htm

3.2.Détermination des valeurs des seuils de passage en mode basse consommation.

Soient Δ_{sleep} et Δ_{idle}, les seuils respectifs à partir desquels il est rentable de passer l'unité en mode *sleep* ou en mode *idle*. Ces seuils sont utilisés pour décider des passages en mode basse consommation lorsque la ou les unités sont inactives. Voyons comment les déterminer.

Considérons une unité exécutant une tâche à une fréquence donnée puis inactive, et une échéance donnée, figure 54.

FIGURE 54. Détermination des temps seuils



Notons

- P_{nop}, la puissance dissipée par l'unité en mode de fonctionnement normal pendant le temps d'inactivité,
- P_{idle}, la puissance dissipée par l'unité en mode *idle* (elle est par exemple de 50 mW pour le StrongARM),
- $P_{tâche}$, la puissance dissipée par l'unité exécutant la tâche T_i,
- T_i^f, la durée d'exécution de la tâche T_i,
- $E_{idle\text{-}run}$, l'énergie dissipée pour passer du mode *idle* au mode *run* (dans le cas du StrongARM, on calcule le pire cas qui serait 400mW multiplié par 10µs),
- $E_{run\text{-}idle}$, l'énergie dissipée pour passer du mode *run* au mode *idle* (dans le cas du StrongARM, on calcule le pire cas qui serait 400mW multiplié par 10µs),
- P_{sleep}, la puissance dissipée par le processeur en mode *sleep* (elle est par exemple de 160µW pour le StrongARM),
- $E_{idle\text{-}sleep}$, l'énergie dissipée pour passer du mode *idle* au mode *sleep* (dans le cas du StrongARM, on calcule le pire cas qui serait 20 mW multiplié par 90µs),
- $E_{sleep\text{-}run}$, l'énergie dissipée pour passer du mode *sleep* au mode *run* (dans le cas du StrongARM, il serait 400 mW multiplié par 160 ms),
- $T_{inactivité}$, le temps d'inactivité du processeur,
- $T_{idle\text{-}run}$, le temps nécessaire au passage du mode *idle* au mode *run* (dans le cas du StrongARM, ce temps est de 10µs),

- $T_{run\text{-}idle}$, le temps nécessaire au passage du mode *run* au mode *idle* (dans le cas du StrongARM, ce temps est de 10µs),
- $T_{sleep\text{-}run}$, le temps nécessaire au passage du mode *sleep* au mode *run* (dans le cas du StrongARM, ce temps est de 160ms),
- $T_{run\text{-}sleep}$,le temps nécessaire au passage du mode *run* au mode *sleep* (dans le cas du StrongARM, ce temps est de 90µs).

Détermination de Δ_{idle} seuil d'inactivité pour passer en mode repos *idle* :

Considérons le cas a), où le processeur restera en mode de fonctionnement pendant la période d'inactivité, et le cas b), où il passera en mode idle pendant cette même période. Définissons le bilan énergétique avec :

a) $E_a = P_{T_1} \times T_1^f + P_{nop} \times T_{inactivité} + P_{T_2} \times T^f_2$

b) $E_b = P_{T_1} \times T^f_1 + P_{T_2} \times T_2^f + P_{idle} \times (T_{inactivité} - (T_{idle\text{-}run} + T_{run\text{-}idle})) + E_{run\text{-}idle} + E_{idle\text{-}run}$

Dans le cas b), il y a un coût énergétique supplémentaire pour passer du mode *run* au mode *idle* et un autre pour passer du mode *idle* au mode *run* pour exécuter la tâche T_2.

Le passage en mode *idle* durant la période d'inactivité est bénéfique si $E_b < E_a$. Calculons la différence de ces deux énergies:

$$E_b - E_a = P_{idle} \times (T_{inactivité} - (T_{idle\text{-}run} + T_{run\text{-}idle})) + E_{run\text{-}idle} + E_{idle\text{-}run} - P_{nop} \times T_{inactivité}$$

On en déduit : $$T_{inactivité} > \frac{E_{run\text{-}idle} + E_{idle\text{-}run} - P_{idle} \times (T_{idle\text{-}run} + T_{run\text{-}idle})}{P_{nop} - P_{idle}} \qquad \text{(equ1)}$$

Dans le même temps, il faut s'assurer que la période d'inactivité est supérieure strictement aux pénalités temporelles (ce qui est une condition minimale): $T_{inactivité} \geq T_{run\text{-}idle} + T_{idle\text{-}run}$ (equ 2).

La valeur de Δ_{idle} est alors:

$$\Delta_{idle} = \frac{E_{run\text{-}idle} + E_{idle\text{-}run} - P_{idle} \times (T_{idle\text{-}run} + T_{run\text{-}idle})}{P_{nop} - P_{idle}} \qquad \text{(equ3)}$$

Détermination de Δ_{sleep} seuil d'inactivité pour passer en mode repos *sleep* :

Procédons de même que précédemment. Considérons le cas b) ci-dessus (passage en mode *idle*) et le cas c) où l'unité passe en mode *sleep* durant la période d'inactivité.

Le bilan énergétique du cas c) est :

$$c) E_c = P_{T_1} \times T^f_1 + P_{T_2} \times T^f_2 + P_{sleep} \times (T_{inactivité} - T_{sleep\text{-}run} - T_{run\text{-}sleep}) + E_{run\text{-}sleep} + E_{sleep\text{-}run}$$

Notons que dans le cas du StrongARM (figure 53), $T_{sleep\text{-}run}$ est plus de 1000 fois supérieur à $T_{idle\text{-}run}$, on ne peut donc pas les négliger, on peut par contre négliger le terme $P_{idle} \times (T_{idle\text{-}run} + T_{run\text{-}idle})$ .Rappelons que P_{idle} est 20 fois moins élevé que la puissance moyenne en mode normal.

On en déduit que $T_{inactivité}$ doit être supérieure à : $\frac{P_{sleep} \times (T_{sleep\text{-}run} + T_{run\text{-}sleep}) - (E_{run\text{-}sleep} + E_{sleep\text{-}run}) + (E_{run\text{-}idle} + E_{idle\text{-}run}) - P_{idle} \times (T_{idle\text{-}run} + T_{run\text{-}idle})}{P_{sleep} - P_{idle}}$

Soit, en négligeant le terme pré-cité:

$$T_{inactivité} > \frac{P_{sleep} \times (T_{sleep\text{-}run} + T_{run\text{-}sleep}) - (E_{run\text{-}sleep} + E_{sleep\text{-}run}) + (E_{run\text{-}idle} + E_{idle\text{-}run})}{P_{sleep} - P_{idle}} \text{(equ4)}$$

De même que précédemment, il faut s'assurer que la période d'inactivité est supérieure strictement aux pénalités temporelles (ce qui est une condition minimale): $T_{inactivité} > T_{run\text{-}sleep} + T_{sleep\text{-}run}$ (equ5) .

Ainsi, le passage en mode *sleep* est bénéfique si $T_{inactivité}$ respecte les inéquations (equ4) et (equ5). Finalement, on en déduit :

$$\Delta_{sleep} = \frac{P_{sleep} \times (T_{sleep\text{-}run} + T_{run\text{-}sleep}) - (E_{run\text{-}sleep} + E_{sleep\text{-}run}) + (E_{run\text{-}idle} + E_{idle\text{-}run})}{P_{sleep} - P_{idle}} \text{(equ6)}$$

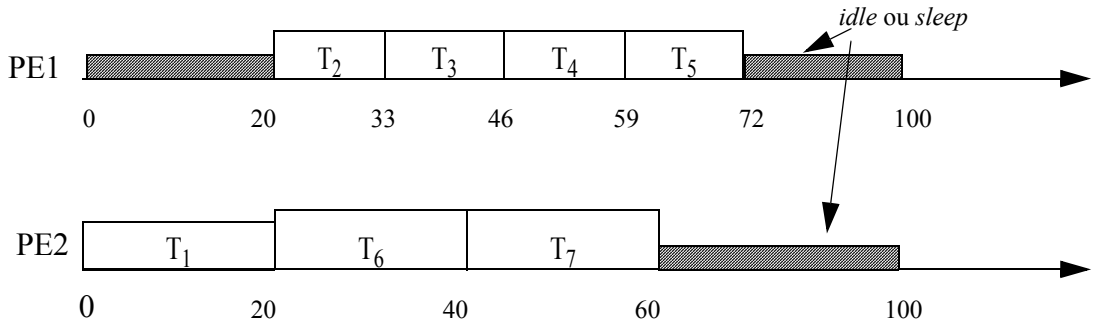
A partir de ces deux valeurs de seuils propres à chaque unité de l'architecture, nous analysons les périodes d'inactivité des unités. Si la période d'inactivité est :

- comprise entre Δ_{idle} et Δ_{sleep}, l'unité est mise en état *idle*,
- supérieure ou égale à Δ_{sleep}, l'unité est mise en état *sleep*, et on compte une pénalité en temps et en consommation au prochain réveil.

3.3.Résultats.

Etudions le cas de l'ordonnancement de [BL97] utilisé préalablement tout au long de ces optimisations. Considérons l'ordonnancement optimisé par ajustement global (figure 55).

FIGURE 55. Ordonnancement de l'exemple de [BL97] après ajustement global



Afin de déterminer dans quel mode (*sleep* ou *idle*) mettre les unités durant les périodes d'inactivité de 72 µs à 100 µs, et de 60 µs à 100 µs, calculons les Δ_{idle} et Δ_{sleep} comme présentés précédemment. Considérons que les deux unités présentes dans cette architecture sont deux processeurs StrongARM.

Dans ces conditions, appliquons les inéquations equ3 et equ6. Après calculs, la période d'inactivité doit être au moins de $\Delta_{idle} \sim 40\mu s$ pour que la mise en mode *idle* soit rentable, et de $\Delta_{sleep} \sim 10000\mu s$ pour que la mise en mode *sleep* soit rentable.

Ainsi, pour les trois périodes d'inactivité, de 20, 18 et 40 µs chacune, seule la dernière période permet de mettre la deuxième unité en mode *idle*. La puissance *idle* du StrongARM est 0,02 W et supposons que la puissance du StrongARM quand il n'exécute aucune tâche mais n'est pas dans un mode basse consommation est P_{nop}= 200 mW.

En supposant que l'ordonnancement est exécuté par deux processeurs StrongARM, nous prenons en moyenne des puissances de 400 mW pour T_6 et T_7 qui ne sont pas ralenties, et de 300 mW pour T_1. Pour les tâches exécutées par la première unité, on prend une puissance en moyenne de 300mW.

Globalement, on passe d'une énergie de 53 600 nJ à 46 000 nJ soit un gain en énergie de 13%. Le pic de puissance n'est pas modifié.

Ici, aucun gain en puissance ne peut être réalisé, alors qu'avec une optimisation ajustant la tension et la fréquence un gain en puissance notable est réalisé. Toutefois, l'ordonnancement et ses résultats obtenus sont intéressants si cette configuration permet dans le même temps à la batterie de récupérer un peu de sa capacité énergéti-

que, et donc d'augmenter sa durée de vie, et ceci, de manière plus importante que dans le cas d'un ajustement.

Mais pour cela, il faudrait pouvoir évaluer la quantité d'énergie récupérée. Des études plus poussées liées à la batterie sont à mener en ce sens.

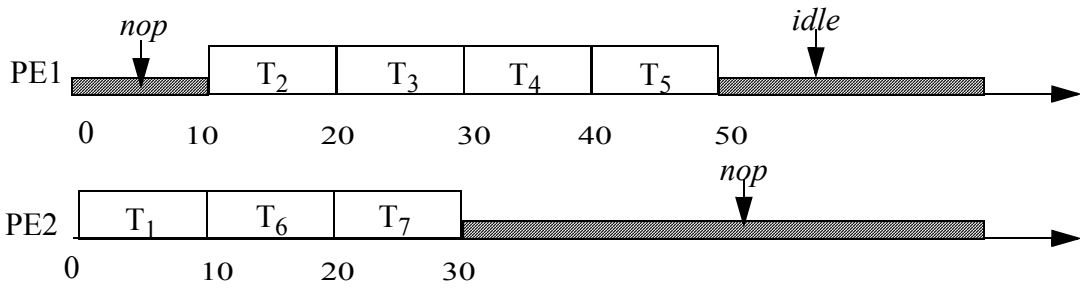
3.4.Comparaison de la gestion DVS et modes basse consommation.

Réduire le pic de puissance permet d'augmenter la durée de vie des batteries. Les périodes de repos aussi. Faisons un bilan énergétique en comparant un ordonnancement ralenti au maximum et un ordonnancement non ralenti et sur lequel une mise en repos (*idle* ou *sleep*) est réalisée.

Prenons l'exemple de COSYN-LP. Considérons en premier l'ordonnancement optimisé obtenu après avoir appliquée la stratégie d'ajustement mixte, ordonnancement qui était présentée à la figure 52.

Considérons maintenant l'ordonnancement initial rappelé sur la figure 56.

FIGURE 56. Ordonnacement initial [BL97]



Dans le cas non optimisé avec mise en repos (figure 56), l'énergie dissipée est de :

$$E_{init} = E(T_1) + E(T_2) + E(T_3) + E(T_4) + E(T_5) + E(T_6) + E(T_7) + E_{idle} + E_{nop}$$

Dans le cas du ralentissement (figure 52), l'énergie dissipée par chaque tâche est réduite (E_{new}), ce qui conduit à l'énergie globale :

$$E_{raf} = E_{new}(T_1) + E_{new}(T_2) + E_{new}(T_3) + E_{new}(T_4) + E_{new}(T_5) + E_{new}(T_6) + E_{new}(T_7)$$

Comparons E_{init} et E_{raf} :

Nous savons que $E_{raf} < E(T_1) + E(T_2) + E(T_3) + E(T_4) + E(T_5) + E(T_6) + E(T_7)$, car ce sont les mêmes tâches exécutées par les mêmes unités à des fréquences différentes. A cette sommation, il faut ajouter les énergies dues aux périodes en mode *nop* et en période *idle*. Il est donc évident que le gain le plus grand est obtenu dans la configuration où l'unité est ralentie. Pour cette raison, l'exploitation des modes basse consommation est réalisée après les optimisations par ajustement.

Comme dans les changements de <fréquence,tension> DVS/DFS, des pénalités dues aux passages du mode *run* au mode *idle* et inversement sont à prendre en compte. Ces pénalités peuvent être importantes (10 µs du mode *idle* au mode *run* et 160 ms du mode *sleep* au mode *run*). Afin de les prendre en compte, nous proposons la même stratégie que dans le paragraphe 2.1.a traitant des pénalités, à savoir minorer le temps d'inactivité par les temps dus aux changements de régime.

Toutefois, pour un système embarqué muni d'une batterie, le comportement de charge et de décharge de celle-ci n'est pas linéaire [MS01, MS99,DN95], ainsi il ne faut pas considérer simplement le bilan énergétique. En effet, la batterie se décharge, puis lors de «moments de relaxation», c'est-à-dire de moments de repos suffisamment importants, un phénomène de légère recharge se produit, augmentant la durée de vie de la batterie. Dans le même temps, il est à considérer que le pic de puissance a un impact plus grand que la puissance moyenne. Un pic important peut décroître rapidement la durée de vie de la batterie [MS01, MS99,DN95]. Là encore, il faut trouver des compromis, mais pour cela des études plus poussées sont à mener.

4•Conclusions et perspectives

Dans ce chapitre, nous avons d'abord présenté les différentes optimisations effectuées lors de l'étape d'allocation/ordonnancement de CODEF. Les optimisations lors de l'allocation permettent des gains en énergie et en puissance intéressants.

Toutefois, l'ordonnancement peut bien souvent encore être raffiné. Nous avons ensuite présenté différentes techniques de raffinement d'ordonnancement exploitant l'ajustement conjoint en tension et en fréquence afin de réduire la consommation globale. Trois stratégies principales ont été développées et testées sur divers exemples.

La première stratégie est basée sur une détermination de fréquence de fonctionnement individuelle pour chaque tâche. Les gains en énergie et en puissance sont modérés en raison de sa focalisation sur les premières tâches rencontrées par l'heuristique. Ainsi, peu de tâches peuvent être ralenties, mais celles qui le sont, le sont au maximum. L'énergie est diminuée mais la puissance ne l'est pas toujours, en raison du recouvrement modifié des tâches.

La deuxième stratégie repose sur la détermination d'un facteur moyen d'ajustement durant toute l'exécution de l'application pour le ou les unités munies du Dynamic Voltage Scaling. Ainsi, un plus grand nombre de tâches est ralenti par rapport à la première stratégie, conduisant à une optimisation plus «équilibrée». Mais en général, il est encore possible de raffiner.

Finalement, la troisième stratégie allie les avantages de ces deux dernières stratégies et élimine les inconvénients de la première réalisée seule et permet d'améliorer encore les gains en puissance et en énergie. En effet, en appliquant d'abord un ralentissement global, puis un ralentissement individuel pour les tâches, il y a moins de changements dynamiques des couples <fréquence,tension> et les recouvrements de tâches modifiées le sont avec des tâches ralenties, diminuant ainsi le pic de puissance. Cette stratégie nous paraît être un bon compromis. En effet, sur quelques cas d'études, nous obtenons des solutions avec des gains en énergie et en puissance notables. Nous avons comparé nos méthodes d'optimisation en consommation à des cas d'études, tels que ceux de [BG00] et de COSYN-LowPower [BL97]. Pour ce dernier exemple, les techniques d'optimisation que nous avons mises en oeuvre permettent d'affiner leurs ordonnancements optimisés et de gagner encore en énergie et en puissance par rapport à leurs solutions optimisées.

Ainsi, l'interaction de tous les facteurs intervenant dans la consommation nous oblige à nous orienter vers la recherche de compromis. On ne peut baisser la puissance sans que cette action ne se répercute sur la surface ou le temps par exemple.

En résumé, optimiser la puissance requiert :

- d'éviter d'augmenter le parallélisme d'exécution de tâches,
- de tout faire pour ralentir les tâches quand il y a des tâches exécutées en parallèle.

Optimiser l'énergie, requiert de :

- ralentir au maximum les tâches, même si pour cela, plus de tâches sont exécutées en parallèle,
- tout faire pour éviter d'avoir des périodes de repos, mais plutôt avoir tout le temps des tâches ralenties exécutées en parallèle.

On voit bien qu'il y a contradiction, et qu'une optimisation simultanément en puissance ou en énergie nécessite de trouver des compromis.

L'optimisation de l'allocation, alliée à la stratégie de raffinement d'ordonnancement mixte permettent d'obtenir de manière efficace ce type de compromis. En effet, nos résultats sur les exemples traités montrent que la puissance et l'énergie sont baissées en respectant les contraintes de temps d'exécution.

Une fois ces optimisations par ajustement <fréquence, tension> effectuées, nous avons optimisé en exploitant les modes basse consommation. Cependant, d'après les analyses comparatives des bilans énergétiques des diverses stratégies de ce chapitre, il est préférable de ralentir au maximum les tâches plutôt que d'exploiter uniquement les modes basse consommation.

Conclusions et Perspectives

Le problème de la consommation d'énergie dans les systèmes embarqués étant devenu prédominant, cette thèse étudie les méthodes de conception basse consommation de systèmes.

Un des problèmes de la conception système est le partitionnement d'applications qui requiert l'utilisation de méthodes complexes. En effet, le partitionnement sous contrainte de temps, basé sur un algorithme d'ordonnancement avec un objectif de minimisation de la surface de silicium ou de la consommation est un problème NP-difficile.

La prise en compte de la consommation (énergie et pic de puissance) nécessite au préalable de caractériser les systèmes. Afin d'atteindre cet objectif, nous avons intégré un estimateur de consommation des architectures systèmes basé sur des modèles de consommation élaboré pour les unités présentes dans les architectures systèmes.

Nous avons présenté les méthodes adoptées pour ces caractérisations. Les comportements des processeurs sont à différencier selon que leur architecture est de type RISC ou de type DSP. Des estimateurs de performances/consommation de processeurs RISC sont disponibles en logiciels libres et efficaces. Les processeurs de type DSP ont nécessité une étude plus fine qui a été présentée au chapitre quatre et dans l'annexe. Leur comportement en consommation est complexe en raison de l'hétérogénéité de ce type d'architecture. Pour ces raisons, des modèles en consommation ont été définis au niveau des unités fonctionnelles du processeur. De plus, les compilateurs existant sont inefficaces pour les processeurs DSP. Pour cette raison, en collaboration avec Philips Semiconductors, l'équipe a développé un outil permettant d'avoir une estima-

tion de performances de code assembleur optimisé à partir d'un code C. Afin d'automatiser l'estimation en consommation d'un code C, nous avons étendu cet outil avec les modèles de consommation.

La réduction de la consommation est à l'heure actuelle un problème très étudié. Pour autant, nombre de difficultés restent à l'ordre du jour liées surtout à l'interaction des divers facteurs entrants dans la consommation. L'action sur un facteur peut avoir des conséquences diverses sur l'architecture. Par exemple, réduire la surface de silicium est un but recherché dans les SOC, or, un des facteurs permettant de diminuer la consommation du système est de paralléliser les unités afin d'exploiter les ajustements conjoints en tension et en fréquence. Ainsi, il est difficile de ne cibler la réduction de la consommation que par un seul de ces facteurs, ils doivent tous être pris en compte afin de trouver des solutions de compromis.

A partir de la connaissance des méthodes de partitionnement, et particulièrement de celle que nous avons exploitée, nous avons cherché à définir les interventions possibles dans le but de réduire la consommation.

Nous avons déterminé deux étapes principales d'intervention dans les méthodes de partitionnement. Ces étapes sont générales aux méthodes de partitionnement et offrent des possibilités d'optimisation.

Ces étapes sont l'allocation et l'ordonnancement, réalisées de manière conjointe dans le cas de l'outil d'exploration d'architectures systèmes CODEF.

Aussi, dans un premier temps, nous avons élaboré une stratégie d'allocation fonction de l'énergie et de la puissance. Dans un deuxième temps, nous avons élaboré des stratégies de raffinement de l'ordonnancement. Celles-ci exploitent l'ajustement conjoint en tension et en fréquence. Trois stratégies de raffinement de l'ordonnancement ont été développées et testées sur divers exemples.

La première stratégie est basée sur une détermination de fréquence de fonctionnement individuelle pour chaque tâche. Les gains en énergie et en puissance sont modérés en raison de sa focalisation sur les premières tâches rencontrées par l'heuristique.

La deuxième stratégie repose sur la détermination d'un facteur moyen d'ajustement pour le ou les unités munies du Dynamic Voltage Scaling, durant toute l'exécution de l'application.

Finalement, la troisième stratégie allie les avantages de ces deux dernières stratégies et élimine les inconvénients de la première réalisée seule. En effet, en appliquant d'abord un ralentissement global, puis un ralentissement individuel pour les tâches, il y a moins de changements dynamiques des couples <fréquence,tension> et les recouvrements de tâches modifiées le sont avec des tâches ralenties, diminuant ainsi le pic de puissance. Cette stratégie nous paraît être un bon compromis.

Ce raffinement permet des gains en énergies notables.

Une fois que ces voies ont été exploitées, nous réalisons une gestion des modes basse consommation, en mettant le ou les processeurs en état de repos ou repos profond dès que la possibilité se présente et que ce changement de régime permet de gagner en consommation.

Malgré les optimisations obtenues avec les techniques développées, certaines améliorations peuvent être apportées à ce travail, telle que la prise en compte dans l'estimation et l'optimisation de la consommation des réseaux sur puce. Cette prise en compte complexe nécessite une étude préalable, car on trouve peu d'études à ce propos dans la littérature, et ce problème est loin d'être trivial. Dans le contexte de notre travail, nous avons besoin de disposer de modèles génériques à intégrer dans nos méthodes.

Comme perspectives ouvertes par ce travail, citons l'intérêt d'étudier la gestion de la puissance possible par les systèmes d'exploitation qui sont amenés à prendre de plus en plus d'importance dans le domaine de la gestion de l'énergie. Ainsi, ils se situent à un niveau stratégique de l'architecture pour gérer finement la puissance durant les ordonnancements des tâches. De plus, les architectures hétérogènes nécessitent parfois de répartir des systèmes d'exploitation temps réel sur plusieurs unités. Là encore, il est nécessaire d'étudier des approches de gestion/réduction de la consommation dans une telle architecture logicielle/matérielle.

Bibliographie

[AM01]

N. AbouGhazaleh, D. Mossé, B. Childers and R. Melhem «Toward The Placement of Power Management Points in Real Time Applications», *COLP'01 (Workshop on Compilers and Operating Systems for Low Power), Barcelona, Spain, September 9th, 2001.*

[Arm]

http://www.arm.com/techdocs.nsf/html/ARM9Docs

[BA98]

L. Bianco, M. Auguin, G. Gogniat, A. Pegatoquet, «A path analysis based partitioning for time constrained embedded systems», *in Proc. Int. IEEE/ACM Workshop Codes/ CASHE Seattle, pp 85-89, March 1998.*

[BD97]

Luca Benini, Giovanni De Micheli, Enrico Macii, Massimo Poncino, Riccardo Scarsi «Fast power estimation for deterministic input streams». *in proceedings of ICCAD 1997 (IEEE/ACM International Conference on Computer Aided Design) pp.494-501, November 9-13 1997, SanJose, California.*

[BD98]

Luca Benini, Giovanni De Micheli, «Dynamic Power Management; Design Techniques and CAD Tools» *Kluwer Academic Publishers, 1998.*

[BE95]

A. Bellaouar, M.I. Elmasry, «Low-Power digital vlsi design : circuits and systems», Kluwer Academic publishers, 1995.

[BG00]

Luca Benini, Giovanni De Micheli, «System-Level Power Optimization : Techniques and Tells», *in proceedings* of *ACM Transactions on Design Automation of Electronic Systems, vol. 5, No. 2, Avril 2000, pages 115-192.*

[BG88]

G. Berry, G. Gonthier, «The ESTEREL synchronous programming language: design, semantics, implementation.» *Rapport de recherche 842, INRIA, mai 1988.*

[BG99]

S. Bilavarn, G. Gogniat, J.L. Philippe, «A Hardware-Software Codesign Methodology for Heterogeneous Architecture Estimation», *in proceedings of ICSPAT'99, Orlando, Novembre 1999.*

[BH03]

BEN FRADJ Hanène, rapport de DEA SiCom « Estimation et Optimisation de la consommation des accès mémoires dans un système embar qué», Université de Nice Sophia-Antipolis, 2003.

[BK02]

K. Ben Chehida, M. Auguin. «HW/SW Partitioning Approach For Reconfigurable System Design», *in proceedings of International conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES02, Grenoble, October 2002.*

[BL97]

Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha, «COSYN: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems» *in proceedings of Design Automation Conference 1997, pp.703-708, June 9-13, 1997 Anaheim Convention Center Anaheim, California.*

[BL99]

Luc Bianco, «Méthode de partitionnement pour la conception de systèmes embarqués de traitement de signal temps réel» *thèse de lUniversité de Nice - Sophia Antipolis, Octobre 1999.*

[BM97]

P. Bjørn-Jørgensen, J. Madsen, «Critical path driven cosynthesis for heterogeneous target architectures» *in Proc. 5th Codes/CASHE97, 15-19, Braunschweig, March 1997.*

[BM99]

Luca Benini, Alberto Macii, Enrico Macii, Massimo Poncino, Riccardo Scarsi: «Synthesis of Low-Overhead Interfaces for Power-Efficient Communication over Wide Buses» *in proceedings of Design Automation Conference 1999 pp.128-133, June 21-25, 1999, New Orleans, LA.*

[BN92]

R. Burch, F. Najm, P. Yang, T. Trick, «McPower: a Monte Carlo approach to power estimation», *in proceedings of IEEE/ACM International Conference on Computer-Aided Design, Santa Clara, CA, pp. 90-97, November 8-12, 1992.*

[CA00]

Capella, M. Auguin, E. Gresset, «Synthesis of signal processing systems from binary controlled data flow specifications», *Int. Conference on Signal Processing Applications and Technology, Dallas, 16-19 Octobre 2000*

[CB94]

P.Chou and G.Borriello, «Software scheduling in the co-synthesis of reactive real-time systems.» *In Proc. Design Automation Conference, pages 1-4, June 1994.*

[CB99]

E.Y. Chung, L. Benini, A. Bogliolo, G.D. Micheli, «Dynamic Power Management for non-stationary service requests», *in proceedings of IEEE Design Automation and Test in Europe, March 9 - 12, 1999, Munich, Germany.*

[CF00]

F. Cuesta, M. Auguin, E. Gresset, «System level communication synthesis for embedded signal processing applications», *in proceedings of Int. Conference on Signal Processing Applications and Technology, Dallas, 16-19 Octobre 2000.*

[CG03]

Gilbert Cabillic, «Minimisation de la consommation énergétique à l'aide du système d'exploitation», *Ecole thématique Architectures des systèmes matériels enfouis et méthodes de conception associées, Roscoff (Finistère) 31 mars - 4 avril 2003.*

[CJ90]

J.P. Calvez «Spécification et conception des systèmes. Une méthodologie.» *Édition Masson, 630 pages,1990.*

[CJ99]

Jean-Gabriel Cousin, «Méthodologies de conception de coeurs de processeurs spécifiques (ASIP): mise en oeuvre sous contraintes, estimation de la consommation», *Thèse de l'Université de Rennes 1, Septembre 1999.*

[CM87]

M. A. Cirit, «Estimation dynamic power consumption of CMOS circuits», *IEEE Int. Conf. on CAD, Nov. 1987, p.534-537.*

[CR95]

T.L.Chou and K.Roy, «Statistical estimation of sequential circuit activity», *in Proc. Int. Conf. Computer-Aided Design, pp. 34-37, nov 1995.*

[DA03]

Dong Wu and Bashir M. Al-Hashimi, Petru Eles, «Scheduling and Mapping of Conditional Task Graphs for the Synthesis of Low Power Embedded Systems», *in proceedings of Design Automation Test in Europe 2003 Munich, Germany, March 3-7, 2003, p.90-95.*

[DJ98]

Robert P. Dick, Niraj K. Jha, «MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Co-Synthesis of Distributed Systems», *IEEE Transactions on CAD of Integrated Circuits and Systems, vol. 17, nunéro 10, pp.920-935, 1998.*

[DJ99]

R.P. Dick and N. K. Jha, «MOCSYN: Multiobjective core-based single-chip system synthesis», *in Proc. Design Automation &Test in Europe Conf., pp. 263-270, Mars 1999.*

[DK01]

Dongkun Shin, Jihong Kim, Seongsoo Lee , «Low-Energy Intra-Task Voltage Scheduling Using Static Timing Analysis», *in proceedings of Design Automation Conference 2001, p. 438.*

[DN95]

M.Doyle, J.Newman, «The use of Mathematical Modeling in the Design of Lithium/Polymer Battery Systems», *Electrochemica Acta, Vol.40, No. 13 14, pp.2191-2196, 1995.*

[EH93]

Ernst R., Henkel J., and Benner T., «Hardware-Software Cosynthesis for Microcontrollers», *IEEE Des. Test Computer, 64-75, 1993.*

[EP97]

Eles, Peng, Kuchcinski, Doboli, «System Level Hardware/Software Partitioning based on Simulated Annealing and Tabu Search», *in proceedings of Design Automation for Embedded Systems 2, pp.5-32, January 1997.*

[FC98]

Francky Catthoor et al, «Custom memory management methodology: Exploration of memory organisation for embedded multimedia system design», *Kluwer Academic Publisher, 1998. ISBN 0-7923-8288-9.*

[FC99]

Supplementary users guide to Fastcap (and FFTCap) with Unix hints Jonathan F. Crawford, Haverford College July 1999. http://www.haverford.edu/physics-astro/smith/fastcap.html

[FP98]

Robert A. Freking and Keshab K. Parhi, «Theorical estimation of power consumption in binary address», *DARPA DA/DABT63-96-C-0050. IEEE 1998, vol2, page début 453, page fin 457.*

[GC02]

Giorgio C. Buttazzo, «Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications», *Kluwer Academic Publishers, 2002.*

[GN02]

Gwenolé Corre, Nathalie Julien, Eric Senn, Eric Martin, “Optimisation de la consommation des unités de mémorisation lors de la synthèse d’architecture», *du recueil des communicaitons du 3ème Colloque de CAO de Circuits et Systèmes Intégrés, Paris, 15, 16-17 Mai 2002.*

[GN03]

Bruno Gaujal, Nicolas Navet, «Ordonnancement sous contraintes de temps et d’énergie», *Ecole d’été Temps Réel-Qualité de Service, Toulouse, 9-12 Septembre 2003, organisée par l’Institut de Recherche en Informatique de Toulouse.*

[GN97]

S. Gupta and F. N. Najm, «Power Macromodeling for high level power estimation», *in Proc. Design Automation Conf., pp. 365-370, June 1997.*

[GNC03]

Bruno Gaujal, Nicolas Navet, Cormac Walsh, «Real-time scheduling for optimal energy use», *du recueil des communicaitons des 4ièmes journées d'études Faible Tension Faible Consommation, 15-16 Mai 2003, Paris France.*

[GP00]

Patricia Guiton-O, «Modélisation de la consommation d'un processeur type DSP», rapport de DEA Propagation, Télécommunications et Télédétection, Université de Nice Sophia-Antiopolis, Juin 2000.

[HC91]

N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. «The synchronous data-flow programming language LUSTRE». *IEEE proceedings, volume 79, Sept. 1991.*

[HD87]

D. Harel. «Statecharts : a visual formalism for complex systems.» *Science of Computer Programming, 8, 231-274 - 1987.*

[HE96]

J. Henkel, R. Ernst. «The interplay of run-time estimation and granularity in HW/SW partitioning», *in Proc. Codes/CASHE96, Pittsburgh, 1996.*

[HL01]

Jorg Henkel, Haris Lekatsas, Venkata Jakkula, «Encoding Schemes for Address Buses in Energy Efficient SOC», in *proceedings of VLSI-SOC pp. 242-246, Montpellier 3-5 December 2001.*

[IE88]

IEC, Genève. «Preparation of function charts for control systems». *International standard IEC 848. Déc. 1988.*

[IK01]

M. J. Irwin, M. Kandemir, and N. Vijarkrishnan, «Simple Power: A Cycle-Accurate Energy Simulator», in proceedings of *IEEE Computer Society Technical Committee on Computer Architecture Newsletter, Januar 2001, pp59-64.*

[In00]

Intel® XScale «Core Developer s Manual December, 2000 Order Number: 273473-001»

[In02]

Intel, Compaq, Microsoft, Phoenix, and Toshiba, «Advanced Configuration and Power Interface Component Architecture Programmer» *Reference Core Subsystem, Debugger, and Utilities Revision 1.13 May 3, 2002.*

[JN04]

Johann Laurent, Nathalie Julien, Eric Senn, Eric Martin, «Functional level power analysis: an efficient approach for modeling the power consumption of complex processors», *in prodeedings of DATE 2004, pp.666-667, 16-20 Février 2004, Paris, FRANCE.*

[Jo]

http://dry-martini.mit.edu/JouleTrack/

[JP97]

Janardhan H. Satyanarayana, Keshab K. Parhi, «A theorical Approach to estimation of bounds on power consuption in digital multipliers» *in proceedings of IEE1997, vol 44, num 6, mois: 06, pages 473-481*

[JR]

Jeff Russel, «Assembly code power analysis of a high performance embedded processor family», *http://www.ece.utexas.edu/~jrussell/power_instr/report-sec2.html*

[KG74]

G. Kahn, «The Semantics of a Simple Language for Parallel Programming», *in J.L. Rosenfeld, editor, Information Processingg 74: Proc. IFIP Congress 74, North-Holland, pages 471-475, August 1974.*

[KG97a]

M. Kamble, K. Ghose «Analytical energy dissipation models for low power caches» *in proceedings of Internatinal Symposium. On low Power Electronic and Design, pp143-148, August 1997.*

[KG97b]

M. Kamble, K. Ghose «Energy-Efficient of VLSI caches: a comparative study» *in Proc. IEEE 10 th Internatinal Conference on VLSI Design, pp261-267, Junuary 1997*

[KG98]

M. Kamble, K. Ghose «Modeling energy dissipation in low power caches». *Technical Repport, CS-TR-98-02, Departement of computer science, SUNY6Binghamton, 1998.*
http://www.eecg.toronto.edu/~wonho/papers/doc/kamble98modeling.pdf

[KK02]

I. Kadayif, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam «EAC: A Compiler Framewor k for High-Level Energy Estimation and Optimization» *Microsystems Design Lab, in Proceedings of DATE2002, pp.436-442. 4-8 March, 2002 Paris, France.*

[KL94]

A. Kalavade, E.A. Lee, «A global critically/local phase driven algorithm for the constrained hardware/software partitioning problem», *in proceedings of the Third Interantional Workshop on Hardware/Software Codesign, Grenoble, pp. 42-48, 1994.*

[LC01]

Jinfeng Liu, Pai H. Chou, Nader Bagherzadeh, Fadi Kurdahi, «A Constraint-based Application Model and Scheduling Techniques for Power-aware Systems», *Proceedings of the Ninth International Symposium on*

Hardware/Software CodesignCODES 2001, [p. 153], April 25-27, 2001 Copenhagen, Denmark.

[LJ00]

Jiong Luo and Niraj K. Jha, «Power-conscious joint Scheduling of Periodic Task Graphs and Aperiodic Tasks in Distributed Real-time Embedded Systems», *ICCAD 2000, Nov 5-9, 2000, San Jose, CA, proceedings, p. 357.*

[LJ01]

Jiong Luo, Niraj K. Jha «Battery-Aware Static Scheduling for Distributed Real-Time Embedded Systems», *in proceedings of 38th Design Automation Conference 2001 [p. 444] June 18-22, 2001 Las Vegas.*

[LL03]

Marisa Lopez-Valejo, Juan Carlos Lopez, «On the Hardware-Software Partitioning Problem : System Modeling and Partitioning Techniques», *in ACM Transactions on Design Automation of Electronic Systems, vol. 8, No.3, July 2003, pp.269-297.*

[LL73]

C.L.Liu and J.W. Layland, «Scheduling algorithms for multiprogramming in hard real-time environment», *journal of the ACM, 20(1):40-61, février 1973.*

[LM87]

E.A. Lee and D.G. Messeschmitt, «Synchronous Data Flow», *IEEE Proceedings, September, 1987.*

[LR03]

Y.-H. Lee, K.P. Reddy, C.M. Krishna, «Scheduling Techniques for reducing Leakage Power in Hard Real Time Systems», *in proceedings of EUROMICRO, pp.105-113, Porto, Portugal, 2-4 July 2003.*

[LR94]

P. Landman, «Low-Power Architectural Design Methodologies», *Ph.d Thesis, University Berkeley, Memorandum No. UCB/ERL M94/62, 30th August 1994.*

[LR95]

Paul E. Landman, Jan M. Rabaey, «Architectural power analysis : the Dual Bit Type», *VLSI Systems, vol. 3 n. 2, June 1995.*

[LX03]

Li-Chuang Weng, XiaoJun Wang, Bin Liu, «A survey of Dynamic Power Optimization Techniques», *the Third IEEE International Workshop on System On Chip for Real-Time Applications, Calgary, Alberta Canada, June 30- July 2, 2003.*

[MA00]

Daniel Mossé, Hakan Aydin, Bruce Childers and Rami Melhem, «Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications», *Workshop on Compilers and Operating Systems for Low Power, Oct. 2000.*

[MB03]

Thèse, Miramond B., «Méthode d'optimisation pour le partitionnement logiciel/matériel de systèmes à description multi-modèles.», *Université d'Evry, Décembre 2003.*

[MD00]

Mariagiovanna Sami, Donatella Sciuto, Cristina Silvano, Vittorio Zaccaria, «Power Exploration for Embedded VLIW Architectures» *of proceedings of ICCAD 2000, Nov 5-9, 2000, San Jose, CA.*

[MF00]

Fabrice Muller, Thèse «Outil pour l'aide à la conception conjointe des systèmes matériel/logiciel», *Université de Nantes, Janvier 2000.*

[MF01]

Marc Fleishmann, «LongRun Power Management Dynamic Power Management for Crusoe Processors», *Transmetta Corporation, January 17, 2001.*

[MH00]

Schmitz, Marcus T. and Al-Hashimi, Bashir M. «Low Power Process Assignment for Distributed Embedded Systems using Dynamic Voltage Scaling», *in Proceedings IEE Hardware-Software Co-Design, pages 7/1-7/4.*

[Mi97]

Microsoft, «OnNow: the evolution of the PC platform» , *http://www.microsoft.com/hwdev/pcfuture/ONNOW.HTM, Août 1997.*

[MP96]

Enrico Macii, Massoud Pedram, Fabio Somenzi, Tutorial: «High level Power modeling, estimation and optimization», in proceedings of *DAC 1997 June 9-13, 1997 Anaheim Convention Center Anaheim, California*

[MR94]

Renu Mehra, Jan Rabaey, «Behavioral Level Power Estimation and Exploration,» *Proc.First International Workshop on Low Power Design, Napa Valley, CA, pp. 197-202, April 1994..*

[MS01]

Thomas L. Martin and Daniel P. Siewiorek, «Nonideal Battery and Main Memory effects on CPU Speed-Setting for Low Power», *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.9, No.1, February 2001.*

[MS99]

Thomas Martin and Daniel Seiwiorek, «Non-Ideal Battery Behavior and Its Impact on Power Performance Trade-offs in Wearable Computing», *in Proceedings of the 1999 International Symposium on Wearable Computers, San Francisco, CA, October 18-19, 1999; pp. 101-106.*

[MT89]

T. Murata. «Petri nets : properties, analysis and applications». *Proceedings of the IEEE 77,4, April 1989, 541-580.*

[NF95]

F.N. Najm, «Power estimation techniques for integrated circuits», *in Proc. Int. Conf. Computer-Aided Design, pp.492-499, nov. 1995.*

[NJ02]

Nathalie Julien, Habilitation à diriger les recherches, mémoire «Méthodes de conception faible consommaiton des circuits VLSI», Habilitation à Diriger les Recherches, *soutenue le 11 Décembre 2002, au LESTER, Université de Bretagne Sud.*

[NM96]

R. Niemann, P. Marwedel. «Hardware/software partitionning using integer programming», in *Proc. European Design and Test Conference. 1996.*

[OAK]

VVF3500 DSP Core User Manual, VLSI technology, Inc.

[PA99]

Thèse, A. Pegatoquet, «Méthodes d'estimation de performance logicielle: application au développement rapide de code optimisé pour une classe de processeur DSP», *Université de Nice Sophia-Antipolis, Octobre 1999.*

[PB99]

Trevor Pering, Tom Burd, and Robert Broder, «Dynamic voltage scaling and the design of a low-power microprocessor system» 1999 (*http://www.infopad.eecs.berkeley.edu/~pering/lpsw*).

[PC95]

Scott R. Powell, Paul M. Chau, «A Model for estimating Power Dissipation in a class of DSP VLSI Chips», *in proceedings of IEE Trans. Circuits Syst. 36, 6(June), 646-650.*

[PG02]

Maurizio Palesi, Tony Givargis, «Multi-objective Design Space Exploration Using Genetic Algorithms», *International Conference on Hardware Software Codesign, in Proceedings of the tenth International Symposium on Hardware/Software codesign pp. 67-72, CODES, 2002.*

[PK89]

P.G. Paulin, J.P. Knight, «Force-Directed Scheduling for the Behavioral Synthesis of ASICs», *IEEE Transactions on Computer-Aided Design, Vol. 8, No.6, pp. 661-679, June 1989.*

[PS01]

Pillai and Kang G. Shin, «Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems,» *in Proc. of 18th Symposium on Operating Systems Principles, Banff, Canada, October, 2001.*

[PS98]

Philip Endecott, Stephen Furber: «Modelling and Simulation of Asynchronous Systems Using the LARD Hardware Description Language», *in proceedings of the 12th European Simulation Multiconference Manchester United Kingdom, 16-19, 1998.*

[QX02]

Gang Quan Xiaobo Sharon Hu, «Minimum Energy Fixed-Priority Scheduling for Variable Voltage Processors», *in proceedings of Design Automation Test in Europe, pp. 782-787.*

[RD96]

A. Raghunathan, S. Dey and N. K. Jha, «Register-transfer level estimation techniques for switching activity and power consumption», *in Proceedings of ICCAD 96, pp 158-165, Nov. 1996.*

[SB02]

Tajana Simunic, Stephen Boyd, «Managing Power Consumption in Networks on Chips», *in proceedings DATE 2002, pp110-116,* 8 Mars 2002 Paris, France.

[SD95]

C. Su, A. Despain, «Cache Design Trade-offs for Power and Perfermance Optimisation: A case Study» *in proceedings of International Symposium. On low Power Electronic and Design, pp63-68, 1995.*

[SK01]

Dennis Sylvester, Himanshu Kaul, «Future Performance Challenges in Nanometer Design», *in proceedings of Design Automation Conference 2001 June 18-22, 2001 Las Vegas p. 3.*

[SJ01]

P. Shivakumar, N. Jouppi, «Cacti 3.0: An Integrated Cache Timing, Power and Area Model», *Technical Repport, Compaq, Western Reaserch Laboratory, august 2001.*

[TJ01]

Taku Uchino, Jason Cong, «An Interconnect Energy Model Considering Coupling Effects», *in proceedings of the 38 th Design Automation Conference, p. 555, Las Vegas, 2001*

[TM96]

Vivek Tiwari, Sharad Malik and Andrew Wolfe, Mike Tien-chien Lee «Instruction level power analysis and optimization of software», *Journal of VLSI Signal Processing Systems, Vol. 13, No. 2, August 1996.*

[Tr00]

«The Technology Behind Crusoe Processors Low-power x86-Compatible Processors Implemented with Code Morphing» *Software Alexander Klaiber Transmeta Corporation January 2000.*

[TW95]

Ti-Yen, Wayne Wolf, «Sensitivity-Driven Co-Synthesis of Distributed Embedded Systems», *in proceedings of the 8 th international Symposium on System Synthesis, pp. 4-9, Cannes, France, 1995.*

[WW99]

Zhao Wu and Waine Wolf, «Iterative Cache simulation of embedded CPUs with trace stripping», *in Proc. IEEE Int'l Workshop on Hardware/Software Codesign (CODES), May 1999.*

[XN94]

M. G. Xakellis, Farid. N. Najm, «Statistical estimation of the switching activity digital circuits», *31ST ACM IEE Design Automation Conference : San Diego, vol. 1, p. 728-733, 1994.*

[ZC03]

Ying Zhang and Krishnendu Chakrabarty, «Energy-aware Adaptative Checkpointing in Embedded Real-Time Systems», *in proc. Design Automation Test in Europe 2003 Munich, Germany, March 3-7, 2003, p.918-923.*

ANNEXE :

Annexe : Modèles de consommations des DSP OAK et PALM

Cette annexe présente les modèles de consommation des architectures des DSP OAK et PALM qui ont été introduits dans l'outil d'estimation VESTIM. Ainsi, nous présentons brièvement la méthode utilisée pour caractériser en consommation ces processeurs puis leurs modèles de consommation obtenus.

Pour ce faire, il est nécessaire de quantifier par expérimentation les différentes sources responsables de la consommation. Par conséquent, une étape préalable consiste à effectuer des mesures précises qui ont été menées sur les deux DSP OAK et PALM.

En raison de leurs architectures hétérogènes, les DSP peuvent avoir des comportements en consommation assez complexes. Leurs multiples possibilités de chemins de données conduisent à développer des méthodes de mesures de la consommation au niveau de leurs unités fonctionnelles.

Une particularité des DSP est que leur architecture est en correspondance avec leur jeu d'instructions, c'est-à-dire que le parallélisme qu'ils supportent au niveau assembleur est obtenu par l'activation parallèle d'unités fonctionnelles des processeurs. Il s'agit par exemple des générateurs d'adresse et des opérateurs de calcul dans une instruction de multiplication/accumulation. En ce sens, ils possèdent un modèle d'instruction de type VLIW.

Par rapport aux processeurs à usage général, les DSP ont également d'autres particularités. Par exemple, les registres sont souvent dédiés et distribués au niveau des unités fonctionnelles. Ils intègrent des opérateurs

et des modes d'adressage spécifiques permettant d'accélérer l'exécution de certaines fonctions de traitement du signal. De nombreux DSP opèrent sur des nombres en virgule fixe et possèdent une largeur de données adaptée à une classe d'algorithme (traitement audio par exemple). Les DSP privilégient un parallélisme de la mémoire plutôt qu'une mémoire rapide hiérarchisée par l'intermédiaire de caches.

1•Méthode de mesures

1.1.Démarche.

Etant donnée l'approche utilisée par VESTIM qui repose sur l'ordonnancement d'opérations de niveau sémantique correspondant aux unités fonctionnelles du processeur, nous avons décidé de modéliser la consommation des DSP en fonction des chemins de données. Pour toutes les unités fonctionnelles du processeur, il s'agit de définir un modèle de consommation. Pour chaque pseudo-instruction agencée par VESTIM, il est ainsi possible de calculer une estimation de la consommation par sommation des contributions de chaque unité.

1.2.Approche expérimentale.

Pour obtenir une mesure de consommation des unités fonctionnelles du processeur, on opère de manière différentielle, par exemple en considérant deux instructions dont les ensembles d'unités fonctionnelles activées ne diffèrent que par l'unité recherchée. Pour estimer la puissance consommée instruction par instruction, la méthode expérimentale consiste à mesurer le courant entre le CPU et l'alimentation avec un ampèremètre [TM96].

On fait une mesure pour chaque instruction individuellement avec les différents modes d'adressage et différentes valeurs de données. Pour avoir une moyenne de la mesure due à l'instruction elle-même, on construit une boucle où est répétée un grand nombre de fois cette seule instruction (environ 150 instructions identiques) pour négliger les effets dus aux instructions de gestion de boucle. On cherche ainsi à atteindre une valeur stable. Afin de mesurer l'*overhead*, on fera de même avec des instructions agencées par paires. L'*overhead* est le coût d'inter-instruction lié aux changements de chemins de données entre deux instructions différentes.

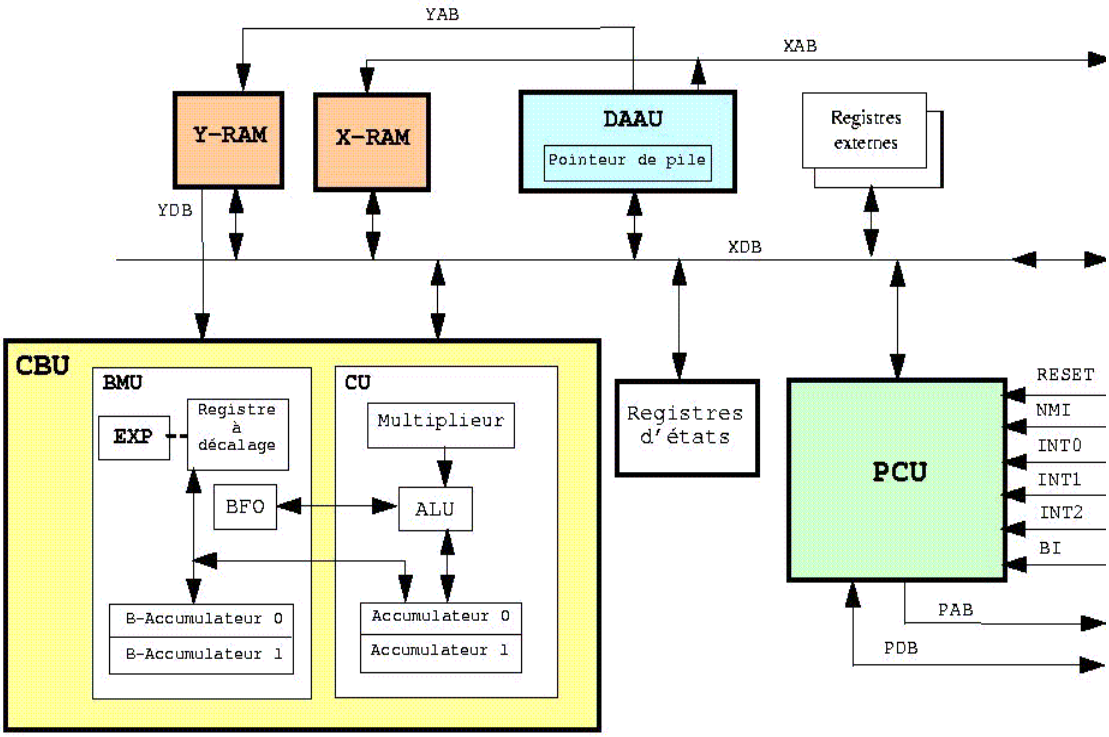
A l'aide de cette approche, nous avons mesuré la consommation de deux processeurs de traitement du signal : le OAK et le PALM. Nous décrivons brièvement dans les paragraphes suivants les architectures de ces DSP et les modèles de consommation déduits.

2•Modèle de consommation du DSP OAK

2.1.Architecture du DSP OAK.

Nous présentons son architecture dans la figure 57:

FIGURE 57. Architecture du DSP OAK



Registres :

- ALU : Accumulateurs a_0, a_1, sv:
- Multiplieur : x, y, p
- DAAU : r_0 à r_5

Les unités fonctionnelles:

Les unités fonctionnelles de ce processeur sont:

- Computation Unit (**CU**);
- Bit Manipulation Unit (**BMU**);
- Data Addressing Arithmetic Unit (**DAAU**);
- Program Control Unit (PCU);
- Mémoire de donnée interne en deux bancs : 16 Ko pour la mémoire X et 16 Ko pour la mémoire Y;

- Mémoire de programme interne : 64 Ko.
- Les bus:

  - «bus globaux» : canal de communication entre les unités,

  - «bus locaux»: canal de communication au sein d'une même unité.

Les bus «globaux» par lesquels les unités communiquent sont préchargés à "1" ce qui n'est pas le cas de tous les bus locaux (exemple: dans l'ALU les bus sur lesquels sont connectés les registres sv et y sont préchargés à "1" alors que pour a_X et p ils ne le sont pas). Ce choix technologique peut entraîner un comportement en consommation du DSP plus complexe.

Unité CU :

Cette unité de traitement est composée d'un multiplieur 16 x 16 bits, d'une Unité Arithmétique et Logique opérant sur des données de 36 bits et de registres.

Unité BMU :

Dans cette unité, sont regroupées les unités EXP de calcul d'exposant et Bit Field Operation qui permet d'effectuer les opérations qui modifient des bits d'un mot : mise à 1 (set), mise à 0 (reset) et complémentation (chng).

Unité DAAU :

Cette unité contient 6 registres de 16 bits appelés r_N [N = {0; 1; 2; 3; 4; 5}] utilisés dans les calculs d'adresse.

En résumé, on peut réaliser une association de ces unités à la majorité des instructions du processeur, par exemple :

- **CU/ALU :** Add, Sub, Xor, Or, And;
- **CU/Multiplier :** Mpy;
- **CU/ALU+Multiplier :** Mac;
- **BMU/BFO :** Rst, Set, Clr ;
- **BMU/Barrel Shifter :** Shfi ;
- **BMU/EXP :** Exp.

Cependant, pour certaines instructions, il n'est pas aisé d'associer des unités particulières : Mov (déplacer), MODr (modifie les registres r_N), Nop (pas d'opérations), BKREP (instruction de boucle).

Les mesures de courant ont été effectuées à une fréquence d'horloge de 40 MHz, une tension d'alimentation du coeur du processeur de 2,5 V et avec une alimentation de 3,3 V de la carte de mesures.

2.2.Modèles de consommation du OAK.

Le but ici est de trouver des modèles de consommation unité par unité [GP00]. Dans un premier temps, la méthode utilisée pour l'unité arithmétique et logique est présentée.

Les valeurs réelles des mesures ne sont pas fournies car le partenaire industriel de ce travail ne l'a pas autorisé. Seules des tendances sont présentées.

2.2.a.Etude de la consommation pour les instructions utilisant l'ALU :

- Premières mesures :

La mesure de courant peut varier de *x* mA à 3 *x* mA. Pour tous les types de registres et de modes d'adressage, la consommation est à peu près constante.

Observations

En prenant des mesures intermédiaires, on constate un saut d'énergie lorsqu'on passe de FFFF à FFF. Ceci s'explique par le fait que l'unité Arithmétique et Logique opère sur des données de 36 bits en binaire signé. Par conséquent, FFFF est étendu à F FFFF FFFF, contrairement à 0FFF.

- Activité de commutation de bits :

L'instruction add ne permet pas de mesurer l'activité de commutation. En effet, l'instruction répétée dans la boucle doit faire varier à chaque exécution un nombre de bits déterminé. Ainsi, l'étude de la variation de la consommation en fonction du nombre de bits modifié donne le coût de la transition d'un bit.

L'instruction Xor a_1,a_0 répond à cet objectif.

Avec a_1*=FFFF,* on mesure un courant de *X+4y+z mA***.**

Dans ce courant, le coût dû à la transition de 4 bits à 1 est de *y* mA, et *z* est le coût lié au préchargement. Par conséquent, le courant associé à l'activité de commutation d'un bit est de *y*/4 mA.

- Le modèle de consommation

On déduit de ces mesures le modèle suivant :

$I = I_{cst} + I_{chargement\ du\ bus} + I_{switching\ avtivity}$

avec $I_{chargement\ du\ bus} = N*I_{unit\ bus} + I_{extension\ de\ signe}$

- N représente le nombre de 0 présents dans la donnée.
- $I_{unit\ bus}$: courant unitaire (par bit) consommé en raison du préchargement des bus (sur 16 bits) et $I_{extension\ de\ signe}$ est le courant dû à l'extension de signe

dans le cas de nombres positifs. Le coût dû à l'extension de signe intervient pour le cas où le bit 15 est à 1.

- La composante constante : il s'agit de la consommation de l'instruction Nop qui correspond à la consommation de la lecture et du décodage de l'instruction qui est une opération présente pour chaque instruction.

Nous avons conforté ce modèle de consommation de l'ALU avec de nombreuses mesures qui donnent une erreur de 2,5% au maximum.

De ces mesures, nous déduisons les sources de consommation dues :

- **au préchargement des bus à un** : les bits mis à 0 consomment plus que ceux mis à 1. Ce préchargement permet d'augmenter la vitesse de l'horloge. En effet, forcer le bus au niveau logique 0 est plus rapide qu'au niveau logique 1.

Pour évaluer les effets du préchargement à un, le courant est mesuré pour une instruction exécutée en premier avec pour opérandes les accumulateurs, et une deuxième fois, avec pour opérandes les registres r_i, dont les bus connectés sont préchargés à un. La différence entre ces deux mesures représente la consommation due au préchargement des bus. Notons, que dans le pire des cas, la valeur du courant mesuré peut doubler quand ces registres sont utilisés. C'est donc un facteur important.

- **à l'activité de commutation des bits** (*switching activity*) : cette activité de consommation est due à la transition des bits de sortie variant à chaque cycle. Comme nous l'avons vu dans le chapitre sur la consommation, cette contribution est due aux charges et décharges des capacités des transistors, qui, réunies, correspondent à la capacité du circuit.

Ainsi, de toutes ces contributions, on déduit dores et déjà un modèle général du courant consommé : $I = I_{cst} + I_{préchargement} + I_{switching\ activity}$ où I_{cst} est le coût de base d'une instruction, mesuré sans activité de commutation.

2.2.b.Modélisation de la consommation de l'instruction MPY.

Dans les mesures suivantes, le premier opérande est placé dans le registre y qui est à l'entrée du multiplieur et le deuxième opérande est le registre d'adressage r_1. Le résultat de l'instruction Mpy op_1,op_2 est placé dans le registre p.

Nous considérons à nouveau le modèle suivant :
$I = I_{cst} + I_{préchargement} + I_{switching\ activity}$.

Après analyse des mesures, nous avons pris en compte le poids des bits (uniquement pour le deuxième opérande dont le contenu transite par le bus global). On a cherché une valeur moyenne par groupe de 4 bits à un.

Mesure de Mpy y=0,r1 sans activité de commutation.

nb de bits à 1	I_{moyen} (mA)
de 0 à 4	I1
de 4 à 8	I2 (<I1)
de 8 à 12	I3 (<I2)
de 12 à 16	I4 (<I3)

Regardons $I_{switching\ activity}$:

Le multiplieur est un multiplieur de Booth, ce qui entraîne que seul le second opérande a une influence sur la consommation. Nos mesures ont montré que la valeur du courant dû à l'activité de commutation est pondéré comme le montre le tableau suivant :

bits hors signe	*x* mA
bit de signe	5 *x* mA

- Vérification du modèle :

L'erreur du modèle pour le multiplieur pour un grand nombre de mesures avec différents opérandes est au maximum de 8%.

2.2.c.Modélisation de la consommation de l'instruction Mov.

- ***Le modèle***

On prend en compte le même préchargement que pour le CU/ALU et l'activité de commutation est mesuré en mettant dans la boucle les instructions Mov r_1, 0x0 et Mov r_2, 0x0 alternées avec les valeurs du tableau ci-dessous. On obtient :

r_1	r_2	I (mA)
0	8000	I1
0	0001	I2<I1

Comme pour la multiplication, on a un poids différent sur le signe.

- ***Vérification du modèle :***

L'erreur moyenne est ici de 2%. Cette faible erreur provient du faible courant dû à l'activité de commutation et à la modélisation précise du courant de préchargement.

2.2.d.Coûts inter-instructions

- **Méthode de mesure** :

On crée une boucle où deux instructions différentes sont alternées. Cette boucle est exécutée pour différentes valeurs des données.

A ces mesures, on retranche la valeur moyenne des coûts des instructions mesurées séparément. Le surcoût constitue le coût inter-instructions ou *overhead* : $I_{ov} = I_{instruction1,instruction2} - \frac{I_{instrucuion1} + I_{instruction2}}{2}$.

Les mesures donnent une grande variation de ce courant : de *z* à 42 *z* mA.

Nous avons distingué ces variations en deux catégories :

- Par paires d'instructions de type identique, lorsque, par exemple, seul le mode d'adressage varie : l'*overhead* est constant et généralement de faible valeur. Il dépend du type des opérandes,
- Par paires d'instructions différentes deux à deux : selon le type des opérandes, l'*overhead* est constant, ou dépend du nombre de bits à un, mais dans ce cas, la variation est moins importante que pour le préchargement des bus.

Toutefois, dans toute notre étude précédente, lors des tests des instructions, on a considéré que l'*overhead* est constant. Cela semble justifié par le fait que pour une paire d'instructions identiques, on a obtenu des *overheads* constants.

2.2.e.En résumé.

En résumé, l'ensemble de toutes les mesures réalisées sur la majorité des instructions de DSP a permis de valider le modèle $I_{constant}+I_{switching\ activity}+I_{préchargement\ du\ bus.}$

- ***Recommandations d'écriture du code***

A travers ces comportements, on peut déduire des règles d'écriture du code en assembleur afin de minimiser l'énergie et la puissance dissipées.

Pour les instructions simples, il est préférable de privilégier les utilisations des ressources suivantes :

- **pour l'ALU** : les accumulateurs et les registres a0, a1 et p et les modes d'adressage direct court, et indirect,
- **pour le multiplieur** : comme la consommation est principalement déterminée par le nombre de 1 présents dans le second opérande, il est préférable de mettre la valeur ayant le plus grand nombre de bits à 1 dans le premier opérande,
- **pour Mov** : utiliser préférentiellement a0, a1 et le registre p, mais ceci contraint fortement l'écriture du code,

- **pour minimiser les *overheads***, en général, il est préférable de faire suivre des instructions activant les mêmes unités fonctionnelles, ou à la rigueur d'utiliser les mêmes modes d'adressage ou les regroupements les moins coûteux comme par exemple la séquence Mov-Mpy.

Toutes les mesures effectuées ne concernent que la consommation du coeur, sans tenir compte de tous les coûts d'accès à la mémoire.

Toutefois, on a vérifié que l'accroissement de courant induit par un accès à la mémoire X est de 6 %, alors qu'il est de 14 % dans le cas de la mémoire Y.

Notons que ces mémoires sont intégrées sur la même puce que le processeur.

Dans le paragraphe suivant, nous présentons le comportement en consommation du processeur de génération suivante, le PALM.

3•Modèle de consommation du DSP PALM

Le PALM constitue la génération suivante des DSP par rapport au OAK. Son architecture (figure 58) se différencie principalement par ses bus qui ne sont plus pré-chargés à 1, sa structure pipeline à 5 étages et sa capacité à exécuter jusqu'à 7 opérations en parallèles.

3.1.Architecture du PALM.

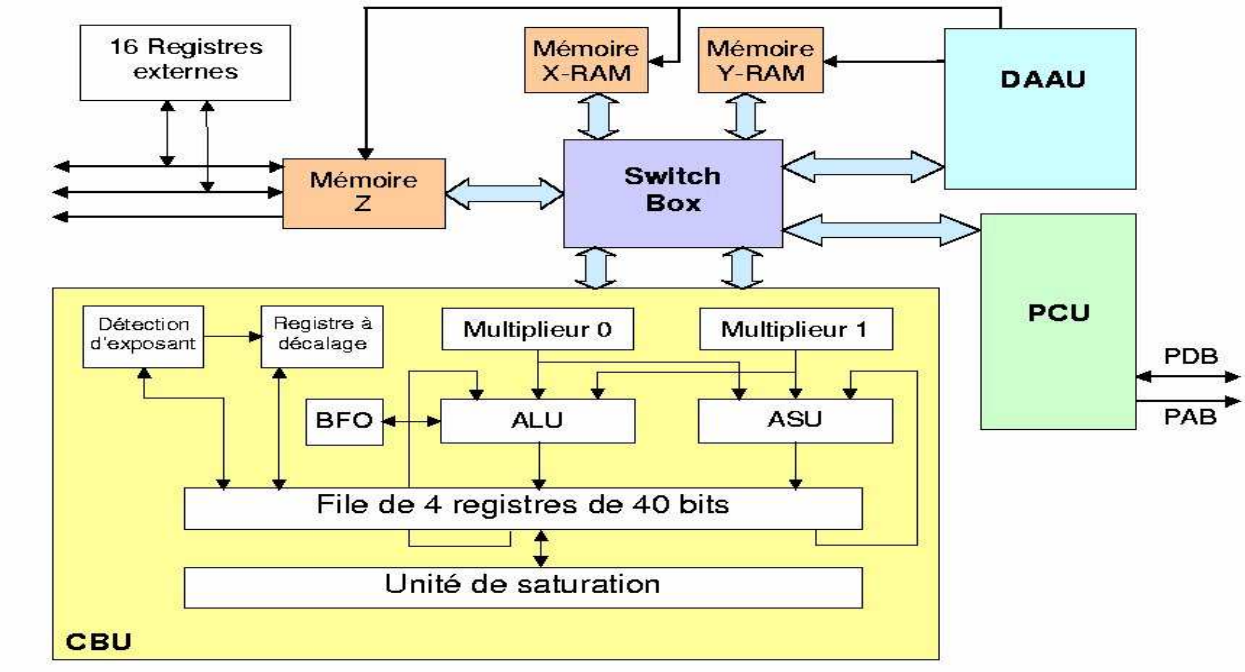
C'est une architecture pensée performance et basse consommation. Il existe trois versions du DSP PALM qui diffèrent selon la taille des mots de données: 16, 20, 24 bits.

Son architecture est composée des cinq unités suivantes capables d'opérer en parallèle:

- ***CBU**, Computation and Bit Manipulation Unit*
- ***DAAU**, Data Address Arithmetic Unit*
- ***PCU**, Program Control Unit*
- ***OFU**, Operand Fetch Unit*
- ***IDU**, Instruction Decode Unit*

Sur la figure 58 est illustrée l'architecture du PALM, il s'agit ici de la version sur 16 bits.

FIGURE 58. **Architecture du PALM**



- CBU

Le CBU est composé de trois unités de calcul, deux unités Computation Unit (CU) et une unité Bit Manipulation Unit (BMU).

Le CU comporte deux unités indépendantes de multiplication, une unité de Barrel Shifter, une unité Exponent, une unité ALU à trois entrées, une unité Add/ Substract (ASU) à trois entrées et quatre accumulateurs arrangés en file de registres. Toutes ces unités travaillent sur des mots de 40 bits.

Le BMU est constitué d'une unité Barrel Shifter, d'une unité Exponent, toutes les deux sur 40 bits, et de l'unité BFO (Bit Field Operation) qui effectue des traitements sur des champs de bits.

- Les multiplieurs.

Les entrées sont étendues en signe ou en zéro à 17 bits (étendu en zéro dans le cas d'une multiplication non signée et étendu en signe dans le cas contraire). Les sorties des multiplieurs sont codées sur 33 bits, étendues en signe à 37 bits.

- Les ALU/ASU

Chaque unité a trois entrées. Le résultat de signe étendu ou étendu en zéro atteint 40 bits. Toutes les opérations dans ces unités s'effectuent en complément à deux.

- Barrel Shifter

Cette unité effectue le décalage de bits (shift) dans les deux modes logique et arithmétique.

- DAAU

Il s'occupe de la gestion des modes d'adressage.

Ainsi, il peut générer jusqu'à 4 adresses : deux adresses courantes avec leurs adresses consécutives. Pour se faire, il possède deux unités de génération indépendantes : BANKI et BANKJ. Les modes d'adressage supportés sont les modes direct, indirect, bit-reverse, indexé, et stack.

- PCU

Il gère les interruptions et les branchements.

3.2.Modèles de consommation du PALM.

On garde la même démarche expérimentale que pour le OAK. Il est à signaler que la mesure a été automatisée dans ce cas. Le banc de test est relié à un PC qui pilote la fréquence et la tension d'alimentation. Ainsi, on balaye automatiquement différentes tensions et fréquences et on relève lors de l'exécution de chaque boucle la valeur du courant consommé. Nous donnons les résultats à 1,8V d'alimentation, mais le courant consommé varie peu entre cette tension d'alimentation et 2,6V. De même, nous faisons varier la fréquence de 20 à 40 MHz.

Contrairement au OAK, les bus du PALM ne sont pas préchargés à 1. Le terme correspondant dans le modèle de consommation du OAK ne sera donc pas présent dans celui du PALM.

Ici, le modèle de consommation sera $I = I_{cst} + I_{switching\ activity}$.

3.2.1.Instructions simples ou non parallèles.

Il s'agit des instructions qui n'invoquent par exemple que l'ALU ou que l'ASU mais pas les deux unités en même temps.

3.2.1.a.Consommation indépendante du signe d'un des opérandes.

Etant donné qu'il n'y a pas de variations en fonction de la valeur des données, on déduit que l'activité de commutation peut être négligée.

Selon les types d'instructions et de modes d'adressage, on distingue trois types de valeurs :

Tableau 22: Consommation indépendante du signe d'un des opérandes

Catégories	Fréquence de 20MHz (courant consommé)	Fréquence de 40MHz (courant consommé)
A	1*x*	2*x*
B	1,5*x* - 2*x*	3*x* - 3,75*x*
C	2,5*x*	5*x*

On retrouve dans ces catégories des instructions avec des modes d'adressage tels que :

Tableau 23: Consommation indépendante du signe d'un des opérandes, détails des catégories

catégorie A	• pour les instructions arithmétiques : adressage avec accumulateurs, • pour les instructions logiques avec mode d'adressage direct, • Certaines instructions invoquant le multiplieur
catégorie B	• pour les instructions arithmétiques sur les registres ou utilisant un mode d'adressage immédiat (comme Sub registre, registre ou Cmp immédiat), • pour les instructions logiques avec registre, accumulateur et adressage direct (Xor direct prenant une valeur négative), • d'autres encore: MPY quand les deux opérandes sont des registres; Mov,
catégorie C	• pour les instructions logiques avec registre, accumulateur (Xor r_0,a_1), • d'autres encore: Nop,

3.2.1.b.Consommation dépendante du signe d'un des opérandes.

Ici apparaît l'activité de commutation puisque le courant consommé varie avec la valeur des données. Dans ce cas, on constate deux catégories de valeurs:

Tableau 24: Consommation dépendante du signe d'un des opérandes

Catégories	Fréquence de 20MHz (courant consommé)		Fréquence de 40MHz (courant consommé)	
	données positives	données négatives	données positives	données négatives
D	1,25*x*	2,5*x*	2,5*x*	5*x*
E	1*x*	2*x*	2*x*	3,75*x*

Tableau 25: Consommation dépendante du signe d'un des opérandes, détails des catégories

catégorie D	• pour les instructions arithmétiques avec adressage indirect ou indexé (du type (r_1+r_7)), • des instructions logiques de mode d'adressage indexé ou utilisant des accumulateurs, • d'autres encore invoquant le multiplieur.
catégorie E	• pour les instructions arithmétiques avec mode d'adressage indexé et indirect, • pour les instructions logiques avec les adressages du type $(r_0),(r_4),a_0$.

La première partie des résultats de ce paragraphe (quand l'opérande est positif) est la même que celle des instructions correspondantes du paragraphe précédent (consommation indépendante des valeurs de données), ce qui confirme la présence d'une activité de commutation pour les valeurs négatives.

Analysons les raisons qui font que certaines instructions et modes d'adressages réagissent au bit de signe.

Prenons par exemple l'instruction multiplication/accumulation, Mac :

A la sortie du multiplieur, le registre de résultat est sur 37 bits. Il est étendu en signe sur 3 bits en entrée de l'ALU. Donc il n'y a pas de différence d'activité notable de commutation au sein de l'ALU entre nombres positifs ou négatifs.

Prenons à l'inverse de ce comportement, celui de l'instruction de comparaison, Cmp qui effectue une soustraction entre les deux opérandes. Considérons Cmp #0x00FF,a_0 indépendant d'un des signes des opérandes et Cmp $(r_0),a_0$ qui

dépend des deux. Or, dans le premier cas, les registres sont de même taille, donc il n'y a pas de surcoût. Dans le deuxième cas, il faut amener la valeur lue en mémoire pointée par r_0 de 16 à 40 bits pour pouvoir effectuer la soustraction. Il y a donc une extension en signe ou en zéro.

Quand le bit de signe de r_0 est à 0, il y a une extension en zéro et les deux instructions ont la même consommation, par contre, si le bit de signe est à 1, il y a une extension de signe et la consommation augmente.

Comme on pouvait s'en douter, la consommation est fortement influencée par les chemins pris par «l'ensemble des acteurs». Pour comprendre la consommation de chaque instruction, il est nécessaire de connaître à l'avance le chemin de données suivi et également si une extension de signe est nécessaire.

Toutefois, nous n'avons pas pu expliquer certains résultats, comme par exemple le cas de l'instruction Mpy y_0,0x00FF, indépendant du signe de y_0 et Mpy y_0,0x0000qui en dépend!

Remarque: le Xor consomme plus quand son bit de signe est à 0, alors que les autres instructions se comportent de manière inverse. Qu'en déduire pour l'activité de commutation ?

- **Modélisation**

Malgré les cas particuliers cités précédemment, on peut donc modéliser la consommation du PALM par autant de courants constants que de catégories identifiées, dépendantes des instructions et des modes d'adressages.

3.2.2.Coûts inter-instructions.

3.2.2.a.Consommation indépendante des bits réservés au signe d'un des opérandes.

Comme le courant ne varie pas avec les données, il n'y a pas de surcoût de courant dû à l'activité de commutation, mais seulement le coût de base de chaque instruction auquel s'ajoute l'*overhead*. Les valeurs mesurées sont très variables. Nous avons déduit des mesures, les trois cas suivants :

Tableau 26: Consommation indépendante des bits réservés au signe d'un des opérandes

Courant consommé dû à la succession de deux instructions	Fréquence de 20MHz (courant consommé, unité)	Fréquence de 40MHz (courant consommé, unité)
identiques	2*x*	2,5*x*
différents sans Mov	2,5*x*	5*x*

Tableau 26: Consommation indépendante des bits réservés au signe d'un des opérandes

Courant consommé dû à la succession de deux instructions	Fréquence de 20MHz (courant consommé, unité)	Fréquence de 40MHz (courant consommé, unité)
différents avec Mov	5*x*	7*x*

On fait une erreur comprise entre 11% et 33%, cette dernière valeur étant dans le pire cas, mais il est difficilement envisageable d'obtenir un modèle plus précis autrement qu'en gardant chacun des résultats dans une librairie, par exemple.

3.2.2.b.Consommation dépendante de la valeur des bits réservés au signe d'un des opérandes.

Ici apparaît l'activité de commutation puisque le courant consommé varie avec la valeur des données. Il s'agit des deuxièmes parties du tableau 24, dans le cas des catégories D et E, c'est-à-dire les parties pour les valeurs de données négatives.

On obtient par exemple le résultat suivant :

Tableau 27: Consommation dépendante de la valeur des bits réservés au signe d'un des opérandes

Courant consommé dû à la succession de ces instructions	Fréquence de 20MHz (courant consommé, unité)	Fréquence de 40MHz (courant consommé, unité)
Add a_0,a_1 - Mpy r_1	2,5*x*	4,4*x*
Xor a_0, a_1 - Mov r_0,r_1	5,6*x*	7,5*x*
Sub - Sub	1,4*x*	3*x*

On a une erreur pouvant aller jusqu'à 20% dans le pire des cas, mais on peut constater que l'erreur est faible en moyenne.

- **Modélisation de l'*overhead***

L'*overhead* est plus important quand les deux instructions sont différentes que lorsqu'elles sont identiques mais de mode d'adressage différent. On peut modéliser les *overheads* par des constantes différentes selon les instructions et le mode d'adressage.

3.2.3.Instructions parallèles.

Ces instructions utilisent dans le même cycle plusieurs unités du CBU, par exemple deux Mac.

3.2.3.a.Consommation indépendante du signe d'un des opérandes .

Il n'y a pas d'activité de commutation. On obtient deux catégories :

Tableau 28: Consommation indépendante du signe d'un des opérandes

Catégories	Fréquence de 20MHz (courant consommé, unité)	Fréquence de 40MHz (courant consommé, unité)
A//	2*x*	3*x*
B//	1,4*x*	2,5*x*

Dans ces catégories, on retrouve des instructions, comme celles du tableau 29.

Tableau 29: Consommation indépendante du signe d'un des opérandes, détails des catégories

Catégories	Exemples d'instructions
A//	And y_1, a_3 // Cmp r_0, a_1 Addl r_5, a_3 // addl (r_3), a_0
B//	Sub 0xFF, a_0 // Add 0xFF, a_1

3.2.3.b.Consommation dépendante du signe d'un des opérandes.

Ici apparaît l'activité de commutation puisque le courant consommé varie du fait des valeurs de données négatives. Les évolutions des consommations suivant les instructions en parallèle et les modes d'adressage peuvent être regroupées en deux catégories :

Tableau 30: Consommation dépendante du signe d'un des opérandes

Catégorie	Fréquence de 20MHz (courant consommé, unité)		Fréquence de 40MHz (courant consommé, unité)	
	données positives	données négatives	données positives	données négatives
C//	1,25*x*	1,5*x*	1,5*x*	3*x*

Tableau 30: Consommation dépendante du signe d'un des opérandes

Catégorie	Fréquence de 20MHz (courant consommé, unité)		Fréquence de 40MHz (courant consommé, unité)	
D//	1*x*	3*x*	2,4*x*	5 *x*- 5,6*x*

Dans ces catégories, on retrouve des instructions, par exemple celles du tableau 31:

Tableau 31: Consommation dépendante du signe d'un des opérandes, détails des catégories

Catégories	Exemples d'instructions
C//	Mpy y_0,(r_1) // Add y_1, (r_1+) Or (r_0),a_0 // Mov r_3,r_4
D//	Mpy y_0,(r_1) // Add y_1, (r_1+) Mpy (r_4),(r_0) // Mpy (r_4+),(r_0+) // Sub p_0,p_1,a_0

3.3.Recommandations pour l'écriture de code :

- utiliser les instructions simples les moins coûteuses, en veillant au mode d'adressage utilisé,
- veiller à ce que les enchaînements séquentiels d'instructions soient ceux pour lesquels les *overheads* sont les plus faibles,
- utiliser au maximum les instructions en parallèles, puisque les coûts sont bien inférieurs aux coûts cumulés des instructions simples équivalentes. Quelques tests ont montré qu'il est possible d'obtenir **un rapport 4 entre ces deux manières de programmer...** Ce DSP, comme ceux de sa génération, a été conçu pour améliorer les performances grâce au parallélisme. L'exploitation du parallélisme allège le problème de l'*overhead*.

Comme il est noté dans la littérature [JR,XN94] que la consommation due aux *overheads* est la plus significative, le PALM, grâce à son parallélisme important permet de diminuer l'impact des *overheads*.

En particulier, ici, nous avons pu noter dans nos mesures le comportement suivant :

- à tension donnée, la puissance consommée par cycle reste strictement constants en fonction de la fréquence,

- à fréquence donnée, le courant consommé varie peu en fonction de la tension d'alimentation. La puissance consommée par cycle augmente en moyenne de 50% lorsque la tension d'alimentation varie de 1,8V à 2,6V.

Comparé au OAK, le parallélisme du PALM permet l'exécution d'un plus grand nombre d'opérations par cycle. Combiné à une profondeur de pipeline plus importante, il est aussi possible de baisser la tension d'alimentation en gardant une vitesse d'horloge comparable. En effet, le parallélisme impose des restrictions au niveau assembleur. La majorité des instructions en parallèle doivent avoir les mêmes types d'adressage et des adresses qui se suivent. Ainsi, si l'on a par exemple comme opérande (r_1), l'instruction suivante, mise en parallèle, devra avoir ($r_1 + 1$) comme opérande. Ainsi, il n'y aura qu'à incrémenter l'adresse grâce à une des deux unités BANKI ou BANKJ. Ceci signifie que la lecture de r1 a lieu une seule fois pour le calcul de deux adresses.

4•Comparaison des consommations des deux types de DSP

Ce paragraphe présente les premiers enseignements sur la basse consommation que l'on peut déduire des études décrites dans les paragraphes précédents.

On peut tout de suite noter que l'optimisation consistant à baisser la tension d'alimentation est très performante, puisque dans notre cas, nous gagnons 50% en puissance en passant de 2,6V à 1,8V.

Incontestablement, la consommation du PALM est moins élevée que celle du OAK. Le PALM est évidemment plus performant (facteur de gain en puissance (mW) entre 3 et 5), il est capable d'exécuter jusqu'à sept instructions en parallèle, par cycle.

Des efforts pour diminuer la consommation ont été portés à divers niveaux :

- *modes d'adressage* : le PALM, contrairement au OAK, a des modes d'adressage imposés, avec une *switch box*, au lieu de multiplexeurs. Or, ceux-ci sont plus économiques en énergie par rapport à un bus global du fait des plus petites capacités induites.
- *gated clock* : le PALM possède 150 domaines d'horloge. Donc, les diverses parties du circuit ne sont activées qu'en fonction des besoins des instructions. Il y a donc moins d'énergie dissipée que dans le OAK, dépourvu de ce système.
- *flip-flop contre latch* : le coût des *overheads* est plus important pour le OAK que pour le PALM. Le PALM est à technologie flip-flop, alors que le OAK

est à technologie latch. Le latch reste «transparent» pendant toute la durée d'un niveau d'horloge, ce qui entraîne une propagation de l'activité de commutation au delà du latch, même si cette activité n'est pas utile à l'exécution de l'instruction. A l'inverse, les sorties d'une bascule flip-flop restent stables entre deux fronts d'horloge supprimant cette commutation nuisible.

Le OAK et le PALM appartiennent à deux générations successives de DSP présentant ainsi l'avantage de nous fournir deux exemples de processeurs ayant des comportements en consommation différents.

Ces modèles sont précis et présentent un taux d'erreurs acceptable. La méthode de mesure mise en œuvre permet de déterminer la consommation de chaque unité fonctionnelle qui participe à la consommation des instructions. Ces contributions sont obtenues par analyse différentielle des consommations des instructions assembleur des processeurs. Une fois connues ces contributions, nous les avons intégrées dans l'outil VESTIM avec les différents calculs de courants définis dans les modèles ci-dessus afin d'évaluer la consommation liée à un code complet.

Résumé : Les systèmes embarqués représentent une part de plus en plus importante du marché semiconducteurs. Les systèmes embarqués visent des applications plus gourmandes en capacité de calcul, augmentant du même coup la surface de silicium et l'énergie dissipée. Un des problèmes de la conception système est le patitionnement d'applications qui requiert l'utilisation de méthodes complexes. En effet, le partitionnement sous contraintes de temps, basé sur un algorithme d'ordonnancement avec un objectif de minimisation de la surface de silicium ou de la consommation est un problème NP-difficile. Ce travail de thèse étudie la prise en compte de la consommation (énergie et pic de puissance) lors de la conception globale de systèmes autonomes. Une première étude consiste à estimer la consommation des divers composants d'une architecture SoC. Puis, nous nous sommes intéressés aux deux étapes principales des méthodes de partitionnement : l'allocation et l'ordonnancement. En particulier, la technique d'ajustement conjoint de la tension et de la fréquence est considérée dans l'ordonnancement pour minimiser l'énergie. A la suite de ces optimisations, une gestion des modes basse consommation est réalisée, ayant pour objectif de mettre les processeurs en état de repos ou repos profond dès que la possibilité se présente, ce changement de mode permettant de gagner en consommation. Ce travail a été testé sur divers exemples, comme une application de détection de mouvement sur fond d'images fixes pour caméra embarquée.

Mots-clés : Gestion de l'énergie et de la puissance, ajustement conjoint de la tension et de la fréquence, modes basse consommation, conception logiciel/matériel, ordonnancement.

Abstract: Embedded systems take an important part of the computer market. Telecommunication applications integrates more and more functionalities, so power consumption increases and the battery lifetime becomes a serious limitation. In this thesis, consumption of different parts of SoCs was studied. Particularly, this work deals with consumption/time optimized methods of HW/SW codesign. Allocation minimizing energy is presented, and then, a refinement step is executed to optimize the schedulings by applying the Dynamic Voltage Scaling/ Dynamic Frequency Scaling technique (DVS/DFS). An other step in this work concerns the exploitation of low power modes of processors (run, sleep, idle). Finally, this work studies how methods of HW/SW codesign can integrate more time and area, the consumption as constraint.

Key words: Power Management, dynamic Voltage Scaling/ Dynamic Frequency Scaling, low power modes, HW/SW codesign, scheduling.